CSI: Clustered Segment Indexing For Efficient Approximate Searching On The Secondary Structure of Protein Sequences

Minkoo Seo, Sanghyun Park, and Jung-Im Won

Department of Computer Science
Yonsei University, Korea
{mkseo,sanghyun,jiwon}@cs.yonsei.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 the 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. 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 6.3 times faster in exact match, 3.3 times faster in range match, and 1.5 times faster in wildcard match compared to the existing indexing methods.

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

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][20].

However, proteins, in evolutionary history, rarely conserve the amino acid arrangment, while retaining its structure[6][16][17]. 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 popularity in conjunction with database approaches[6][15].

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][12]. For example, '$HLLELLEELH$' is less likely to occur than '$HHLLLLLLEEE$'.

Exploiting this property, Hammel et al.[12] proposed a segment-based indexing method. The method combines consecutive characters into a single segment and then builds a B^+-tree on the two attributes of segments: Type (character) and Len (number of occurrences of the character). For example, '$HHLLLLLEEE$' is segmented into '$HH/LLLLL/EEE$' and represented as $(H, 2)(L, 5)(E, 3)$.

Although the segmentation enables an efficient searching on the secondary structures, it has innate limitations. First, (Type, Len) pairs do not have uniform distribution. According to our preliminary experimentation on 80,000 secondary structures, almost all Len of segments are between 1 and 100. However, 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*, each index search will produce lots of intermediate results and thus the overall search performance will certainly be bad. Secondly but more importantly, the number of distinct (Type, Len) pairs is not large enough to provide good selectivity. Because there are three Types (E,H, and L) and almost all segments have the Len between 1 and 100, there are 3,000 possible segments. In addition, the number of segments increases as the number of sequences increases. As a result, all segments from proteins are categorized into the one of 3,000 (Type, Len) pairs and the finding a (Type, Len) pair will return 'the number of segments in the database/3,000', which might be very large if there are many proteins, on average.

In this paper, we propose 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. The clustering combines several Type occurrences of segments into a string to increase the number of Type occurrence categories where segments can be belonged. Also, information about the succeeding segments of a given segment is stored as a field of a cluster to enhances search performance.

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[20] : 1) entire data set should be loaded into a main memory for fast searching, 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[18] have been recognized as the best index structure for string or sequence searching, but Suffix trees have been notorious for its storage consumption. Recently, algorithms for building a suffix tree from a data set larger than a main memory was proposed[14]. However, the internal structure of a suffix tree is not suitable for pagination and therefore it is not easy to incorporate suffix trees into database systems[18][19].

RAMdb[8] is an indexing system for the primary structures of protein sequences and was proved to be faster than heuristic approaches up to 800 times by experiments. However, queries in RAMdb should be of a length close to the length 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 ideas of RAMdb directly to the secondary structures of protein sequences.

Hammel et al.[12] proposed a 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 segment length and the character type. As mentioned in the previous section, however, this segment-based approach does not support good selectivity, resulting in an innate limitation of search performance.

DALI[13] and VAST[11] are also structure-structure similarity search programs which provide three-dimensional superpositions. VAST uses vectors derived from secondary structure elements, with no sequence information being used. VAST, like BLAST, is run on all entries in the database in an $N \times N$ manner, and DALI looks for similar contact patterns between two proteins using matrices, and return the best set of structure alignment solutions[5]. Both of them are different from CSI, introduced in this paper, in that 1) they consider not only the structure but also the spatial location of the each structure, 2) they return alignments among the given query and structures in database, but CSI returns sequences which meet the given segment pattern.

3 Indexing Metod

3.1 Segment Table

In the segment table, each segment is expressed by two attributes: the type of consecutive characters and the length of the segment. For example, a sequence S_1 = '$EEEHHLL$' is segmented into '$EEE/HH/LL$' and then expressed as $(E, 3)(H, 2)(L, 2)$. 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 three positions, 0, 3, and 5, 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 = '$EEEHHLL$.'

Table 1. Segment table for the sequence S_1 ='$EEEHHLL$'

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

3.2 Clustered Segment Table(CST)

Rather than using the segment table, we use a set of clustered segment tables. The procedure to construct a clustered segment table is as follows:

1. For each sequence S, convert S into a series of segments. Let N_S be the number of segments obtained from the sequence S.
2. Using the sliding window of size 2^0, generate a set of clusters, each of which is composed of 2^0 neighboring segments. Store each cluster into the clustered segment table named CST_0.
3. Repeat the above step using sliding windows of size 2^k where $1 \leq k \leq min(\lfloor log_2(N_s) \rfloor, MaxK)$. Here, $MaxK$ is a parameter which can be determined by the storage availability of a user. The resultant set of clusters is stored in the clustered segment table named CST_k.

A cluster, generated in step 2, consists of neighboring segments. In detail, let ($Type_1$, Len_1)($Type_2, Len_2$)...($Type_{2^k}, Len_{2^k}$) be 2^k neighboring segments. Then a cluster can be generated as ($CluStr = Type_1 \cdot Type_2 \cdot ... \cdot Type_{2^k}, CluLen = Len_1 + Len_2 + ... + Len_{2^k}$).

There may be a series of segments following a cluster. Then, the characters expressing the types of such segments can be concatenated, producing, CluLA, the lookahead of the cluster. Like MaxK of CluStr, the length of CluLA is limited by a predetermined parameter, *MaxCluLA*, for space efficiency. The schema for each clustered segment table is shown in Table 2.

Table 2. Schema of clustered segment table

Field Name	Description
ID	The identifier of the protein from which the cluster is made.
Loc	The beginning position of the cluster.
CluStr	The string obtained by concatenating the characters for the underlying segments.
CluLen	The length of the cluster obtained by summarizing the lengths of the underlying segments.
CluLA	The string obtained by concatenating the characters for the segments following the cluster.

For example, CST_0 and CST_1 tables are constructed from a sequence $S_1 =$ '$EEEHH\ LLEEE$' when $MaxK = 1$ and $MaxCluLA = 2$ like Table 3.

Table 3. Clustered segment table CST_0 and CST_1

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	

After populating all the tuples of CSTs, the tuples are sorted according to 'CluStr', 'CluLen', and 'CluLA' fields for the sake of locality. As the final step, B^+-trees are built on CluStr and CluLen of 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 2bits like: L=00_2, H=10_2, and E=11_2.

4 Query Processing

4.1 Overall Query Processing Procedure

Suppose that there CST_0, CST_1, ..., and CST_{MaxK}. If we denote the number of segments in a query Q as $|Q|$, the overall query processing procedure is as follows:

1. To compute the k value, which is required for determining a CST table and a subquery length, use the following formula: $k = min(\lfloor log_2|Q| \rfloor, MaxK)$.

2. Decompose the given query, Q, into $n(=\lceil |Q|/2^k \rceil)$ non-overlapping subqueries such that each of subquery has 2^k segments in it. However, to make the length of the last query 2^k, the last subquery is allowed to be overlapping with the previous one.
3. Search each subquery q_i (i=1,2,. . . ,n) using CST_k and generate intermediate results.
4. Merge n intermediate results into a candidate answer set.
5. Postprocess the candidate answer set and return a final result.

4.2 Exact Match Query Processing

Exact Match is a basic query and can be expressed as $Q =< S_1(cnt_1)S_2(cnt_2)\ldots S_n(cnt_n) >$ where $S_i(\in \{E, H, L\})$ represents the segment type of the ith segment and cnt_i is the length of the ith segment.

Suppose that we determined k value and that Q is decomposed into n subqueries, each of which is size of 2^k, using the procedure described in section 4.1. Algorithm Exact_Match searches proteins using cluster segment table CST_k which matches Q.

Algorithm 1: Algorithm Exact_Match

Input : Q: Query, CST_k: Clustered segement table k, ϵ: Searching threshold
Output: Set of answers

```
1  begin
2  for (each subquery q_i from Q) do
3      Let qCluStr, qCluLen, and qCluLa be CluStr, CluLen, and CluLA of q_i, respectively
4      N_i:=Execute_Query("select * from CST_k where CluStr=qCluStr
                           and CluLen=qCluLen and CluLA=qCluLA")
5      if (count(N_i) < ε) then
           break;
6  End For
7  Merge all N_i into N using ProteinID, Loc, and CluLen
8  answers := PostProcessing(N)
9  return answers
10 end
```

After searching a q_i using CST_k, an intermediate result is stored in N_i (Line 2-4). If the number of elements in N_i is less than predefined threshold ϵ, we stop searching, leaving $q_{i+1}, q_{i+2}, \ldots, q_n$ out (Line 5). After merging intermediate results into N, a post processing phase compares each of candidate answers with each sequence in the database (Line 7-8). The post processing step is mandatory because we set the CluLen value as the sum of each segments in a cluster; thus we lost the information about the length of each segment, resulting in false alarms.

As an example, suppose that $Q =< E(3)H(2)L(2) >$ and that CST_0 and CST_1 are given, which means $MaxK = 1$, as Table 3. Then, we can compute the k value as $k = min(\lfloor log_2 3 \rfloor, 1) = 1$. The query Q can be decomposed, using the k value, into two subqueries of size 2^1: $q_1 = (CluStr = EH, CluLen = 3 + 2 = 5, CluLA = L)$, $q_2 = (CluStr = HL, CluLen = 2 + 2 = 4, CluLA = null)$. In searching phase, we can find $N_1 = S_1(Loc = 0)$ for q_1 and $N_2 = S_1(Loc = 3)$ for q_2. (For the sake of convenience, we do not consider search threshold, ϵ, here.) For merging, the location difference between

$S_1(Loc = 0)$ of N_1 and $S_2(Loc = 3)$ of N_2 are compared with the difference between q_1 and q_2. Because the location difference is the same, we merge N_1 and N_2 into $N = S_1(Loc = 0)$. As a post processing, the string starting at the location of $S_1(Loc = 0)$ is compared with the query Q. There are no false alarms in this example, so we return $S_1(Loc = 0)$ as the result of Q.

4.3 Range Match Query Processing

Range Match queries are in the form of $Q =< S_1(Lb_1\ Ub_1)S_2(Lb_2\ Ub_2)\dots S_n(Lb_n\ Ub_n) >$ where Lb_i is the minimum length and Ub_i is the maximum length of the ith segment.

On the contrary to the queries of Exact Match, Range Match queries have the problem of *widened search space* because the length of a segment is given as the range, i.e., the minimum length of a CluLen is the sum of all Lb_i of each segment in a subquery and the maximum length is the sum of all Ub_i. For example, suppose that $Q =< E(3\ 5)H(3\ 6)L(3\ 7) >$ and that $MaxK = 1$. In this case, Q is decomposed into two subqueries of $q_1 = (CluStr = EH, CluLen = 6 \sim 11, CluLA = L)$ and $q_2 = (CluStr = HL, CluLen = 6 \sim 13, CluLA = null)$. At this point, the range of CluLen of q_1 is $6(= 11 - 6 + 1)$, and the range of CluLen of q_2 is 7. In other words, the CluLen is not a specific value, there by resulting in possible low searching performance.

We propose *selective clustering method(SCM)* to avoid such a problem. SCM generates subqueries $q_{i,j}$ from q_i and select one which is predicted to return the smallest result set as a representative query of q_i, and then use $q_{i,j}$ instead of q_i.

In detail, if a subquery q_i is consisted of 2^k segments, a secondary subquery $q_{i,j}$ is generated from 2^n neighboring segments for each n where $0 \leq n \leq k$. To illustrate, consider q_1 stated above. Because q_1 has 2^1 segments in it, secondary subqueries of size 2^0 and 2^1 are generated. For 2^0 size secondary subqueries, SCM generates $q_{1,1} = (CluStr = E, CluLen = 3 \sim 5, CluLA = HL)$ and $q_{1,2} = (CluStr = H, CluLen = 3 \sim 6, CluLA = L)$. For 2^1, $q_{1,3} = (CluStr = EH, CluLen = 6 \sim 11, CluLA = L)$ is generated. Then, most selective subquery is choosed as a representative of q_1. If $q_{1,2}$ is predicted to return smaller result set than $q_{1,1}$ and $q_{1,3}$, then SCM searches $q_{1,2}$ instead of searching q_1.

To apply SCM method efficiently, we need to predict the size of result set returned by a secondary subquery exactly. For this, we use CluLen histogram and CluStr histogram (Table 4) populated from CSTs and algorithm 2.

Table 4. CluLen, CluStr histogram

CluLen Histogram

Field Name	Description
k	k value of clusters
CluLen	CluLen value of clusters
#Clusters	Number of clusters generated from 2^k segments and CluLen value of each of which is the same as the field CluLen

CluStr Histogram

Field Name	Description
k	k value of clusters
CluStr	CluStr value of clusters
#Clusters	Number of clusters generated from 2^k segments and CluStr value of each of which is the same as the field CluStr

As an example of predicting the size of result sets, consider secondary subqueries mentioned above. For $q_{1,1}$, $|q_{1,1}| = T_{0,3\sim5} \times T_{0,E}/T_0$. Similarly, $|q_{1,2}| = T_{0,3\sim6} \times T_{0,H}/T_0$, and $|q_{1,3}| = T_{1,6\sim11} \times T_{1,EH}/T_1$.

Algorithm 2: Algorithm Estimate_SizeOf_ResultSet

Input : $q_{i,j}$: a secondary subquery
Output: Predicted size of result set

```
1 begin
2 Let qCluStr and qCluLen be CluStr and CluLen of q_{i,j}, respectively
3 Assume that q_{i,j} is composed of 2^{qk} segments
4 T_{qk} := Execute_Query("select sum(#Clusters) from CluStrHistogram where k=qk")
5 T_{qk,qCluStr} := Execute_Query("select #Clusters from CluStrHistogram where k=qk
                                   and CluStr=qCluStr")
6 T_{qk,qCluLen} := Execute_Query("select #Clusters from CluLenHistogram where k=qk
                                   and CluLen=qCluLen")
7 Return T_{qk,qCluLen} × T_{qk,qCluStr}/T_{qk}
8 end
```

4.4 Wildcard Match Query Processing

Wildcard Match queries can use '?' to represent a gap and also can have ranges in CluLen like Range Match. Consequently, due to the range in CluLen, this type of queries has the problem of windened CluLen search range and also CluStr is widened due to wildcard characters. Therefore, we apply SCM approach to Wildcard Match queries. For example, if $Q =< E(3\ 5)?(3\ 6)E(5\ 6) >$ and $MaxK = 1$, for a subquery $q_1 = (CluStr = E?, CluLen = 6 \sim 11, CluLA = E)$, SCM generates following four secondary subqueries: $q_{1,1} = (CluStr = E, CluLen = 3 \sim 5, CluLA =?E)$, $q_{1,2} = (CluStr = H, CluLen = 3 \sim 6, CluLA = E)$, $q_{1,3} = (CluStr = L, CluLen = 3 \sim 6, CluLA = E)$, and $q_{1,4} = (CluStr = E?, CluLen = 6 \sim 11, CluLA = E)$. Among them, the most selective one will be used instead of q_1 like Range Match algorithm.

5 Performance Evaluation

5.1 Experimental Environment

We used two pentium 4 PCs, each of which was equipped with a 512MB main memory and a 7200rpm hard disk. In one PC, a commercial RDBMS Oracle 8i was installed, and protein sequences were stored on it. In the other, searching applications of CSI and Hammel et al. were installed.

For experimental data, we applied PREDATOR[9][10] to primary protein sequences downloaded from PIR[21]. Five data sets were populated: 20000, 40000, 60000, 80000, and 160000 sequences.

To assess the performance of CSI, we compared it to MISS(1), MISS(2), and SSS proposed by Hammel et al. MISS(n) searches most selective n segments from the segment table described in section 3 using a B^+-tree, then removes false alarms by a post processing. In SSS, one most selective segment is retrieved by a full table scan of the segment table, and false alarms are removed by a post processing.

5.2 Parameter Setting

Before the performance evaluation, we need to set the maximum k value of CST_k tables(MaxK) and the maximum length of CluLA(MaxCluLA). For this purpose, we used 80K sequences.

MaxK The *selectivity*, a ratio of the number of tuples returned by a search to the total number of tuples, was used as an index for determining the maximum value of k. Figure 1 shows the selectivity of (CluStr, CluLen). According the result, the selectivity became good as k increases, but it maintained an almost even value after $k \geq 3$. Therefore, in the following, we assign 3, where the selectivity is 0.002%, to MaxK.

MaxCluLA The *filtering* was used as an index for setting MaxCluLA. Figure 2 shows the filtering effect with varying MaxCluLA. In detail, the filtering value here is the percent of false alarms removed from tuples retrieved by (CluStr, CluLen). According to results, as MaxCluLA increased the filtering became more effective, but the effect became small when MaxCluLA was more than 8. Hence we assign MaxCluLA 8 where 98% of false alarms were removed from CST_3.

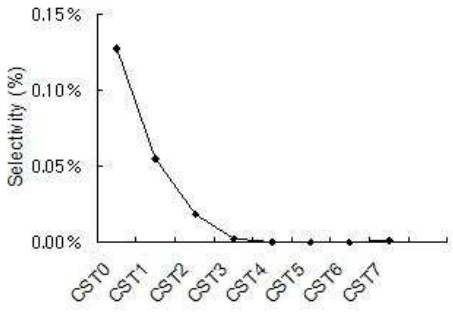

Fig. 1. The relationship between CST_k and the selectivity

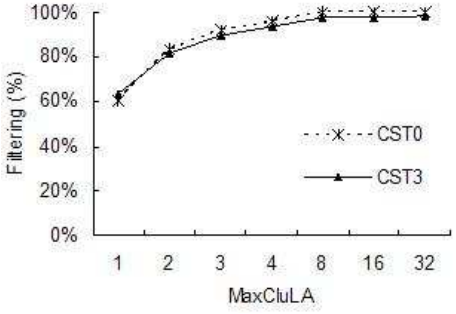

Fig. 2. The filtering effect and the maximum length of CluLA

5.3 Query Processing Time

Query Processing Time with Varying $|Q|$ We used 80K sequences for evaluating the performance with varying $|Q|$(number of segments in a query). Queries were produced by choosing sequences from actual sequences in the database randomly. In case of Range Match and Wildcard Match, we set $|Ub_{|Q|/2} - Lb_{|Q|/2}|$=30. Wildcard Match queries used here had a '?' character at the ($|Q|/2$)th segment. Figure 3 describes query processing times of Exact Match, Range Match, and Wildcard Match with varying $|Q|$.

According to experiments, query processing times of all algorithms decreased as $|Q|$ increased. This is because that the possibility of the existence of selective segments in queries become high as $|Q|$ increase. In case of CSI, in addition to the high possibility mentioned above, the increase of k resulted from larger $|Q|$ value makes it possible to choose more efficient CST_k. Hence, CSI has almost two-fold chances of gaining higher performance than the others. As a result, CSI was 1.7∼13.0 times, 1.3∼6.0 times, and 1.0∼3.4 times faster than MISS(2) in Exact Match, Range Match, and Wildcard Match, respectively.

Query Processing Time with Varying Data Size Figure 4 describes query processing time with variable data size. We set $|Q| = 5$ and used five data sets: 20K, 40K,

60K, 80K, and 160K sequences. Here, we compare CSI with MISS(2) only. According to experimental results, the query processing time of both CSI and MISS(2) were proportional to the data 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.

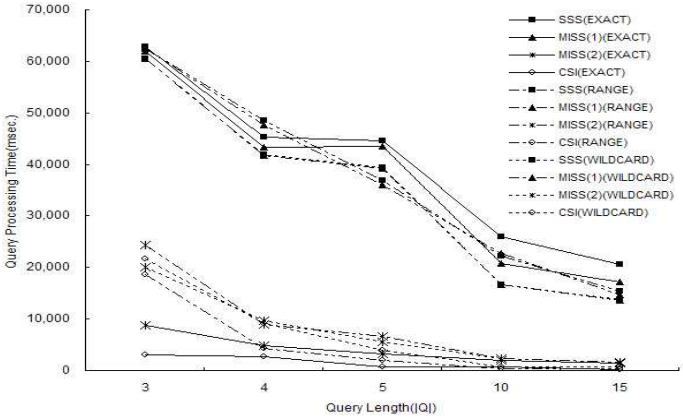

Fig. 3. Query Performance regarding $|Q|$ of Exact Match, Range Match, and Wildcard Match. Y Axes represent processing times in msec.

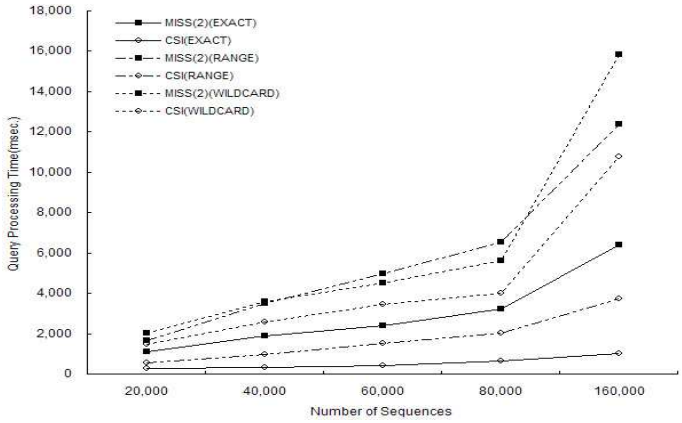

Fig. 4. Query Performance regarding data size of Exact Match, Range Match, and Wildcard Match . Y Axes represent processing times in msec.

6 Conclusion

The approximate searching on the secondary structure of protein sequences is the basis in predicting functions and roles of newly discovered unknown proteins.

Hence, in this paper, we propose CSI which targets the secondary structure of protein sequences and improves the selectivity by means of the clustering concept and look ahead. Algorithms for Exact Match, Range Match and Wildcard Match are also proposed and tested. According to our experimental results, CSI is 6.3 times faster in Exact Match, 3.3 times in Range Match and 1.5 times in Wildcard Match than MISS(2).

In the future, we would like to research protein structure similarity searching in depth.

References

1. B. Alberts, D. Bray, J. Lweis, M. Raff, K. Roberts, and J. D. Watson. Molecular Biology of the Cell, 3rd ed., 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.
3. S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman, "Basic Local Alignment Search Tool," J. Molecular Biology, 215: 403–410, 1990.
4. Z. Aung, W. Fu and K.-L. Tan, "An Efficient Index-based Protein Structure Database Searching Method," In Proc. DASFAA, IEEE 2003.
5. A. D. Baxevanis and B. F. F. Ouellette, BIOINFORMATICS: A Practical Guide to the Analysis of Genes and Proteins, 2nd ed., WILEY INTERSCIENCE, 2001.
6. Orhan Camoglu, Tamer Kahveci, Ambuj K. Singh, "Towards Index-based Similarity Search for Protein Structure Databases," CSB 2003.
7. I. Eidhammer and I. Jonassen, "Protein structure comparison and structure patterns - an algorithmic approach," ISMB tutorial, 2001.
8. 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 Bioscience, 11(3): 273–279, 1995.
9. D. Frishman and P. Argos, "75% accuracy in protein secondary structure prediction", in Proteins, 27:329-335, 1997
10. D. Frishman and P. Argos, "Incorporation of long- Distance interactions into a secondary structure prediction algorithm", in Protein Engineering, 9:133–142, 1996.
11. J. F. Gibrat, T. Madel, and S. H. Bryant, "Surprising similarities in structure comparison," Current Opinion in Structural Biology, 6:377-385, 1996.
12. L. Hammel and J. M. Patel, "Searching on the Secondary Structure of Protein Sequence," In Proc. VLDB Conf., 2002.
13. L. Holm and C. Sander, "Protein structure comparison by alignment of distance matrices," J. Mol. Biol., 233:123–138, 1993.
14. E. Hunt, M. P. Atkinson and R. W. Irving, "Database indexing for large DNA and protein sequence collections," The VLDB Journal, Vol. 11, No. 3, pp. 256–271, 2002.
15. P. Koehl, "Protein Structure Similarities," Current Opinion in Structural Biology, 11:348–353, 2001.
16. D. W. Mount, Bioinformatics, Cold Spring Harbor Laboratory Press, 2000.
17. A. Pastore and A. Lesk, "Comparison of globins and physocyanins: evidence for evolutionary relationship," Proteins: Struc., Func., Gen., 8:133-155, 1990.
18. G. A. Stephen, String Searching Algorithms, World Scientific Publishing, 1994.
19. H. Wang, C.-S. Perng, W. Fan, S. Park, and P. S. Yu, "Indexing Weighted Sequences in Large Databases," in Proc. 19th IEEE International Conference on Data Engineering, pp. 63–74, Bangalore, India, March, 2003.
20. H. E. Williams, "Genomic Information Retrieval" , In K.-D. Scheme and X. Zhou, Proc. Australasian Database Conference, Adelaide, Australia, 27–35, 2003.
21. C. H. Wu, L-S. L. Yeh, H. Huang, L.Arminski, J. Castro-Alvear, Y. Chen, Z-Z. Hu, Robert S. L., P. Kourtesis, B. E. Suzek, C. R. Vinayaka, J. Zhang, and Winona C. Barker. "The Protein Information Resource", in Nucleic Acids Research, 31: 345–347, 2003.