An efficient approach for sequence matching in large DNA databases

Jung-Im Won and Sanghyun Park

Department of Computer Science, Yonsei University, Korea

Jee-Hee Yoon

Division of Information Engineering and Telecommunications, Hallym University, Korea

Sang-Wook Kim

College of Information and Communications, Hanyang University, Korea

Received 14 March 2005
Revised 8 August 2005

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 respect of 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 two parts: the primary part realizes the trie as binary bit-string representation without any pointers, and the secondary part helps fast access to the trie's leaf nodes 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.

Correspondence to: Jung-Im Won, Department of Computer Science, Yonsei University, 134 Sinchon-dong, Seodaemun-gu, Seoul 120–749, Korea. E-mail: jiwon@cs.yonsei.ac.kr

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 A, C, G, and T known as the nucleotide bases. In molecular biology, a basic way to understand a newly discovered DNA sequence is to infer its characteristics by referring to other DNA sequences whose characteristics have been identified [1]. DNA sequence matching is an operation that finds DNA sequences whose base arrangement is similar to that of a DNA sequence given in a query from a DNA database. Therefore, DNA sequence matching enables molecular biologists to understand the characteristics, which are the role, evolution, and chemical structure, of the new DNA sequence, thereby being a crucial operation in molecular biology [2, 3].

The problem of DNA *subsequence matching* is formulated as follows. Given a DNA 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 not larger than T. A DNA database contains a huge volume of DNA sequences. In 1992, GenBank [4], which is a well-known DNA database, initially contained around 100 million bases of 78,000 DNA sequences. However, by 2004, it included over 44,575 million bases of 40 million DNA sequences. Historically, the database has roughly doubled in size every 14 months, and the rate of increase is also growing gradually. Therefore, efficient processing of DNA subsequence matching is fairly important in such large databases.

BLAST [5, 6] is a de facto standard tool widely used by molecular biologists to perform DNA subsequence matching. To test the homology of two DNA sequences, it first searches for pairs of seeds, which are short words of a fixed length obtained from the two sequences, and then extends them to higher-scoring regions. BLAST provides high performance by using a heuristic algorithm. However, it does not guarantee accuracy; i.e. it may lose some true answers. The most popular method that guarantees accuracy is to combine the Smith-Waterman algorithm and the sequential scan [7]. However, it is unsuitable for wide-spread use for practical applications since its CPU and disk access costs are high in a large DNA database.

The size of DNA databases increases considerably so that fast indexing is essential to support DNA subsequence matching efficiently. The suffix tree [8] has been known to be a good index structure for DNA subsequence matching. The suffix tree is a compressed digital trie built on all 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 it already has various approximate matching algorithms proposed in the literature [9–12].

The elapsed time of subsequence matching when using such algorithms, however, increases seriously as a query sequence length or a tolerance increases. To alleviate this problem, Navarro and Baeza-Yates proposed a hybrid indexing method [13, 14], which divides a query sequence into multiple shorter pieces, performs their subsequence matchings with a smaller tolerance, and then merges the results thus obtained. Also, Meek et al. [15] suggested a method which applies the best-first (A*) search [16] in traversing a suffix tree. They reported that this method gives a good performance in subsequence matching, comparable to that of BLAST in cases where query sequences are not too long.

The suffix tree still has the drawbacks listed below, due to its structural features.

(1) **Storage space.** The suffix tree requires a large storage space; it is often some ten times larger than a database [13, 17, 18]. Hunt et al. [12] 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 access [18]. Thus, overall search performance deteriorates in DNA databases [19].
(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 difficulty in employing a page as a storage unit due to its structural characteristics [8, 20]. Thus, the suffix tree has a problem in integrating itself seamlessly with DBMS.

In this paper, we propose a novel indexing method that supports DNA subsequence matching efficiently as well as resolving the drawbacks of the suffix tree mentioned above. The proposed index basically adopts a trie [8, 21] as its primary conceptual structure and realizes the trie by pointerless binary bit-string representation. In addition, it employs a multi-dimensional index as a secondary structure, which supports fast access to the target leaf nodes when traversing the trie. With these characteristics, the proposed index successfully resolves all the problems of the suffix tree in respect of storage space, search performance, and integration with DBMS. This paper also proposes algorithms that effectively process exact and approximate DNA subsequence matchings based on the proposed index. Through extensive experiments, we verify the effectiveness of our approach quantitatively. The results reveal that, compared with the previous ones, our approach requires smaller storage space and achieves up to some ten times better performance in DNA subsequence matching.

The paper is organized as follows. Section 2 briefly reviews previous research efforts related to DNA sequence matching. Section 3 proposes a novel index structure and discusses its characteristics. Section 4 presents query processing algorithms for exact and approximate DNA sequence matching, which exploit the proposed index. Section 5 shows the superiority of our approach via performance evaluation with a series of experiments. Finally, Section 6 summarizes and concludes the paper.

2. Background

In this section, we present related work on DNA sequence matching. Section 2.1 reviews previous DNA subsequence matching methods by subdividing them into two categories: sequential scan-based and index-based ones. Section 2.2 introduces the suffix tree, an index structure known to be appropriate for subsequence matching.

2.1. Previous work

The Smith-Waterman algorithm [7] is a representative one for DNA subsequence matching, that basically assumes the sequential scan for accessing DNA sequences. It uses a dynamic programming technique and determines the optimal local alignment between two sequences S and Q so that their similarity score gets maximized. However, it requires a long processing time due to its high time complexity of $O(|Q|*|S|)$.

BLAST [5] is a heuristic algorithm that resolves the performance problem of the Smith-Waterman algorithm. It performs DNA subsequence matching via two steps as follows [2]. In step 1, BLAST extracts all the words of length k starting from every base position within a query sequence Q. As a result, we have $|Q| - k + 1$ words via this extraction. Next, it searches for all the *k-tuples*, each of which is exactly matched to some query word. These *k-tuples* thus obtained are called *seeds*. In step 2, BLAST extends a query word and its corresponding seed to find longer similar segment pairs whose dissimilarity is not larger than a given tolerance T. Finally, it returns subsequences similar to the query sequence as a query result by using the similar segment pairs.

Currently, BLAST is the most widely-used method for DNA subsequence matching since it obtains near optimal alignment among DNA sequences in a reasonable time. BLAST, however, has the following problems: its target database should be loaded into main memory for its processing; its search performance is inversely proportional to the database size; and its accuracy is affected by the length of a query word.

Recently, a variety of methods have been proposed for overcoming these problems [22, 23]. These methods also show low search performance in huge DNA databases because they basically employ the sequential scan for accessing the database. The performance of DNA subsequence matching can be improved by exploiting indexing mechanisms. Previous indexing mechanisms use the inverted index [24–26], the multi-dimensional index [27], and the persistent tree index [12, 15, 19].

The methods in references [24–26] 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 compressed inverted index [26] can reduce the index size; however, it has the drawback of low accuracy in subsequence matching.

The method proposed in reference [27] maps every subsequence into a point in multi-dimensional space by the wavelet transform, and then constructs a multi-dimensional index on those points. By using the index, it processes range queries and nearest neighbor queries. This method enjoys nice search performance owing to the relatively small size of the index. However, it does not allow similarity measures other than the edit distance, and also supports only the global alignment rather than the local alignment.

The suffix tree [8] is an index in the form of a persistent tree, and has been widely used in DNA subsequence matching. Hunt et al. [12] suggested an algorithm for constructing a disk-based suffix tree on huge DNA databases, and proposed an approximate subsequence matching algorithm that exploits the suffix tree. Sadakane and Shibuya [19] devised a subsequence matching method that uses a compressed suffix array [28] maintained within main memory. Meek et al. [15] proposed a subsequence matching method called OASIS, which combines the suffix tree and the best-first (A*) search scheme [16]. OASIS returns the answers in the order of the similarity score. It performs the optimal local alignment exactly the same as in the Smith-Waterman algorithm [7], but provides several ten times better performance. Also, it gives a high search performance comparable to BLAST where query sequences are not overlong. However, the suffix tree still suffers from three problems due to its structural characteristics:

(1) high storage overhead,
(2) low search performance, and
(3) difficulty in seamless integration with DBMS.

2.2. Suffix tree

The suffix trie is a data structure for storing all the suffixes of a given sequence [8]. In a suffix trie, each edge has a symbol belonging to the alphabet as its label. If a series of symbols on the edges along the path from the root to a node N is a suffix of a sequence to be indexed, the node N has a pointer to the offset from which the suffix starts in the sequence. A suffix trie is generalized to allow multiple sequences to be stored in the same trie. Figure 1 shows an example of a suffix trie constructed from two DNA sequences, S1 = ‘ACGT’ and S2 = ‘ACT’, where symbol ‘$’ represents an end marker for each suffix, and thus is used to uniquely identify suffixes. For instance, a suffix 'CGT' of S1 is stored in the path from the root to a leaf node labeled as (*S1,1*). Here, the leaf node contains a pair of <*sequence number, offset*> as its label.

The suffix tree is a compressed form of a suffix trie. A path where every node has a subsequent node as a unique child node in a suffix trie is represented as a single edge in a suffix tree. The label of this edge is the concatenated one of all the labels existing on that path in the suffix trie. If we concatenate all the labels on the path from the root to a leaf node, we obtain a suffix of a sequence indexed. Figure 2 shows an example of a suffix tree, which has been converted from the suffix trie in Figure 1. We observe that a large portion of internal nodes have been removed.

3. Indexing method

This section proposes an indexing method which supports efficient DNA sequence matching in large DNA databases. Section 3.1 introduces the trie on which our proposed index structure is based, and Section 3.2 discusses how to construct the binary suffix trie from a set of DNA sequences. Section 3.3 shows the idea of indexing the leaf nodes of the binary suffix trie, and Section 3.4 presents an index construction algorithm.

3.1. Trie

The trie [8, 21], its name originating from the word re*trie*val, was first developed by Briandais [29], and has been intensively discussed by Knuth [30] and in other data structure textbooks. A trie is defined as a |Σ|-*ary* tree in which each edge has a symbol from the alphabet Σ and symbols in each root-to-leaf path form a key. Here, |Σ| 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.

Trie structures have the following properties:

(1) The common prefixes of all key elements are stored only once. This may give substantial data compression.
(2) Trie searching is directed by a search string, and gives search time proportional to the length of a search string rather than the trie size.
(3) Trie shapes are independent of the order in which a data set is presented to the trie construction algorithms, i.e. a trie shape is uniquely determined by its data set.
(4) A trie does not need various reorganization algorithms to keep tries balanced.

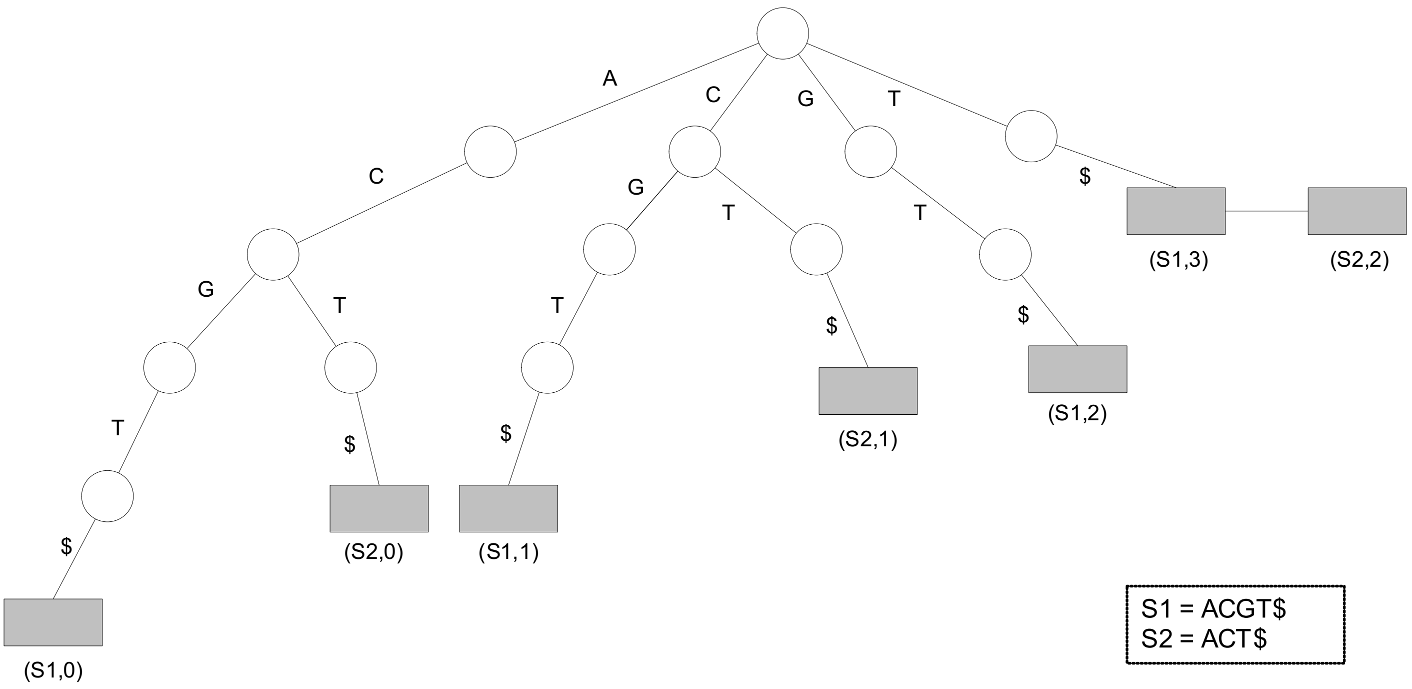


Fig. 1. Suffix *trie* built for two DNA sequences, S1 = 'ACGT' and S2 = 'ACT'.

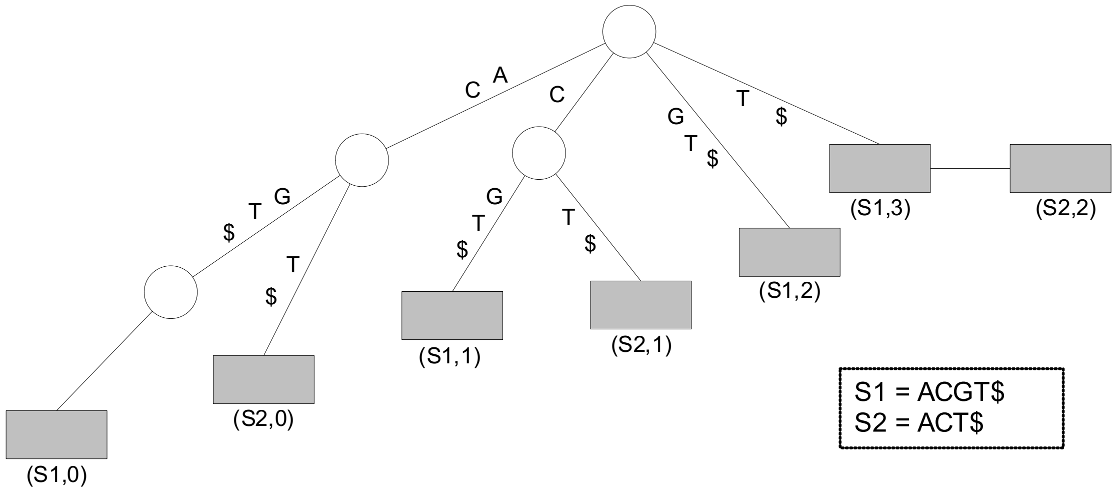


Fig. 2. Suffix *tree* built for two DNA sequences, S1 = 'ACGT' and S2 = 'ACT'.

3.2. Binary suffix trie

The most straightforward implementation of $|\Sigma|$-ary tries is to store $|\Sigma|$ pointers in each node. This method enables a child node to be selected in constant time. However, it is not space-efficient because trie nodes may contain lots of NULL pointers when $|\Sigma|$ 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 $|\Sigma|$ 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. [31] suggested pointerless binary tries which attained competitive search speed with a minimal storage requirement. Pointerless binary tries require the alphabet Σ 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 when enforcing the following rules:

(1) the outgoing edge is labeled with 0 connects to the left child node, and
(2) the outgoing edge is 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. Figure 3 shows the binary trie constructed from the two binary sequences S_1 = '001010' and S_2 = '110100' and Figure 4 shows its pointerless binary bit-string representation.

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 find efficiently 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 [8].

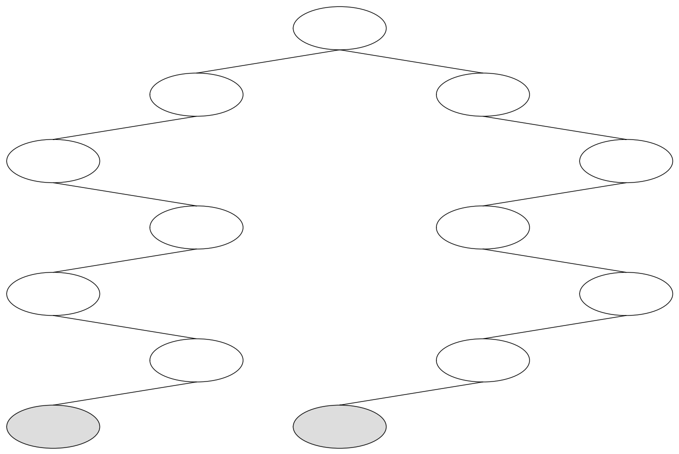


Fig. 3. Binary trie from S1 = '001010' and S2 = '110100'.

```
11
10  01
01  10
10  01
01  10
10  10
00  00
```

Fig. 4. Pointerless binary bit-string representation of Fig. 3.

Table 1
Binary code of each symbol in the alphabet

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

Table 2
Binary representations of the suffixes from S_1 = 'ACGT' and S_2 = 'ACT$'

Suffix		Binary representation
S_1:	ACGT$	001010011101000
	CGT$	010011101000
	GT$	011101000
	T$	101000
S_2:	ACT$	001010101000
	CT$	010101000
	T$	101000

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 $\Sigma = \{A,C,G,T\}$. Since the alphabet size (which is 4) is small, it is highly possible that the suffixes have a considerable number of common prefixes. In this research, we use the minimum number of bits to represent each symbol rather than using a character of eight bits, to obtain a 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 three bits to encode each symbol uniquely. If we construct the suffix trie from DNA sequences encoded in 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. Table 1 shows a binary code for each symbol in our alphabet. Here, '$' is a special character used as an end marker for 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 Table 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 in only one direction and thus facilitate the disk-based trie construction. Figure 5 shows the binary suffix trie constructed from the suffixes of Table 2, and Figure 6 shows its internal representation.

For the trie construction, we use a disk-based algorithm [32]. Therefore, whenever the main memory space of a predetermined size (i.e. page size) is occupied by a sub-trie, it is written onto secondary storage (i.e. a disk). To prevent a sub-trie larger than a page from being written onto a disk page, we pre-calculate 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* to figure out the page connections. Each entry of the page table corresponds to a page and stores information related to that page. Table 3 shows the page table for the page-partitioned trie in Figure 5. Here, *#Page* is the page number, *Node* is the number of nodes contained in the page, and *Addr* is the page address in disk space. *Top* and *Bottom* are the counters which contain the number of edges into and out of the page level. The counting stops right before the page they belong to. Each entry

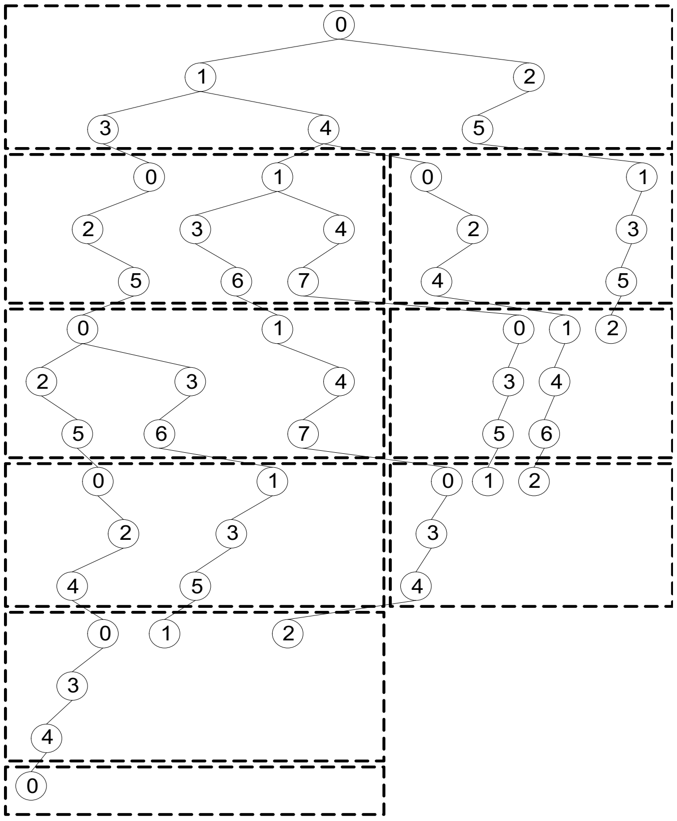


Fig. 5. Binary suffix trie constructed from the suffixes of Table 2.

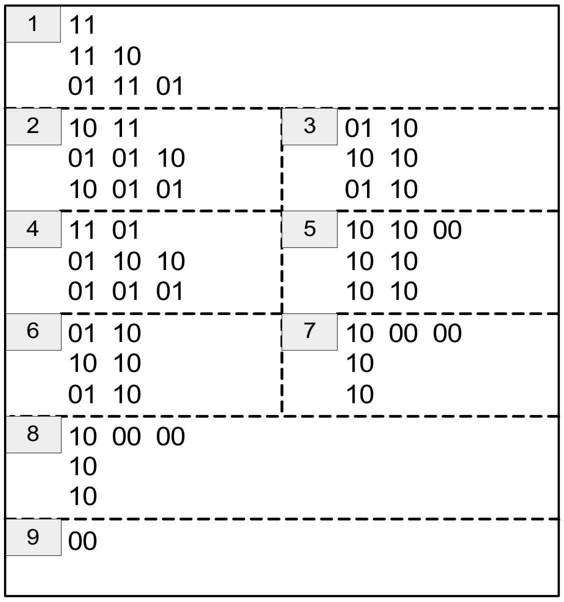


Fig. 6. Internal representations of the binary suffix trie in Fig. 5.

Table 3
Page table for the page-partitioned trie in Fig. 5

#Page	Top	Bottom	Node	Addr
1	0	0	6	84
2	0	0	8	54
3	2	3	6	108
4	0	0	8	24
5	2	3	7	132
6	0	0	6	0
7	2	2	5	159
8	0	0	5	180
9	0	0	1	201

Table 4
Leaf node table for the index in Fig. 5

#Page	R/C	#Seq	Offset
9	0 0	S1	0
8	0 1	S2	0
8	0 2	S1	1
7	0 1	S2	1
7	0 2	S1	2
5	0 2	S1	3
5	0 2	S2	2

of the page table is filled right after the corresponding page has been written on the disk.

3.3. 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. For example, Table 4 keeps the suffix identifiers for the trie shown in Figure 5. Here, *#Page* denotes the page number, *R/C* indicates the row and column numbers at which the leaf node is positioned, and *#Seq* and *Offset* are the sequence identifier and the starting offset respectively.

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-integer*s.

(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.

Figure 7 shows the multi-dimensional index used for direct access to the leaf nodes in the binary suffix trie for two sequences S_1 = 'ACGT' and S_2 = 'ACT'. In this figure, we use a two-byte integer to denote each binary suffix, and also assume k is 1. For example, the suffix 'ACT$' has the binary bit-string representation '001010101000' and it is shorter than two bytes; therefore, four 0s are appended to the end of the binary bit-string thus making it '001010101000*0000*'. The two-byte integer '10880' corresponding to '0010101010000000' is then inserted into the multi-dimensional index with its suffix identifier.

3.4. Index construction algorithm

The proposed algorithm for constructing the binary suffix trie from DNA sequences is summarized as follows:

(1) **Extracting suffixes.** We extract all the suffixes from each of the DNA sequences in the target database.
(2) **Converting into binary suffixes.** Using the minimum number of bits for an alphabet, we convert every suffix into the corresponding binary bit-string representation.
(3) **Constructing the trie.** We sort the suffixes according to their binary bit-string representation, and insert each one of them sequentially into the trie.
    (a) When a new binary suffix is inserted into the trie, existing trie nodes are modified and/or new trie nodes are created. A two-bit number representing a trie node is written on an appropriate page area.
    (b) When an overflow may happen on inserting a new trie node into the current page, we first write the current page onto the disk and start a new page with the trie node we are going to add. Right after the current page is written onto the disk, the information on that page is recorded in the page table.
    (c) After inserting all the binary suffixes into the trie, we denote each leaf node by *k-integer*s and then insert it into the *k*-dimensional index.

4. Query processing method

This section proposes query processing methods to answer the queries of DNA sequence matching using the index structure described in Section 3. Section 4.1

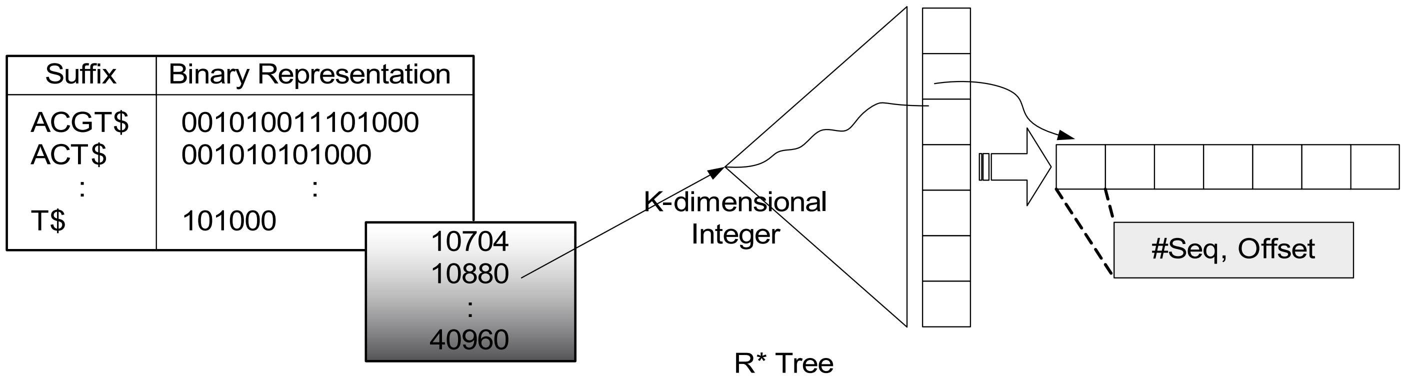


Fig. 7. Multi-dimensional index supporting direct leaf node access within the binary suffix trie in Fig. 5.

gives an algorithm which traverses the binary suffix trie for exact subsequence matching and Section 4.2 describes the method which exploits the multi-dimensional index for leaf-node retrieval. Section 4.3 explains the method used to find the subsequences approximately matching a query sequence.

4.1. 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 that *implicit* information using the data in the page.

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 ith 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 ith 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.

Alogrithm 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* 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 ith 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 *Ni,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.

To see how the algorithm actually works, let us consider the query sequence 'T' and the suffix trie shown in Figure 6. Figure 8 shows how the variables change their values as the algorithm traverses down the index. Here, we use the leaf node table in Table 4 to retrieve the leaf nodes under the node that has been matched with the last query bit. A more sophisticated method for this operation is presented in Section 4.2.

The algorithm starts with the binary bit-string representation '101' for the query sequence 'T'. The algorithm first compares the first query bit with the node 0

[#Page : 1]						
i	Node($N_{i,c}$)	Q_i	C_i	$N_{i,c}$	S_i	$N_{i,f}$
			0	0	1	0
0	11	1	2,0	2	2	2
1	11		2		3	
2	10	0	3,0	5	3	5
3	01		1		3	
4	11		3		4	
5	01	1	4,0	9	4	9
[#Page : 3]			0	1	2	1
0	01					
1	10					
2	10					
3	10					
4	01					
5	10					

Fig. 8. The trace of four variables C_i, $N_{i,c}$, S_i, and $N_{i,f}$ during the traversal of the index in Fig. 6 for a given query sequence 'T'.

of the first page, which is a root node. Then, it compares the second and third query bits with the nodes 2 and 5, respectively. Then, page 3 becomes the current page and all the variables change their values accordingly. Since all the query bits are matched successfully, all the leaf nodes under node 1 of page 3 are retrieved by the depth-first traversal on the index. When the node whose value is '00' is found during this depth-first traversal, we consult the leaf node table to retrieve the corresponding suffix identifiers. In this example, node 2 in page 5 is the leaf node we are looking for. Two suffix identifiers (S_1,3) and (S_2,2) are then obtained from the leaf node table.

4.2. Direct access to leaf nodes

The algorithm *Search-Trie* has a 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 3.3 enables direct retrieval of the leaf nodes under *Ni,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 a 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 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 a length longer than *k*-integers**: let p_t be the prefix of *p* with *k*-integer length. From the multi-dimensional index, we retrieve all the leaf nodes having the value p_t. Then, we perform post-processing to detect and discard false matches.

Let us consider the query sequence 'T' again. Figure 9 shows how the multi-dimensional index is used to directly retrieve the leaf nodes under a given internal node. In this figure, we simply assume that the multi-dimensional index has only one dimension for 2-byte integers. The path *p* with the label '101' matches the binary bit-string representation '101' of the query sequence 'T'. To find the leaf nodes under *p*'s end node, we consult the multi-dimensional index. Since the length of *p* is shorter than a 2-byte integer, we generate p_0 = '101*0000000000000*' by appending thirteen 0s to the end of *p*; and p_1 = '101*1111111111111*' by appending thirteen 1s to the end of *p*. Then, we search the multi-dimensional index for the leaf nodes having a value between p_0 and p_1. As a result, two suffix identifiers (S_1,3) and (S_2,2) are obtained.

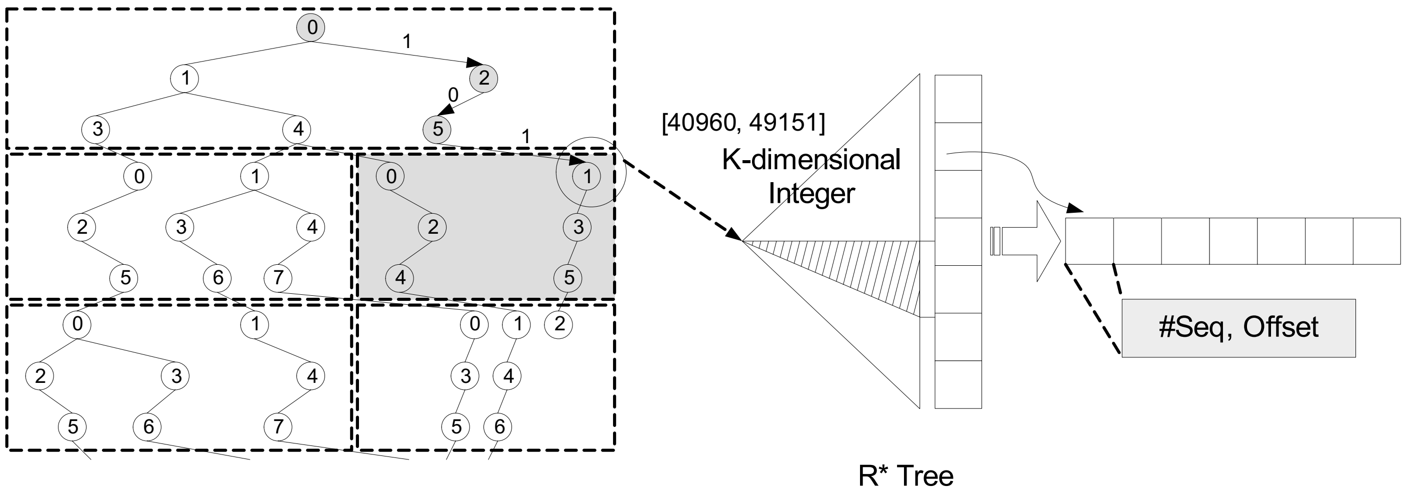


Fig. 9. Multi-dimensional index used to directly retrieve the leaf nodes under a given internal node.

4.3. *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, 12] is a popular similarity measure for approximate subsequence matching.

There have been several approaches [9, 12, 13] 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 can also 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.

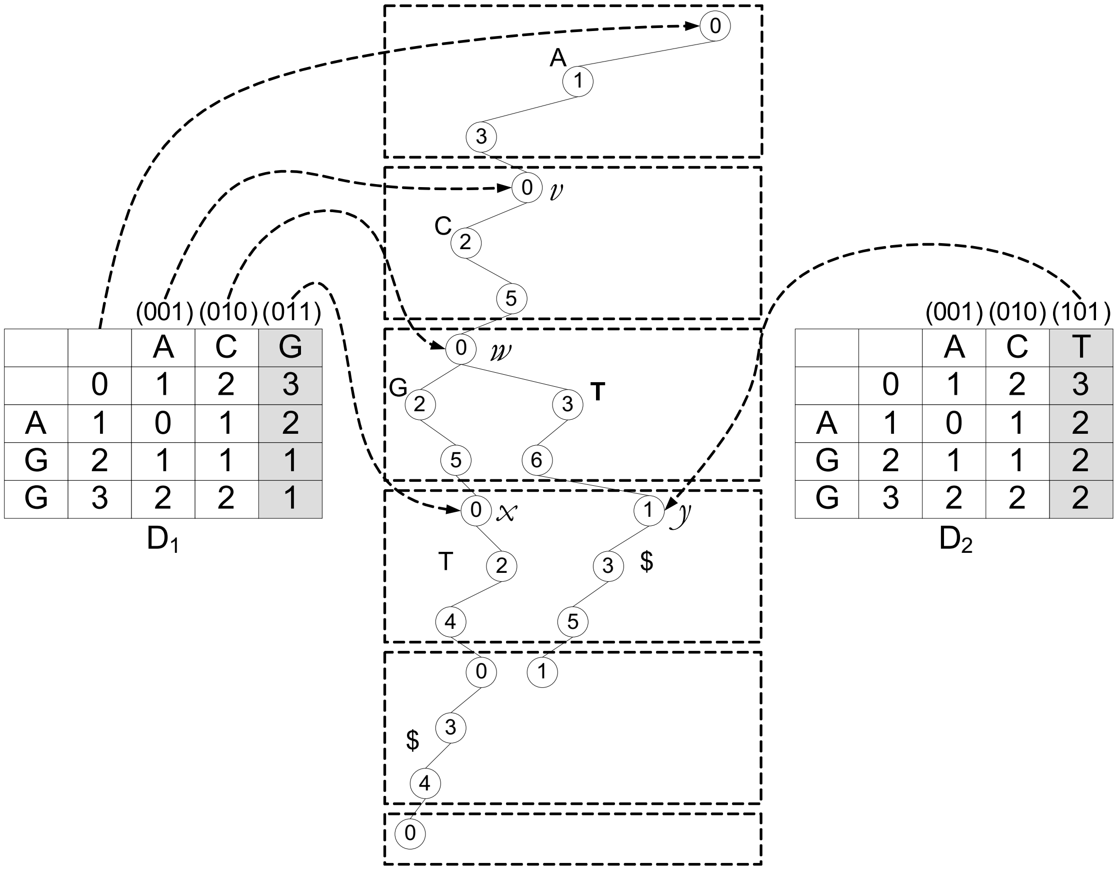
Let us use an example to explain the proposed approximate subsequence matching algorithm. Suppose that we want to find the subsequences whose edit distances to the query sequence ‘AGG’ are not larger than 1. Figure 10 shows how the DP tables are constructed during the traversal of the binary suffix trie shown in Section 3. 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 10 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 10, 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 node x satisfy the query. The DP table D_2 is obtained when node y is visited. Since all the cells in the last column have values larger than 1, the traversal stops at node y and comes back to its parent. Note that the first two columns of tables D_1 and D_2 are identical. These two columns are shared by the two tables to save space and time.

5. Performance evaluation

In this section, we show the effectiveness of our approach via performance evaluation with extensive experiments. Section 5.1 describes the environment for experiments, and Section 5.2 presents and analyzes the results.

5.1. *Environment*

In the experiments, we have used five sets of DNA sequences downloaded from GenBank [33]. They are a human chromosome 18 that consists of three different



		A	C	G
	0	1	2	3
A	1	0	1	2
G	2	1	1	1
G	3	2	2	1

		A	C	T
	0	1	2	3
A	1	0	1	2
G	2	1	1	2
G	3	2	2	2

Fig. 10. DP tables constructed from the binary suffix trie of Fig. 5.

sequences of 1.07 Mbp, 2.16 Mbp, and 4.22 Mbp, a human chromosome 21 of 43.3 Mbp, and a human chromosome 19 of 72.3 Mbp. As query sequences, we have randomly extracted some subsequences of arbitrary lengths from such DNA sequences. Other than A, C, G, and T, the DNA sequences contain some infrequent wild-card characters such as N, S, and Y. In addition, we have to use $ to represent the end of a sequence. Thus, eight different characters appear within the DNA sequences used in our experiments.

The hardware platform is the Pentium IV 2GHz PC equipped with 1 Gbytes main-memory and 40 Gbytes HDD. The software platform is Windows 2000 Server. The performance factors for comparing approaches are the size of indexes and the elapsed time for DNA sequence matching.

5.2. Results and analyses

In experiment 1, we have compared the three approaches *Trie-Rtree*, *Trie-Naive*, and *Suffix* in respect of index size. Trie-Rtree represents our approach employing 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 [34], which is most widely used in the literature. Trie-Naive also represents our approach employing the trie using pointerless binary bit-string representation but 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 [32] for suffix tree construction, and also have allocated a 32 byte memory chunk for each node in the suffix tree.

Table 5 shows the sizes of the index components in the three approaches with changing data sizes. We have set the page size for each index to 4K bytes. The index in Suffix consists of internal nodes and leaf nodes for the suffix tree. The index in Trie-Naive consists of a binary suffix trie, a page table, and a leaf node table. The page table maintains the entries, each of which corresponds to a page in a trie, and thus its size is not

Table 5
Index sizes of three approaches

Data size	Trie-Rtree			Trie-Naive			Suffix
	trie_idx	page_tbl	R*_tree	trie_idx	page_tbl	leaf_tbl	
1.07 Mbp	15.9M	80.3K	30.5M	15.9M	80.3K	12.7M	54.5M
2.16 Mbp	31M	155K	59.6M	31M	155K	25.6M	108M
4.22 Mbp	59.9M	300K	116.5M	59.9M	300K	49.7M	209M
43.3 Mbp	577M	2.82M	1,132M	577M	2.82M	517M	2.07G
72.3 Mbp	878M	4.29M	1,637M	878M	4.29M	858M	3.31G

that large. On the other hand, the leaf node table is fairly large since it has the entries, each of which corresponds to a suffix in sequences. The index in Trie-Rtree is almost the same as that in Trie-Naive, but it additionally maintains an R*-tree for fast access to leaf nodes in the trie.

Figure 11 shows the change in index size in the three approaches with different data sizes. We observe that the index size increases linearly in proportion to the data size in all the approaches. In comparison with Suffix, our Trie-Naive and Trie-Rtree save around 48% and 24% storage space, respectively.

In experiment 2, we have compared the three approaches along with *Seqscan* in terms of the elapsed time for exact subsequence matching. Seqscan is based on sequential scan and is regarded as the simplest baseline method for DNA sequence matching. For this experiment, we have used human chromosome 21 of 43.3Mbp 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 a query sequence start.

In the case of Suffix, the post-processing time grows as a query sequence decreases in length. When the

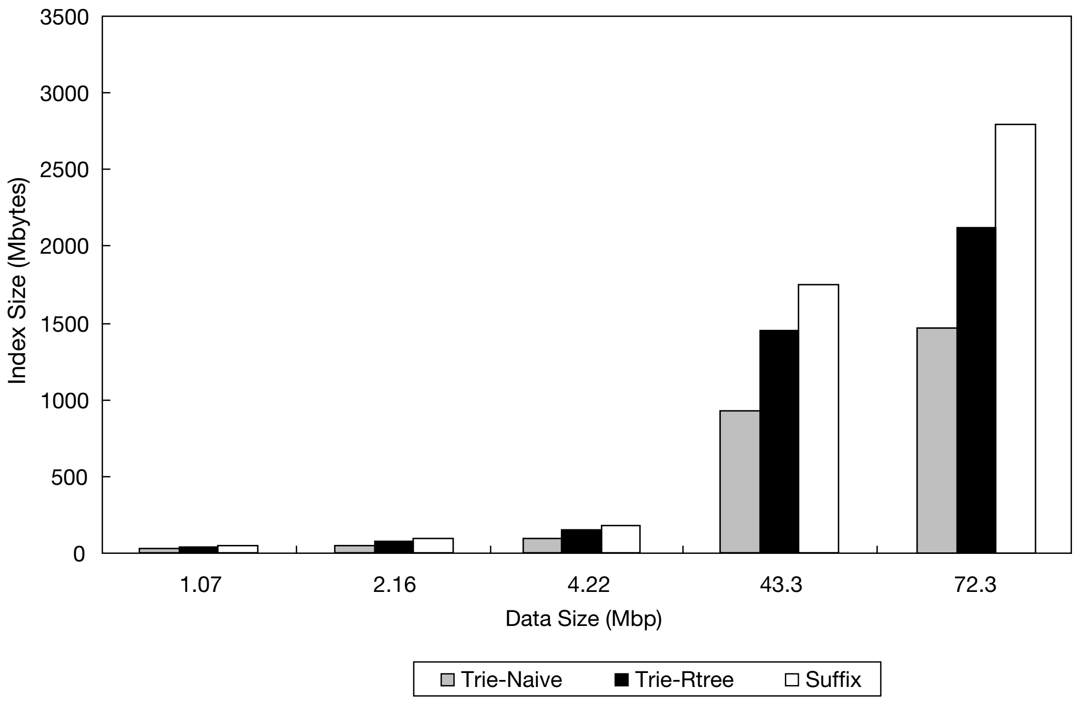


Fig. 11. Index sizes with different data sizes.

lengths are larger than 10, however, the post-processing time becomes nearly 0. After a path in a suffix tree or a binary suffix trie is found to be matched with the query sequence at an internal node *N*, we have to find all the leaf nodes under the node *N*. We call the time spent in this processing the *post-processing time*. In cases of Trie-Naive and Trie-Rtree, the post-processing time occupies a large portion of the total elapsed time when a query sequence is short. In particular, Trie-Naive spends most of its time in post-processing.

Table 6 shows the elapsed time in the four approaches with changing query sequence lengths. The values within parentheses represent the post-processing times. In Suffix, as the length of a query sequence increases up to 10, the post-processing time decreases. In cases of Trie-Naive and Trie-Rtree, the post-processing time occupies a majority of the total elapsed time.

Figure 12 shows the total elapsed times for DNA sequence matching when using all four approaches. The *X* axis denotes a varying query sequence length, and the *Y* axis the elapsed time in the unit of a millisecond. We note that the *Y* axis is in log-scale. 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 13–29 times speedup compared with Suffix, and 54–145 times speedup compared with Seqscan.

In experiment 3, we have compared the 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 DNA subsequence matching; and the other is to find similar subsequences using the best-first(A*) search algorithm. The data sequence used in the experiment is human chromosome 21 of 43.3 Mbp.

Table 7 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 method 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 performance due to direct access to 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 13 depicts the result of comparing the elapsed times of Suffix*, Trie-Rtree* and **SW**. (SW represents elapsed time of approximate subsequence matching by the Smith-Waterman algorithm.) 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 [15]. Also, the result shows that Trie-Rtree* performs better than Suffix*. This is because the way nodes are stored in the suffix trie harmonizes with the level-first traversal fashion of the best-first (A*) search algorithm. That is, as mentioned in Section 3.2, 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 an

Table 6
Query processing times of four approaches

Query length	Query processing time (ms)			
	Trie-Rtree	Trie-Naive	Suffix	Seqscan
6	68.1 (62.1)	230,423.5 (230,417.5)	919.2 (256.2)	3702.7
8	37.3 (24.2)	13,612.4 (13,599.3)	741.3 (14.5)	3701.9
10	33 (19.2)	1,152.4 (1,138.6)	717.8 (1.7)	3681.1
15	25.1 (9.4)	104.9 (89.2)	726.8 (0.1)	3643.6
30	34.7 (8.1)	79.4 (52.8)	729.8 (0)	3761.6
60	43.7 (8.2)	67.5 (32)	823.4 (0)	3677.5

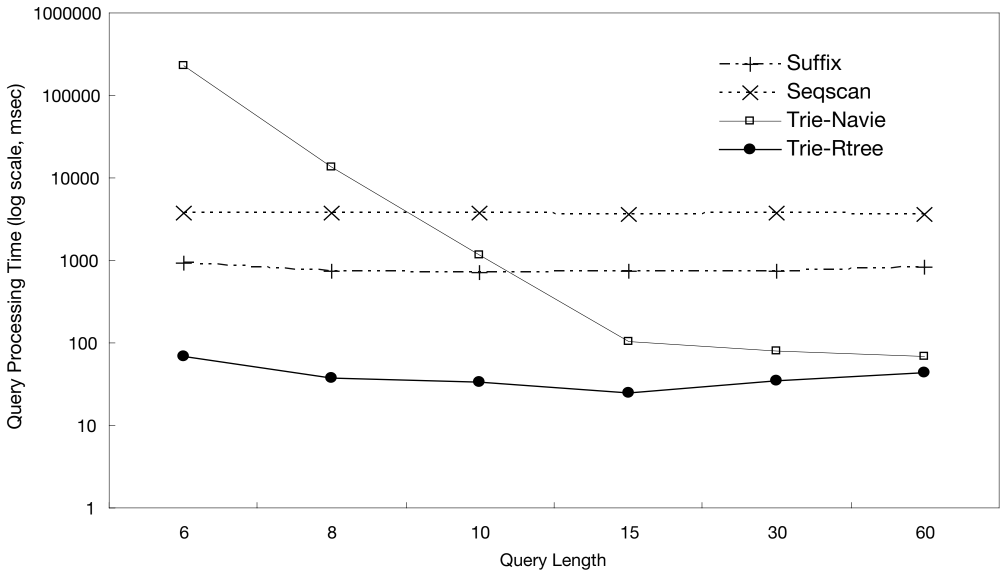


Fig. 12. Elapsed time of exact subsequence matching with different query sequence lengths.

Table 7
Elapsed times spent in finding all the subsequences whose edit distances to a query sequence are not larger than 1

Query length	Total hits	Query processing time (ms)	
		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)

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 four to nine times better and about 592 to 2,505 times better, respectively.

6. Conclusions

DNA sequence matching is a widely-used operation in molecular biology. Since DNA databases are huge in general, fast indexing is crucial for efficient processing of DNA sequence matching. In this paper, we have first pointed out the problems occurring in the suffix tree for DNA sequence matching: (1) high storage overhead, (2) low search performance, and (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 access to the target leaf nodes after traversing the trie. With the proposed index, we can successfully alleviate the three problems of the suffix tree. We have also proposed 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 cases of exact matching, Trie-Rtree, our enhanced approach, runs 13 to 29 times faster than Suffix. In cases of approximate matching, it achieves four to nine times speedup over Suffix.

As a further study, we are investigating sophisticated techniques to improve the performance of approximate subsequence matching in cases with long query sequences as well as large tolerance values. For this, we are considering employing a novel method for tree traversal that fully makes use of the arrangement of index nodes in a disk. Also, we plan to devise

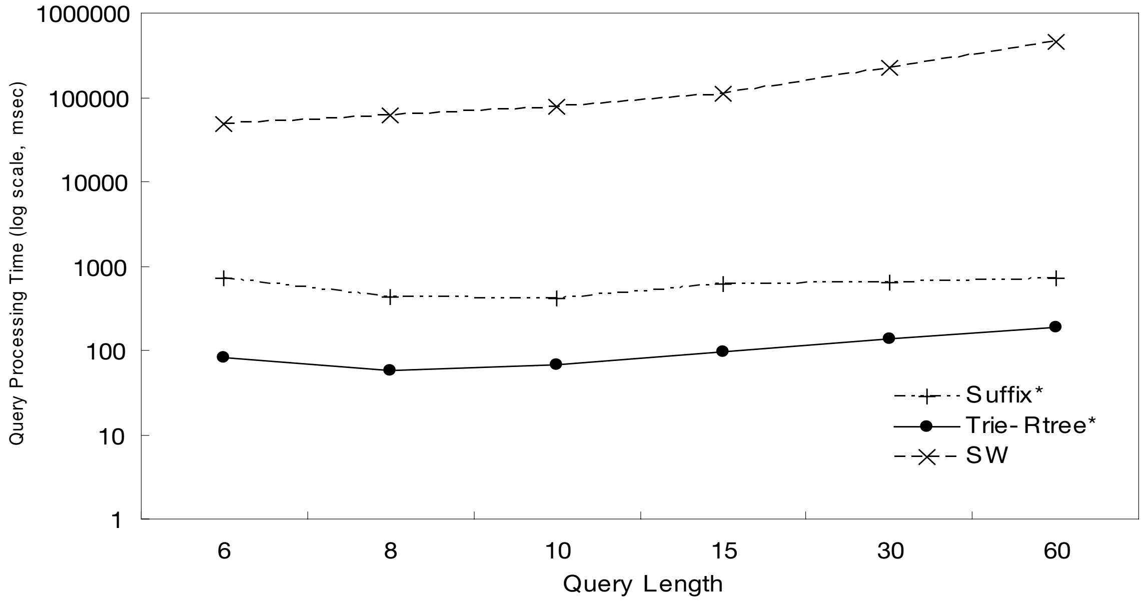


Fig. 13. Elapsed times spent in finding the subsequence most similar to a query sequence.

techniques that allow incremental updates on the binary trie disk structure.

Acknowledgements

This work was supported by a Korea Research Foundation Grant funded by the Government of Korea (MOEHRD, Basic Research Promotion Fund) KRF-2005–206-D00015; and by KOSEF Basic Research Program Grant R04–2003–000–10048–0; and by the MIC (Ministry of Information and Communication) of Korea under the IT Research Center support program (IITA–2005–C1090–0009) supervised by the IITA (Institute of Information Technology Assessment).

References

[1] C. Gibas and P. Jambeck, *Developing Bioinformatics Computer Skills* (O'Reilly and Associates, Tokyo, 2001).
[2] D.W. Mount, *Bioinformatics: Sequence and Genome Analysis* (Cold Spring Harbor Laboratory Press, Cold Spring Harbor, 2001).
[3] R.S.C. Goble, P. Baker and A. Brass, A Classification of tasks in bioinformatics, *Bioinformatics* 17(2) (2001) 180–88.
[4] D.A. Benson, M.S. Boguski, D.J. Lipman, J. Ostell and B.F. Quellette, Genbank, *Nucleic Acids Research* 26(1) (1998) 1–7.
[5] S. Altschul, W. Gish, W. Miller, E. Myers and D. Lipman, Basic local alignment search tool, *Journal of Molecular Biology* 215 (1990) 403–10.
[6] 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.
[7] T. Smith and M. Waterman, Identification of common molecular subsequences, *Journal of Molecular Biology* 147 (1981) 195–97.
[8] G.A. Stephen, *String Searching Algorithms* (World Scientific Publishing, River Edge, 1994).
[9] E. Ukkonen, Approximate string matching over suffix trees. In: A. Apostolico et al. (eds), *Proceedings of the 4th Annual Symposium on Combinatorial Pattern Matching (CPM93) 1993* (Springer, 1993) 228–42.
[10] A.L. Delcher, S. Kasif, R.D. Fleischmann, J. Peterson, O. White and S.L. Salzberg, Alignment of whole genomes, *Nucleic Acids Research* 27 (1999) 2369–76.
[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* 29(22) (2001) 4633–42.
[12] 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.
[13] G. Navarro and R. Baeza-Yates, A hybrid indexing method for approximate string matching, *Journal of Discrete Algorithms* 1(1) (2000) 205–39.
[14] G. Navarro and R. Baeza-Yates, A new indexing method for approximate string matching. In: M. Crochemore and

M. Paterson (eds), *Proceedings of the 10th Annual Symposium on Combinatorial Pattern Matching (CPM99) 1999* (Springer, 1999)163–85.

[15] C. Meek, J.M. Patel and S. Kasetty, OASIS: an online and accurate technique for local-alignment searches on biological sequences. In: J.C. Freytag et al. (eds), *Proceedings of the 29th International Conference on Very Large Databases(VLDB03) 2003* (Morgan Kaufmann, 2003) 920–21.

[16] K. Kelly and P. Labute, *The A* Search and Applications to Sequence Alignment* (1996). Available at: www.chemcomp.com/Journal_of_CCG/Articles/astar.htm (accessed 28 July 2005).

[17] S. Kurtz and C. Schleiermacher, REPuter: fast computation of maximal repeats in complete genomes, *Bioinformatics* 15(5) (1999) 426–7.

[18] R. Giegerich, S. Kurtz and J. Stoye, Efficient implementation of lazy suffix trees, *Software-Practice and Experience* 33 (2003) 1035– 49.

[19] K. Sadakane and T. Shibuya, Indexing huge genome sequences for solving various problems. In: H. Matsuda et al. (eds), *Proceedings of the 12th Genome Informatics (GIW01) 2001* (Universal Academy Press, 2001) 175–83.

[20] H. Wang et al., BLAST++: a tool for BLASTing queries in batches. In: Y.P.P. Chen (ed.), *Proceedings of the 1st Asia-Pacific Bioinformatics Conference (APBC03) 2003* (Australian Computer Society, 2003) 71–9.

[21] E. Horowitz, S. Sahni and S. Anderson-Freed, *Fundamentals of Data Structures in C* (Computer Science Press, New York, 1993).

[22] J. Buhler, Efficient large-scale sequence comparison by local-sensitive hashing, *Bioinformatics* 17 (2001) 419–28.

[23] B. Ma, J. Tromp, and M. Li, Patternhunter: faster and more sensitive homology search, *Bioinformatics* 18 (2002) 440–45.

[24] A. Califano and I. Rigoutsos, FLASH: a fast look-up algorithm for string homology. In: L. Hunter et al. (eds), *Proceedings of the 1st International Conference on Intelligent Systems for Molecular Biology, 1993* (AAAI Press, 1993) 56–64.

[25] C. Fondrat and P. Dessen, A rapid access motif database (RAMdb) with a searching algorithms for the retrieval patterns in nucleic acids or protein databanks, *Computer Applications in the Biosciences* 11(3) (1995) 273–9.

[26] H.E. Williams and J. Zobel, Indexing and retrieval for genomic databases, *IEEE TKDE* 14(1) (2002) 63–78.

[27] 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) 2001* (Morgan Kaufmann, 2001) 351–60.

[28] U. Manber and G. Myers, Suffix arrays: a new method for on-line string searches, *SIAM Journal on Computing* 22(5) (1993) 935–48.

[29] R. de la Briandais, File searching using variable length keys. In: *Proceedings of Western Joint Computer Conference, 1959* (AFIPS Press, 1959) 295–8.

[30] D.E. Knuth, *The Art of Computer Programming 3: Sorting and searching* (Addison-Wesley, Boston, 1973).

[31] H. Shang and T.H. Merrett, Tries for approximate string matching, *IEEE Transactions on Knowledge and Data Engineering* 8(4) (1996) 540–47.

[32] P. Bieganski, J. Riedl and J.V. Carlis, Generalized suffix trees for biological sequence data: applications and implementation. In: *Proceedings of the 27th International Conference on System Sciences 1994* (IEEE, 1994) Vol. 5, 35–44.

[33] *Genbank*. Available at: www.ncbi.nlm.nih.gov (accessed 28 July 2005).

[34] N. Beckmann, H. Kriegel, R. Schneider and B. Seeger, The R*-tree: an efficient and robust access method for points and rectangles. In: H. Garcia-Molina and H.V. Jagadish (eds), *Proceedings of ACM SIGMOD International Conference on Management of Data 1990* (ACM Press, 1990) 322–31.