Fundamenta Informaticae 78 (2007) 525–542
IOS Press

Towards Efficient Searching on the Secondary Structure of Protein Sequences

Minkoo Seo, Sanghyun Park*
Department of Computer Science, Yonsei University
134 Sinchon-dong, Seodaemun-gu, Seoul 120-749, Korea
mkseo@cs.yonsei.ac.kr; sanghyun@cs.yonsei.ac.kr

Jung-Im Won
College of Information and Communications Hanyang University
17 HaengDangDong, SeongDongGu, Seoul, Korea
jiwon@hanyang.ac.kr

Abstract. Approximate searching on the primary structure (i.e., amino acid arrangement) of protein sequences is an essential part in predicting the functions and evolutionary histories of proteins. However, because proteins distant in an evolutionary history do not conserve amino acid residue arrangements, approximate searching on proteins' secondary structure is quite important in finding out distant homology. In this paper, we propose an indexing scheme for efficient approximate searching on the secondary structure of protein sequences which can be easily implemented in RDBMS. Exploiting the concept of *clustering* and *lookahead*, the proposed indexing scheme processes three types of secondary structure queries (i.e., exact match, range match, and wildcard match) very quickly. To evaluate the performance of the proposed method, we conducted extensive experiments using a set of actual protein sequences. According to the experimental results, the proposed method was proved to be faster than the existing indexing methods up to 6.3 times in exact match, 3.3 times in range match, and 1.5 times in wildcard match, respectively.

Keywords: Indexing method, Secondary structure of proteins, and Approximate searching

*Address for correspondence: Department of Computer Science, Yonsei University, 134 Sinchon-dong, Seodaemun-gu, Seoul 120-749, Korea

1. Introduction

It is well known to biologists that the amino acid arrangements of proteins determine their structures and functions. Therefore, it is possible to predict the functions, roles, structures, and categories of newly discovered proteins by searching for the proteins whose amino acid arrangements are similar to those of newly discovered proteins [1, 19].

However, the amino acid arrangement of one protein is rarely preserved in another protein if the two proteins are distant in an evolutionary history [5, 15, 16]. Therefore, approximate searching on protein structures, rather than on amino acid arrangements, is more important in finding out distant homology. Among structure searching algorithms, comparing structural arrangements based on the secondary structure elements is gaining more popularity in conjunction with database approaches [5, 14].

The secondary structures are expressed using the three characters: E (beta sheets), H (alpha helices), and L (turns or loops). These characters tend to occur contiguously rather than interspersedly [4, 11]. For example, '$HHLLLLLEEE$' is more likely to occur than '$HLLELLEELH$'.

Exploiting this property, Hammel et al. [11] proposed a segment-based indexing method. The method combines consecutive characters of a same type into a single segment and then builds a B^+-tree on two attributes of segments: (1) Type which denotes the type of consecutive characters, and (2) Len which denotes the number of consecutive characters. For example, '$HHLLLLLEEE$' is segmented into 'HH / $LLLLL$ / EEE' and expressed as $(H, 2)(L, 5)(E, 3)$. After then, given a query specified as (Type, Len)$^+$, n most infrequently occurring segments are chosen, and then matched against segments populated from sequences in the database. As the final step, candidate sequences each of which matches the n segments are compared to the query itself.

Although the segmentation enables an efficient searching on the secondary structures, it has innate limitations. First, the pair of (Type, Len) does not have uniform distribution. According to our preliminary experimentation with 80,000 proteins, 87% of E segments have a length between 3 and 6, 62% of H segments have a length between 5 and 14, and 41% of L segments are of length between 3 and 6. Therefore, if every segment in a query is close to one of these *hot spots*, index or full table scan for a segment of the query will produce a large result set. Furthermore, candidate set size populated from the most selective n segments might be still large even after the result sets are merged. Thus, the overall performance will be bad even if another indexing scheme like horizontal partitioning is adopted. Second, the number of distinct (Type, Len) pairs is not large enough to provide good selectivity. Our investigation on 80,000 proteins indicates that the total number of distinct (Type, Len) pairs is about 300 but the total number of segments to be indexed is more than 3 millions. Therefore, the average number of segments with the same (Type, Len) pair is more than 10,000.

In this paper, we propose CSI (Clustered Segment Indexing), an efficient indexing scheme for approximate searching on the secondary structure of protein sequences. The proposed indexing scheme exploits the concept of *clustering* and *lookahead* to overcome the aforementioned limitations. A predetermined number of neighboring segments are grouped into a cluster which is then represented by three attributes (we revisit the issue of determining the number of neighbors in Section 5.2): (1) CluStr which denotes the type string of the cluster obtained by concatenating the Type attributes of the underlying segments, (2) CluLen which denotes the length of the cluster obtained by summing up the Len attributes of the underlying segments, and (3) CluLA which denotes the lookahead of the cluster obtained by concatenating the Type attributes of the segments *following* the cluster. If more than one segments

are gathered together then the triple (CluStr, CluLen, CluLA) for a cluster is more discriminative than the pair (Type, Len) for a segment.

In the proposed method, if a user enters a query in the console as a series segments, then they are transformed into clusters by the system. Among those clusters, n clusters are chosen and matched against the cluster tables which contains protein sequences represented as (CluStr, CluLen, CluLA). After getting canidates, postprocessing stage will match each of candidate against the original query.

Organization of this paper is as follows. Section 2 describes related work in the field of homologous sequence searching and structure searching. Section 3 and Section 4 present the proposed indexing and query processing algorithms, respectively. Section 5 and 6 shows the effectiveness of the proposed approach via performance evaluation and computational analysis, respectively. Finally, Section 7 summarizes and concludes our work.

2. Related Work

BLAST [2, 3] is the most widely used tool for approximate searching on DNA and protein sequences. BLAST is based on the sequential scan method basically, but it makes use of heuristic algorithms to reduce the number of sequences to be aligned against a query. However, BLAST still has two main drawbacks [19]: (1) entire data set should be loaded into a main memory for fast searching, and (2) since it is based on sequential access, its execution time is directly proportional to the number of sequences in the database. Due to these drawbacks, index-based approaches for approximate searching are demanding.

Suffix trees [17] have been recognized as the best index structure for string or sequence searching, but they have been notorious for large space requirement. Recently, algorithms for building a suffix tree from a data set larger than a main memory were proposed [13]. However, the internal structure of suffix trees is not suitable for pagination and therefore it is not easy to incorporate suffix trees into database systems [17, 18].

RAMdb [7] is an indexing system for the primary structures of protein sequences and was proved, by experiments, to be faster than heuristic approaches up to 800 times. However its search performance deteriorates when the length of a query is not close to that of the interval used for indexing. In addition, RAMdb is an indexing system mainly for the primary structures of protein sequences and therefore it is not easy to apply the proposed idea directly to the secondary structures of protein sequences.

Hammel et al. [11] proposed the segment-based indexing method. The method combines the consecutive characters of the same type into a single segment, and then builds a B^+-tree index on the number and type of consecutive characters. As mentioned in the previous section, however, this segment-based approach does not support good selectivity, thus resulting in an innate limitation of search performance.

VAST [10] and DALI [12] support three dimensional structure-based similarity search algorithms. VAST is motivated by the fact that the number of secondary structure elements (SSEs) is much smaller than the number of C_α and C_β atoms [15]. Hence, VAST performs substructure alignments in three steps: (1) rapid identification of SSE pair alignments, (2) clustering identified SSEs into groups, and (3) scoring the best substructure alignment.

DALI, on the other hand, compares C_α atoms using distance matrices. For each protein, a distance matrix which resembles a dot matrix is populated. Each dot in the matrix represents the distance between C_α atoms along the polypeptide chain and between C_α atoms within the protein structure [15]. Therefore, by comparing matrices, DALI can find the proteins whose three dimensional structures are similar to that of a given query.

As stated above, VAST and DALI tackle the problem of structure comparisons, making it primary concerns to decrease the time in comparing three dimensional coordinates of secondary structure elements. CSI is different from these approaches in that it searches for similar proteins by comparing types and lengths, rather than three dimensional coordinates, of their secondary structure elements.

3. Segment Table And Clustered Segment Table

CSI (Clustered Segment Indexing), the proposed indexing method, utilizes the idea of segment table introduced in [11]. Therefore, in this section, we first explain the structure of segment table and then present the structure of clustered segment table, the main data structure used in the proposed indexing method.

3.1. Segment table

A segment in a protein is defined as consecutive characters of the same secondary structure type. A segment can be expressed by two attributes: Type to denote the type of consecutive characters and Len to denote the number of consecutive characters. For example, the sequence S_1 = '$EEEHHLLEEE$' is segmented into '$EEE/HH/LL/EEE$' and then expressed as $(E,3)(H,2)(L,2)(E,3)$. In addition to the two attributes, the position at which a segment begins is needed to identify the segment. In case of S_1, each of the four positions, 0, 3, 5, and 7, is associated to the corresponding segment. The information on each segment is stored in the segment table. Table 1 shows the segment table for S_1 = '$EEEHHLLEEE$'.

Table 1. Segment table for S_1 = '$EEEHHLLEEE$'.

SegID	ProteinID	Loc	Type	Len
1	S_1	0	E	3
2	S_1	3	H	2
3	S_1	5	L	2
4	S_1	7	E	3

After then, a B^+ tree is built on Type and Len column to support index scan for a query represented by (Type, Len)$^+$.

3.2. Clustered segment table

A segment itself has a limitation in selectivity. Therefore, we group a pre-determined number of neighboring segments into a cluster and express it using more discriminative attributes. In this approach, it is assumed that the sequences in the database is almost never updated.

Before building clustered segment tables, it is needed to set $MaxK$ which is a system parameter used to control the total number of clustered segment table being constructed. (How to set the value of $MaxK$ is explained in Section 5.2.1.) The procedure to construct clustered segment tables is as follows:

1. Convert each protein sequence S into a series of segments. Let N_S be the number of segments obtained from S.

2. For each k from 0 to $min(\lfloor log_2(N_s) \rfloor, MaxK)$, do the following:

   (a) Using the sliding window of size 2^k, generate a set of clusters, each of which is composed of 2^k neighboring segments.

   (b) Store each cluster into the clustered segment table named CST_k.

Let $(T_1, L_1)(T_2, L_2)...(T_{2^k}, L_{2^k})$ be 2^k neighboring segments where T_i is Type and L_i is Len of the i^{th} segment. Then the cluster is represented concisely as $(\mathsf{CluStr} = T_1 \cdot T_2 \cdot ... \cdot T_{2^k}, \mathsf{CluLen} = L_1 + L_2 + ... + L_{2^k})$ where CluStr denotes the type string of the cluster obtained by concatenating the Type attributes of the underlying segments, and CluLen denotes the length of the cluster computed by summing up the Len attributes of the underlying segments. For example, when $(H, 2)(L, 2)$ is combined into a cluster, it is represented as $(HL, 4)$. Figure 1 depicts the conceptual view of the clustering procedure for the sequence S_1 whose segment table is shown in Table 1.

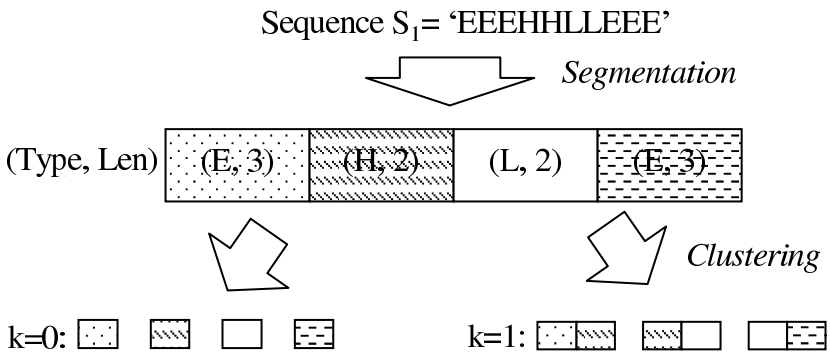

Figure 1. Generating clustered segment tables for S_1 = '$EEEHHLLEEE$'.

There may be a series of segments following a cluster. The Type attributes of such segments can be concatenated, producing the lookahead, CluLA, of the cluster. The maximum length of CluLA is controlled by a system parameter $MaxCluLA$ for space efficiency. The overall schema for each clustered segment table is shown in Table 2.

For example, as shown in Table 3, two clustered segment tables, CST_0 and CST_1, are constructed from S_1 = '$EEEHHLLEEE$' when $MaxK = 1$ and $MaxCluLA = 2$.

After populating all the tuples of CST_k, the tuples are sorted according to CluStr, CluLen, and CluLA for the sake of locality. As the final step, we build one B^+-tree on two columns, CluStr and CluLen, for each CST_k. It is also worth mentioning that the duplication of information in CST_k will bring about more storage consumption than the segment table. Hence, we store each character using 2 bits like: $L = 00_2$, $H = 10_2$, and $E = 11_2$.

Table 2. Schema of each clustered segment table.

Field Name	Description
ID	The identifier of the protein from which a cluster is made.
Loc	The beginning position of the cluster.
CluStr	The type string of the cluster obtained by concatenating the Type attributes of the underlying segments.
CluLen	The length of the cluster obtained by summing up the Len attributes of the underlying segments.
CluLA	The type string obtained by concatenating the Type attributes of the segments following the cluster.

Table 3. Clustered segment tables, CST_0 and CST_1, from S_1 = '$EEEHHLLEEE$'.

CST_0

ID	Loc	CluStr	CluLen	CluLA
S_1	0	E	3	HL
S_1	3	H	2	LE
S_1	5	L	2	E
S_1	7	E	3	

CST_1

ID	Loc	CluStr	CluLen	CluLA
S_1	0	EH	5	LE
S_1	3	HL	4	E
S_1	5	LE	5	

4. Query Processing

4.1. Overall query processing algorithm

As stated in the introduction, a user query is specified as a series of segments like (Type, Len)$^+$. For compactness, in what follows, let $< T_1(L_1)\ T_2(L_2)\ \ldots\ T_{N_Q}(L_{N_Q}) >$ be a user query where T_i represents a Type or a wildcard, ?, that matches any Type and L_i represents the length of the type T_i. Then, the typical example query to search for $EEEHHLLEEE$ can be specified as $< E(3)\ H(2)\ L(2)\ E(3) >$. In the proposed method, it is also possible to specify the lower bound and the upper bound of a segment. Such an example query is $< E(3\ 5)H(3\ 6)L(3\ 7) >$.

In this section, the query processing algorithm to search for secondary structure of protein sequences that matches a user query is presented. Suppose that $MaxK + 1$ tables, CST_0, CST_1, ..., and CST_{MaxK}, were created along with their associated B$^+$-tree indices. The overall query processing algorithm which uses these tables and associated indices to process a query is as follows:

1. Convert a query Q into a series of segments. Let N_Q denote the number of segments obtained from Q.
2. Determine the target table CST_k by computing the expression $k = min(\lfloor log_2(N_Q) \rfloor, MaxK)$.
3. Decompose the segmented query into n (=$\lceil N_Q/2^k \rceil$) non-overlapping subqueries, each of which

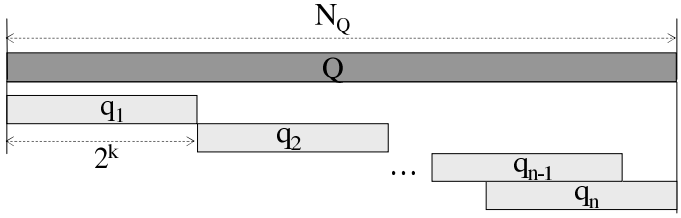

Figure 2. Subqueries generated from a query Q.

has 2^k segments in it. The last two subqueries may overlap each other when N_Q is not a multiple of 2^k (see Figure 2).

4. For each subquery q_i (i=1,2,...,n), do the following:
   (a) Compute its CluStr, CluLen, and CluLA values. Let qCluStr, qCluLen, and qCluLA denote these three values, respectively.
   (b) Search the table CST_k for the tuples whose CluStr, CluLen, and CluLA match with qCluStr, qCluLen, and qCluLA, respectively. The B$^+$-tree index for CST_k is used at this step.
5. Perform the sort-merge on n sets of intermediate results using their ID and Loc as joining attributes.
6. Perform the post-processing to detect and discard false matches.

4.2. Exact match query

Exact match queries are expressed as $Q =< T_1(L_1)\ T_2(L_2)\ \ldots\ T_{N_Q}(L_{N_Q}) >$ where $T_i\ (\in \{E, H, L\})$ and L_i represent the type and length of the i^{th} segment of Q, respectively. Suppose that we already chose the target table CST_k and decomposed the query into n subqueries, each of which consists of 2^k segments. The algorithm for processing exact match queries is shown in Algorithm 1.

The result of subquery q_i is stored in N_i. If the number of tuples in N_i is less than a predefined threshold ϵ, then we believe that irrelevant answers have been filtered out sufficiently. Therefore, if this happens, we directly go to the merging step (Line 5) without considering the remaining subqueries, $q_{i+1}, q_{i+2}, \ldots, q_n$.

4.3. Range match query

Range match queries are expressed as $Q =< T_1(Lb_1\ Ub_1)\ T_2(Lb_2\ Ub_2)\ \ldots\ T_{N_Q}(Lb_{N_Q}\ Ub_{N_Q}) >$ where Lb_i and Ub_i represent the minimum and maximum length of the i^{th} segment of Q, respectively. Unlike exact match queries where every search condition is expressed as an *equality* predicate, this type of query contains a search condition expressed as a *range* predicate. More specifically, the search condition of CluLen for each subquery q_i has the form of 'CluLen between qLb and qUb' where qLb is the sum of the minimum lengths and qUb is the sum of the maximum lengths of the underlying

Algorithm 1: ProcessExactMatchQuery

Input : Query Q, Clustered segment table CST_k, Threshold ϵ
Output : Set of answers

```
1 for (each subquery q_i from Q) do
2     Let qCluStr, qCluLen, and qCluLA be CluStr, CluLen and CluLA of q_i, respectively;
3     N_i := ExecuteQuery("select * from CST_k where CluStr = qCluStr
                                and CluLen = qCluLen and CluLA = qCluLA");
4     if (count(N_i) < ε) then
          break;
5 Merge all N_i into N using ID and Loc as joining attributes;
6 answers := PostProcessing(N);
7 return answers;
```

segments of q_i. If q_i consists of the first 2^k segments of a query, then qLb and qUb are expressed as $qLb = Lb_1 + Lb_2 + ... + Lb_{2^k}$ and $qUb = Ub_1 + Ub_2 + ... + Ub_{2^k}$. Therefore, when the difference of qLb and qUb is large, the cost for processing q_i becomes high due to an enlarged search space for CluLen.

To overcome the problem of an enlarged search space of a subquery q_i, we propose the *selective clustering method* (SCM) where q_i is decomposed into a set of *secondary* subqueries and then a secondary subquery with the smallest *estimated* search space is chosen and executed in replacement of q_i. In detail, when a subquery q_i has 2^k segments, its secondary subqueries are generated from $2^{k'}$ underlying segments of q_i for each k' in $[0, k]$. Figure 3 shows the secondary subqueries from a subquery q_i.

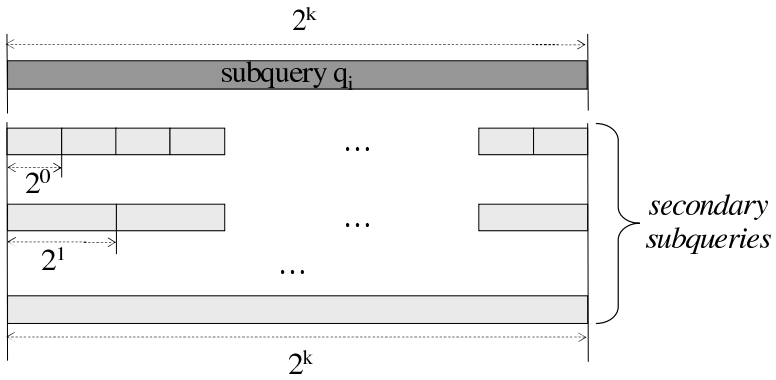

Figure 3. Secondary subqueries from a subquery q_i.

For example, let us consider the query $Q =< E(3\ 5)H(3\ 6)L(3\ 7) >$ where $MaxK = 1$ and $MaxCluLA = 2$. Then, k is computed as $k = min(\lfloor log_2 3 \rfloor, 1) = 1$. Thus, every subquery has 2^1 segments and the subquery q_1 is expressed as q_1 = ($qCluStr = EH$, $qCluLen = [3 + 3, 5 + 6]$, $qCluLA = L$).

Let $q_{i,j}$ denote the j^{th} secondary subquery of the i^{th} subquery. Then, in SCM, three secondary subqueries are generated from q_1. When $k' = 0$, we obtain two secondary subqueries, $q_{1,1}$ = $(qCluStr = E,\ qCluLen = [3,5],\ qCluLA = HL)$ and $q_{1,2}$ = $(qCluStr = H,\ qCluLen = [3,6],\ qCluLA = L)$, each of which has 2^0 segment in it. Similarly, when $k' = 1$, we obtain one secondary subquery, $q_{1,3}$ = $(qCluStr = EH,\ qCluLen = [6,11],\ qCluLA = L)$, which has 2^1 segments in it. Among these secondary subqueries, we choose the most selective one by estimating the number of tuples to be retrieved by each secondary subquery. If $q_{1,2}$ is predicted to retrieve the smallest number of tuples, for example, then it is executed in replacement of q_1. The same procedure is applied to q_2 = $(qCluStr = HL,\ qCluLen = [3+3, 6+7])$ also.

The effectiveness of SCM depends on the conciseness of an estimation process as well as the accuracy of estimated results. To achieve conciseness while maintaining accuracy at a satisfactory level, we keep two histograms separately, one for CluLen and the other for CluStr, and combine them whenever necessary. Both of these histograms are updated whenever sequences are inserted into clustered segment tables. The schemas for these two histograms are shown in Table 4, and the algorithm for estimating the number of tuples to be retrieved by each secondary subquery is presented in Algorithm 2.

Table 4. Schemas of CluLen and CluStr histograms.

CluLen Histogram

Field Name	Description
hK	k value of a cluster
hCluLen	CluLen value of the cluster
#Clusters	Number of clusters whose k value is the same as hK and CluLen value is the same as hCluLen

CluStr Histogram

Field Name	Description
hK	k value of a cluster
hCluStr	CluStr value of the cluster
#Clusters	Number of clusters whose k value is the same as hK and CluStr value is the same as hCluStr

Algorithm 2: Estimate_SizeOf_ResultSet

Input : Secondary subquery $q_{i,j}$

Output : Predicted number of tuples in the result set

1 Let $qCluStr$ be $CluStr$ of $q_{i,j}$;
2 Let [qLb, qUb] of $q_{i,j}$ be the range of $qCluLen$;
3 Suppose that $q_{i,j}$ is composed of 2^{qK} segments;
4 N_1 := ExecuteQuery("select sum(#Clusters) from CluStrHistogram where hK=qK");
5 N_2 := ExecuteQuery("select #Clusters from CluStrHistogram where hK=qK
and hCluStr=qCluStr");
6 N_3 := ExecuteQuery("select #Clusters from CluLenHistogram where hK=qK
and hCluLen between qLb and qUb");
7 return $N_3 \times N_2 / N_1$;

4.4. Wildcard match query

Wildcard match queries are specified as $Q =< T_1(Lb_1\ Ub_1)\ T_2(Lb_2\ Ub_2)\ \ldots\ T_{N_Q}(Lb_{N_Q}\ Ub_{N_Q}) >$ where T_i takes a value from $\{E, H, L, ?\}$. The meanings of Lb_i and Ub_i are same as before. Note that T_i may take '?' to express that the i^{th} segment can be of any secondary structure type. To accommodate this wildcard type, we just use 'CluStr like qCluStr' predicate in Algorithm 1 (Line 3) instead of 'CluStr=qCluStr' predicate.

We also apply SCM to this type of query because it has the problem of an enlarged search space for both CluStr and CluLen. Since qCluStr may contain '?', we use 'hCluStr like qCluStr' predicate in Algorithm 2 (Line 5) instead of 'hCluStr=qCluStr'.

5. Performance Evaluation

5.1. Experimental environment

We used two Pentium-4 PCs for experiments, each of which was equipped with a 512 MB main memory and a 7200 rpm hard disk. A commercial RDBMS Oracle-8i was installed in one PC and a set of protein sequences was stored in it. Segment table, clustered segment tables, and indices for segment-based indexing method [11] and CSI were built. In another PC, we implemented searching algorithms.

To obtain the secondary structures of proteins, we applied PREDATOR [8, 9] to the amino acid arrangements of proteins downloaded from PIR [20].

To verify the effectiveness of the proposed method, CSI was compared to MISS(1), MISS(2), and SSS. MISS(n) chooses the most selective n segments from a query and treats each of them as a subquery.It then executes each subquery using a B^+-tree on the segment table. SSS chooses the most selective segment from a query and executes it by performing a full table scan on the segment table.

5.2. Parameter setting

To determine the values of two system parameters, $MaxK$ and $MaxCluLA$, we first performed the preliminary experiments with a data set of 80,000 protein sequences. Remember that the parameter $MaxK$ is to control the total number of clustered segment tables being constructed and the parameter $MaxCluLA$ is to control the maximum length of lookahead.

5.2.1. MaxK

The *selectivity*, a ratio of the number of tuples retrieved by a search to the total number of tuples stored in a table, was used as a measure for determining the optimal value of $MaxK$. Figure 4 shows the selectivity of a pair (CluStr, CluLen) for each clustered segment table CST_k. It is clear from the figure that the selectivity becomes better as k increases but it is saturated after k exceeds 3. Therefore, we set 3 as the optimal value of $MaxK$.

5.2.2. MaxCluLA

The selectivity becomes better as the value of $MaxCluLA$ increases. To measure the degree of improvement in selectivity (DIS), we use the following formula:

$$DIS(\%) = \frac{selectivity\ of\ (CluStr, CluLen) - selectivity\ of\ (CluStr, CluLen, CluLA)}{selectivity\ of\ (CluStr, CluLen)} \times 100$$

Figure 5 shows the degree of improvement in selectivity for each value of $MaxCluLA$. According to the result, the degree of improvement grows as the value of $MaxCluLA$ increases but the growth is almost saturated after the value of $MaxCluLA$ exceeds 8. Therefore, we set 8 as the optimal value of $MaxCluLA$.

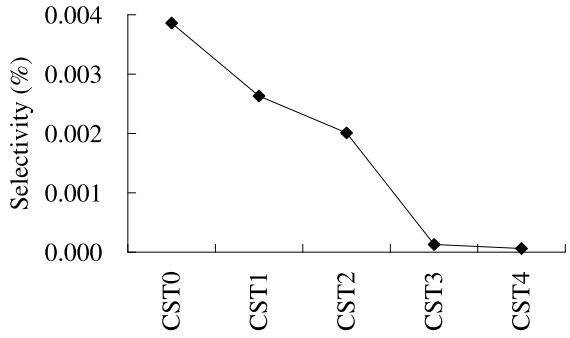

Figure 4. Selectivity of (CluStr, CluLen) for each clustered segment table CST_k.

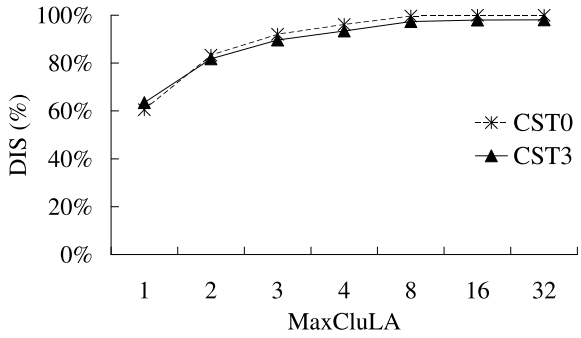

Figure 5. Degree of improvement in selectivity (DIS) for each value of $MaxCluLA$.

5.3. Accuracy of CluStr and CluLen histograms

To demonstrate the accuracy of CluStr and CluLen histograms in predicting the number of tuples to be retrieved by a secondary subquery, we used two variables: (1) *error rate* which is defined as $\frac{|N_a - N_p|}{N_a}$ where N_a is the actual number of tuples retrieved by a secondary subquery and N_p is the predicted value of N_a obtained from CluStr and CluLen histograms, and (2) *correlation* which is a measure of linear relationship between N_a and N_p.

As shown in Figure 6, the error rate is in the range of 0.9 and 1.3. But the correlation is at least 0.9, which shows that N_a and N_p have high linear dependence on each other. Therefore, although the error rate is not small, it is highly likely that a secondary subquery with the smallest N_a value must have the smallest N_p value.

5.4. Index size

We compared the storage requirement of CSI with that of SSS and MISS(n). Because SSS does not use any other index other than the segment table, its storage consumption is the same as the size of the segment table. In addition to the segment table, MISS(n) uses the B^+-tree built on the type and the length attributes of segments. Therefore, the storage requirement of MISS(n) is the sum of the sizes of the segment table and its B^+-tree. CSI utilizes the clustered segment tables and the B^+-tree indexes built

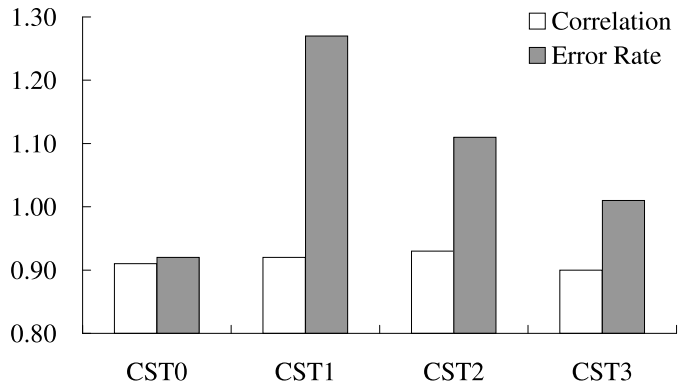

Figure 6. Error rate and Correlation of the proposed prediction scheme for each clustered segment table CST_k.

on the type string and the length attributes of clusters. Therefore, its storage consumption is the sum of the sizes of the clustered segment tables and their associated B^+-tree indexes. According to the result shown in Table 5, all the methods consume storage space almost proportional to the number of protein sequences. The storage requirement of CSI is several times larger than that of other three methods, but it is not a big problem when considering the continuous drop of storage costs.

Table 5. Index size comparison of SSS, MISS(n), and CSI (in KBytes).

Number of Sequences	SSS	MISS(n)	CSI
20,000	20,288	35,626	149,640
40,000	37,160	65,258	272,350
60,000	51,992	91,304	378,907
80,000	67,688	118,852	481,516

5.5. Index selectivity

In Section 1, we claimed that the tuples of clustered segment tables are more discriminative than those of segment table. To prove this claim, we compared them in terms of selectivity. The *selectivity* of a clustered segment table CST_k is defined as a ratio of the average number of tuples retrieved by a search condition on (CluStr, CluLen, CluLA) to the total number of tuples stored in CST_k, and the *selectivity* of a segment table is defined as a ratio of the average number of tuples retrieved by a search condition on (Type, Len) to the total number of tuples stored in the segment table. According to the result shown in Figure 7, the selectivity of CST_k is lower than that of segment table at least about 33 times. In other words, given a query, the expected number of rows returned from CST_k is $\frac{1}{33}$ times as many as segment table. This reduction in selectivity decreases the number of candidates significantly that are to be verified at the post-processing step.

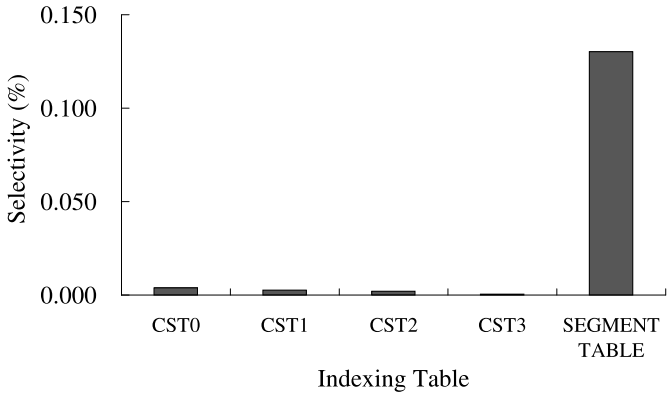

Figure 7. Selectivity comparison between the CST_k table and the segment table.

5.6. Query processing time

5.6.1. Query processing time with various numbers of segments in a query

While changing the number of segments, N_Q, in a query, we measured the query processing times of four methods, CSI, MISS(1), MISS(2), and SSS. For this experiment, we used a data set of 80,000 protein sequences from which queries were randomly extracted. Note that segment lengths of exact match queries must be specified exactly but those of range match and wildcard match queries can be specified as a range. For the simplicity of experimentations, we let only the segment in the middle of range match and wildcard match queries have a range in its length specification. Considering the distribution of segment lengths, we set the size of the range as 30. In case of wildcard match queries, only the segment in the middle had the wildcard character '?'. Figures 8, 9, and 10 shows the query processing times of four methods for exact match, range match, and wildcard match queries, respectively.

According to the experimental results, query processing times of all methods decrease as N_Q increase. This is because more segments with high selectivity are contained in queries as N_Q increases. If N_Q is large, CSI gets extra benefit by choosing a larger k value when deciding a clustered segment table to be searched. As a result, CSI was 1.7∼13.0 times, 1.3∼6.0 times, and 1.0∼3.4 times faster than the best one of the other methods in exact match, range match, and wildcard match, respectively.

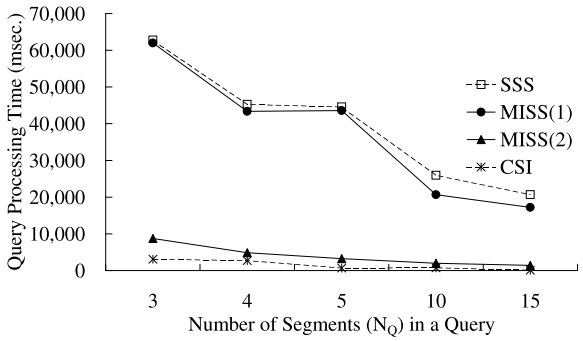

Figure 8. Exact match query processing times with increasing N_Q.

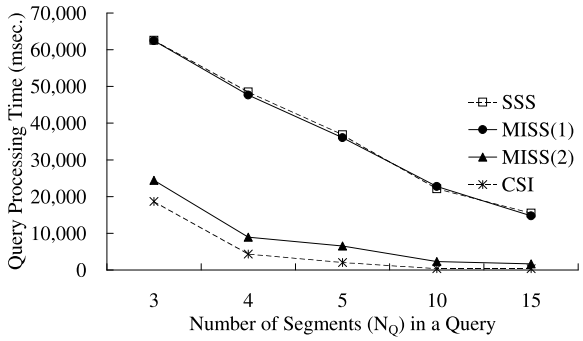

Figure 9. Range match query processing times with increasing N_Q.

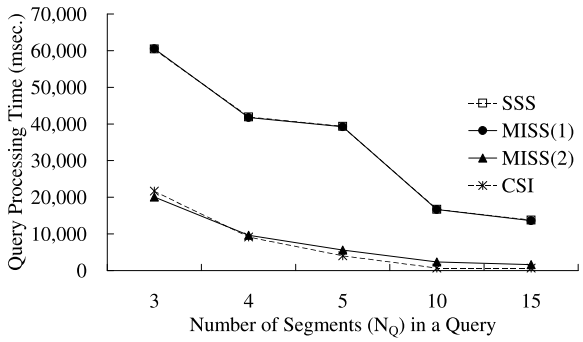

Figure 10. Wildcard match query processing times with increasing N_Q.

5.6.2. Query processing time with various data set sizes

While increasing the number of protein sequences, N_{seq}, from 20,000 to 160,000, we measured the query processing times of CSS and MISS(2). SSS and MISS(1) were not included in this experiment because they proved to be less efficient than MISS(2) in most cases. The number of segments in a query was 5, and ranges and wildcard characters were given only to the third segment. According to the experimental results shown in Figures 11, 12, and 13, the query processing times of both CSI and MISS(2) were proportional to the data set size, and CSI was 4.3∼6.3 times, 3.0∼3.3 times, and 1.4∼1.5 times faster than MISS(2) in exact match, range match, and wildcard match, respectively.

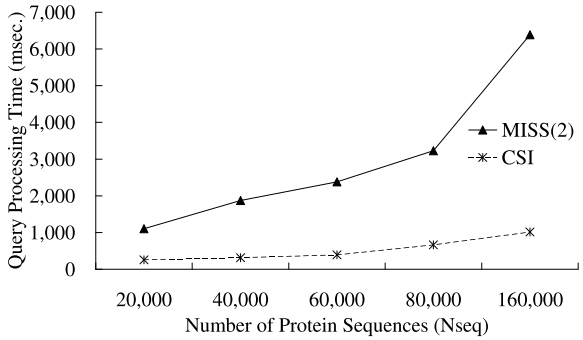

Figure 11. Exact match query processing times with increasing N_{seq}.

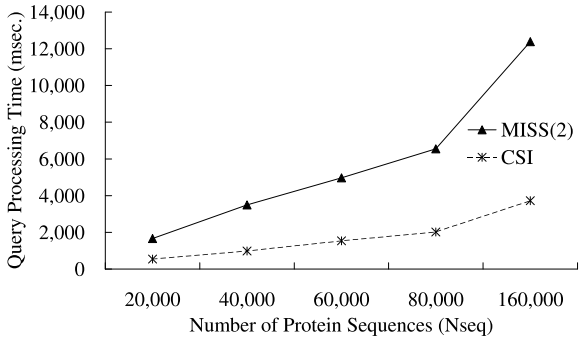

Figure 12. Range match query processing times with increasing N_{seq}.

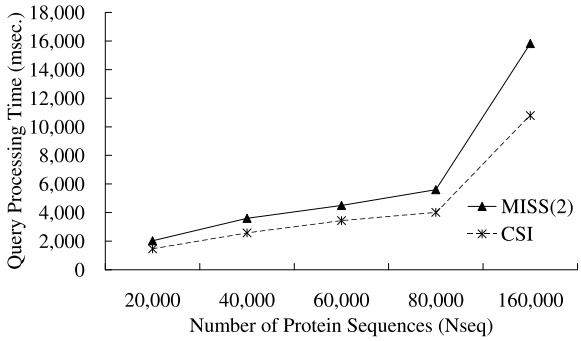

Figure 13. Wildcard match query processing times with increasing N_{seq}.

6. Computational Analysis

In this chapter, performance of CSI is analyzed and compared to the previous segment based indexing method. For the simplicity, CluLA attributes of clusters are ignored if not mentioned otherwise, and only Exact Match was considered. Still, we believe that the following analyses suffice for the purpose of computational performance comparisons.

6.1. Selectivity Comparison

As the first step of analyses, we compare the theoretical selectivity of clusters and segments. For that purpose, we assume that the Type of a segment is randomly chosen from $\{E, H, L\}$. Also, Len is assumed to be chosen from [0, 100] randomly. Then, the number of distinct (Type, Len) pair is 300 (=3×100).

To the contrary, a cluster is represented by (CluStr, CluLen) pair. Because we gather together 2^k segments for a cluster in CST_k table, the length of CluStr is 3^{2^k}. It is also followed that $CluLen \in [0, 100 \times 2^k]$. Therefore, the number of distinct (CluStr, CluLen) pairs is $3^{2^k} \cdot 100 \cdot 2^k$. Figure 14 depicts the number of distinct segments or clusters for CST_k and Segment table.

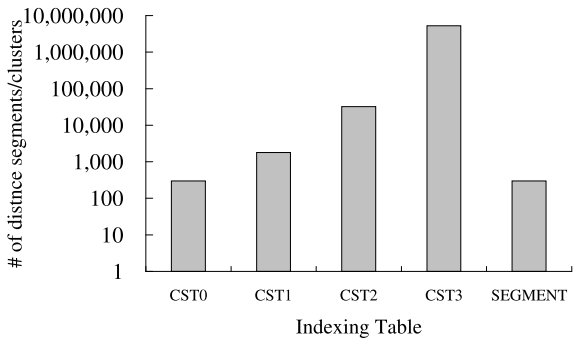

Figure 14. The number of theoretically distinct segments/clusters

6.2. Complexity Comparison

In broad perspective, the purpose of MISS(1), MISS(2), SSS, and CSI, is to filter out as many sequences as possible in the index stage such that only few sequences are retrieved and post processed. The advantages of avoiding postprocessing are two fold. Firstly, postprocessing requires very expensive random IOs. Suppose that there are 100 candidate answers. Then, it is required to retrieve 100 sequences from 100 places, which incurs 100 random I/Os. Secondly, verifying whether a sequence match a segment/cluster takes lots of computational time. To verify a sequence, the sequence has to be parsed into segments/clusters, and then matched against query specified as a sequence of segments or clusters. Therefore, in this section, we delve into the comparison of filtering effectiveness which plays a crucial role in reducing IOs and computations.

Let us suppose that the number of segments in a sequence is 100. If a query is given by a user, the first step is to find out the most selective segment from the query. Then, the segment will be matched against all segments stored in the segment table. In case of segments, the probability that a segment exactly matches another segment is $\frac{1}{3\times 100}$. Now, to filter out a sequence which is comprised of 100 segments, there must be no matching segments in the sequence, and such a probability is $(1-\frac{1}{300})^{100}$.

On the other hand, $100-2^k+1$ clusters will be generated from a sequence, and the probability that two clusters are the same is $\frac{1}{3^{2^k}\cdot 100\cdot 2^k}$. Hence, the probability of filtering a sequence out is $(1-\frac{1}{3^{2^k}\cdot 100\cdot 2^k})^{100-2^k+1}$. Figure 15 compares expected filtering effectiveness based on the above calculations.

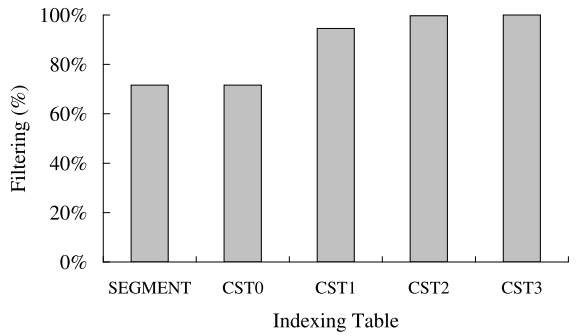

Figure 15. The percent of sequences expected to be filtered out

6.3. Index Size Comparison

The improved search speed of CSI comes at a price. In this section, we analyze the size of segment table and clustered segment tables.

Suppose that there are 100 segments in a sequence. Then, in case of a segment table, 100 rows are needed to store the sequence. In each row, 5 values are stored: SegID(int), ProteinID(int), Loc(int), Type(char), and Len(int). Assuming that the size of integer is 4 bytes and the size of character is 1 byte, $17 \times 100 \times n = 1700 \cdot n$ bytes are required for storing n sequences.

Clustered segment tables stores five values for a cluster: ID(int), Loc(int), CluStr(char array), CluLen(int), and CluLA(char array). Given a sequence of 100 segments, $100 - 2^k + 1$ clusters are populated. Hence, $(8 + 2^k + 8) \times 100 \times n$ bytes are required for n sequences in CST_k, assuming that CluLA contains 8 characters. If we take 3 as the value of $MaxK$, then the total size of clustered segment tables is as follows:

$$100 \cdot n \times \sum_{k=0}^{3}(16 + 2^k) = 7900 \cdot n \tag{1}$$

Based on the above observation, although the storage consumption of CSI is about 4.5 times larger that that of segment table, it is still linear. It is also possible to limit the storage consumption by setting $MaxK$ with smaller value like 2. In that case, the storage consumption of CSI is about 3.2 times as large as that of segment table.

7. Conclusion

Approximate searching on protein structures, rather than on amino acid arrangements, are essential in finding out distant homology. In this paper, we proposed CSI, an efficient indexing scheme for approximate searching on the secondary structure of protein sequences. The proposed indexing scheme exploits the concept of clustering and lookahead to improve the selectivity of indexing attributes. Algorithms for exact match, range match, and wildcard match queries were also proposed and evaluated. The experimental results revealed that CSI is faster than MISS(2) up to 6.3 times, 3.3 times, and 1.5 times in exact match, range match, and wildcard match queries, respectively.

Acknowledgements

This work was supported by Korea Research Foundation Grant funded by Korea Government (MOEHRD, Basic Research Promotion Fund)(KRF-2004-003-D00302 and KRF-2005-206-D00015).

References

[1] B. Alberts, D. Bray, J. Lweis, M. Raff, K. Roberts, and J. D. Watson (3rd), *Molecular Biology of the Cell* (Garland Publishing Inc., 1994).

[2] 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.

[3] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman, Basic Local Alignment Search Tool, *Journal of Molecular Biology* (1990) 403–410.

[4] Z. Aung, W. Fu, and K.-L. Tan, An Efficient Index-based Protein Structure Database Searching Method, *Proc. IEEE DASFAA Conf.* (2003) 311–318.

[5] O. Camoglu, T. Kahveci, and A. K. Singh, Towards Index-based Similarity Search for Protein Structure Databases, *Proc. IEEE Computer Society Bioinformatics Conf.* (2003) 148–158.

[6] I. Eidhammer and I. Jonassen, Protein Structure Comparison and Structure Patterns - An Algorithmic Approach, *ISMB tutorial* (2001).

[7] C. Fondrat and P. Dessen, A Rapid Access Motif Database(RAMdb) with a Searching Algorithm for the Retrieval Patterns in Nucleic Acids or Protein Databanks, *Computer Applications in the Bioscience* 11(3) (1995) 273–279.

[8] D. Frishman and P. Argos, Seventy-five Accuracy in Protein Secondary Structure Prediction, *Proteins* 27(3) (1997) 329–335.

[9] D. Frishman and P. Argos, Incorporation of Long-Distance Interactions into a Secondary Structure Prediction Algorithm, *Protein Engineering* 9(2) (1996) 133–142.

[10] J. F. Gibrat, T. Madel, and S. H. Bryant, Surprising Similarities in Structure Comparison, *Current Opinion in Structural Biology* 6(3) (1996) 377–385.

[11] L. Hammel and J. M. Patel, Searching on the Secondary Structure of Protein Sequence, *Proc. VLDB Conf.* (2002) 634–645.

[12] L. Holm and C. Sander, Protein Structure Comparison by Alignment of Distance Matrices, *Journal of Molecular Biology* 233(1) (1993) 123–138.

[13] E. Hunt, M. P. Atkinson, and R. W. Irving, Database Indexing for Large DNA and Protein Sequence Collections, *The VLDB Journal* 11(3) (2002) 256–271.

[14] P. Koehl, Protein Structure Similarities, *Current Opinion in Structural Biology* 11(3) (2001) 348–353.

[15] D. W. Mount, *Bioinformatics* (Cold Spring Harbor Laboratory Press, 2000).

[16] A. Pastore and A. Lesk, Comparison of Globins and Physocyanins: Evidence for Evolutionary Relationship, *Proteins: Struct., Func., Gen.* 8(2) (1990) 133–155.

[17] G. A. Stephen, *String Searching Algorithms* (World Scientific Publishing, 1994).

[18] H. Wang, C.-S. Perng, W. Fan, S. Park, and P. S. Yu, Indexing Weighted Sequences in Large Databases, *Proc. IEEE ICDE Conf.* (2003) 63–74.

[19] H. E. Williams, Genomic Information Retrieval, *Proc. Australasian Database Conf.* (2003) 27–35.

[20] C. H. Wu, L.-S. L. Yeh, H. Huang, L. Arminski, J. Castro-Alvear, Y. Chen, Z. Hu, P. Kourtesis, R. S. Ledley, B. E. Suzek, C. R. Vinayaka, J. Zhang, and W. C. Barker, The Protein Information Resource, *Nucleic Acids Research* 31(1) (2003) 345–347.