Accelerating Approximate Subsequence Search on Large Protein Sequence Databases

Jiong Yang	Wei Wang	Yi Xia[*]	Philip S. Yu
T.J. Watson Research	CS Department	CS Department	T. J. Watson Research
IBM	UNC at Chapel Hill	UCLA	IBM
jiyang@us.ibm.com	weiwang@cs.unc.edu	xiayi@cs.ucla.edu	psyu@us.ibm.com

Abstract

Bioinformatics has become an active research area in recent years. The amount of mapped sequences doubles every fourteen months. BLAST has been widely employed for retrieving sequences which has similar portion(s) to a given sequence. However, BLAST has to scan the entire database every time when a query is issued. This can be very time consuming especially when the database is large. In this paper, we study the problem on how to build a persistent index structure for protein sequences to support approximate match. The suffix tree has been proposed as a solution to index sequence database and has been deployed on organizing DNA sequences (Hunt et al. 2001). Unfortunately, it suffers from the problem of "memory bottleneck" that prevents it from being applied efficiently to a large database. The performance even degrades further for protein database due to a larger fanout at each node. Here, we employ an indexing structure, called **BASS-tree***, to support approximate match in sublinear time on a large protein database. We call this indexing method as sequence approximate match (SAM) index method. The search of approximate matches can be properly directed to the portion in the database with a high potential of matching quickly. It has been demonstrated in our experiments that the potential performance improvement is in an order of magnitude over alternative methods such as the BLAST algorithm and the suffix tree.*

1 Introduction

Due to the significant advance in molecular biology, analysis of biological sequences has become an active research area in recent years. The amount of mapped sequences grows at an exponential pace, while the flat files have continued to be used as the standard format to store the huge sequence database. Searching on the a large sequence database is usually carried out by sequentially scanning the entire database using a screening approach to identify the set of desired sequences (e.g., BLAST (Altschul et al., 1990, Altschul et al., 1997)). This approach inevitably suffers from a prolonged response time when dealing with a large amount of sequences, even though a biologist sometime may be able to localize the search to a relatively smaller scope. Alternative methods that can achieve a fast delivery (such as the q-grams based approaches (Burkhardt et al., 1999, Navarro et al., 2000, Ukkonen, 1992)) may only be suitable for search of *very high* similarity matches and have been applied mainly on DNA sequences. As a result, how to accelerate the search in a broader domain become a topic of great practical importance and challenge, especially when dealing with protein sequences where symbol mutations have to be taken into account. Building an appropriate index structure is one of the possibilities to achieve such a goal, which is the focus of our investigation in this paper. The following observations provide the motivation and feasibility.

- An proper index may help to direct the search to the high potential portion of matching in the database and hence avoid the scan of the entire database. This is particularly important in improving the performance of searching a large database since it may take hours to evaluate every sequence in a large database and the saving in computation cost increases in proportion to the size of the database.
- Despite the rapid growth in the database size, the existing sequences in the database are relatively stable. (Most updates to the database are adding new sequences.) This justifies the potential benefit of supporting indexed search on large biological sequence databases since the construction cost of the index structure can be easily amortized over many searches. This also suggests that the index structure needs to support not only efficient search but also efficient insertion of new sequences.

[*]Part of Yi Xia's work was supported under NSF grants IIS-0086116, ANI-0085773, and EAR-9817773.

Many researchers have suggested to use the suffix tree or suffix array as an index of the sequence database (Farach et al., 1998, Hunt et al., 2001, McCreight, 1976, Navarro and Baeza-Yates, 2000, Navarro and Baeza-Yates, 2001, Ukkonen, 1995). However, this approach may not be the ideal choice due to the following reasons.

- Efficient algorithms (Andersson and Nilsson, 1995, McCreight, 1976, Ukkonen, 1995) have been developed to build a suffix tree in memory on a relatively small sequence database. Successful attempts on building suffix tree on disk have been very limited (Farach et al, 1998) due to the high I/O cost incurred. With respect to the current database size, a persistent index structure (i.e., residing on disk) turns out to be a more appropriate choice.

- Even under the circumstance where a suffix tree on a large sequence database is available and properly resides on disk, the search of approximate match to a given pattern may still incur a large amount of random disk I/Os since the suffix tree was originally design for exact match and the set of similar patterns may scatter all over the suffix tree. This, to a certain extent, reduces the potential benefit that could be obtained from employing an index structure.

In Yang et al. (2002), a new string indexing structure of approximate substring match in a large string database, BASS-tree, is proposed. We found that this scheme can be easily adopted for the use of protein search in the bioinformatics domain. In this paper, we focus on how to apply the indexing structure to support *approximate match* in sublinear time on a large protein database. The key idea behind the design is that all *position points*[1] are grouped recursively into a fully balanced tree according to the similarities of the *subsequent* segments, where the similarities can be defined through a *substitution matrix* (Durbin et al., 1998). Each node in the tree has a bounded size and is designed to be fit in a (predefined) disk block. Leading positions of similar segments would share a common ancestor and each node is labeled with a regular expression that describes the commonality of the segments indexed through the subtree. Each leaf node contains a collection of position points and each internal node holds links to a list of children nodes. The search of similar segments to a given pattern is then composed of a recursive traversal of relevant nodes in the BASS-tree. Starting from the root, the given pattern is compared to the label of every child. A child is believed to be *relevant* if one of its leaf descendants may contain a pointer to a similar segment. The same procedure is then invoked again on each relevant child until the relevant leaf nodes are reached. The set of position points in these relevant leaf nodes are then returned as (highly potential) candidates for further evaluation. We want to mention that, even though the general procedure for similarity search on BASS-tree is similar to that of the Fixed-queries tree (Baeza-Yates et al., 1994), the underlying data structures of these two trees do not resemble each other since the BASS-tree is designed exclusively for accelerating subsequence match. Many properties of the BASS-tree, including the balanced height and compactness, provide the theoretical foundation of the efficiency of the search as well as the maintenance. We shall show that applying the BASS-tree index on a protein database could achieve an order of magnitude speed-up over the BLAST algorithm.

[1] We will define later that a position point is the combination of a sequence ID and an offset relative to the beginning of the sequence. The notation of position point is introduced to uniquely identify a specific position in the protein sequence database.

We also notice that there are some recent advances in similarity searches. Floratos et al. (1999) proposed a methodology for homology detection based on only *significant patterns* discovered in the underlying database by TEIRESIAS (Rigoutsos and Floratos, 1998). Buhler and Tompa (2001) studied the challenging (l, d)-motif problem and introduced a motif discover algorithm based on the use of *random projections*. However, our objective in this paper is different: we want to design and implement a universal sequence index that benefits any generic subsequence similarity search. To the best of our knowledge, this paper is the first work on building persistent indexing structure for fast similarity search on protein sequence databases.

2 Measure of Similarity

The similarity between two segments, between a segment and a regular expression (that describes a set of segments), or between two regular expressions can be assessed in many ways (Durbin et al., 1998, Gusfield, 1997). In this paper, we adopt the *weighted edit distance* to measure the similarity and define the similarity between two regular expressions[2] ψ_1 and ψ_2 as the *maximum* similarity between any segment following ψ_1 and any segment following ψ_2. Let $\psi_1 = (s_1^1 + \ldots s_{m_1}^1)(s_1^2 + \ldots s_{m_2}^2)\ldots(s_1^j + \ldots s_{m_j}^j)$ and $\psi_2 = (t_1^1 + \ldots t_{n_1}^1)(t_1^2 + \ldots t_{n_2}^2)\ldots(t_1^k + \ldots t_{n_k}^k)$. The similarity $Sim(\psi_1, \psi_2) = F(|\psi_1|, |\psi_2|)$ and $F(j,k)$ can be recursively defined as

$$\max \begin{cases} F(j-1, k-1) + \max_{1 \le i \le m_j, 1 \le i' \le n_k} sm(s_i^j, t_{i'}^k), \\ F(j, k-1) + \max_{1 \le i' \le n_k} sm(-, t_{i'}^k), \\ F(j-1, k) + \max_{1 \le i \le m_j} sm(s_i^j, -) \end{cases} \quad (1)$$

where $sm(x, y)$ is the score in the substitution matrix (Gusfield, 1997) and "$-$" denotes a gap.

[2] Note that a segment can be regarded as a special case of a regular expression.

3 Model of SAM Index Method

In this section, first we give an overview of the structure of the BASS-tree and then discuss the SAM index method that is used to search protein database. (A more detailed description of BASS-tree is in Yang, 2002). Let $\Im$ be the alphabet. For example, the alphabet for protein sequences are the set of 20 amino acids. Let $S = \{S_1, S_2, \ldots, S_n\}$ be a set of sequences over the alphabet where $S_i = s^i_1 s^i_2 \ldots s^i_{l_i}$ and $s^i_j \in \Im$ for $1 \le j \le l_i$. l_i is referred to as the **length** of the sequence S_i. i is also referred to as the **ID** of the sequence S_i. Given a sequence $S_i = s^i_1 s^i_2 \ldots s^i_{l_i}$, a consecutive portion $s^i_{j_s} \ldots s^i_{j_e}$ $(1 \le j_s \le j_e \le l_i)$ is referred to as a **segment** whose **length** is $j_e - j_s + 1$. We define $S_i[j_s..j_e] = s^i_{j_s} \ldots s^i_{j_e}$ in the following discussion. Each individual symbol s^i_j in a sequence $s^i_1 s^i_2 \ldots s^i_{l_i}$ can be regarded as a segment of length 1. In this case, we use $S_i[j]$ as an abbreviation of $S_i[j, j]$. It is easy to see that each distinct position in a sequence serves as the starting position of some segment(s). Each of these positions can be uniquely identified by the combination of sequence ID and offset within the sequence. Each distinct combination (of sequence ID i and offset j) is also referred to as a **position point** $\langle i, j\rangle$. We sometimes say that a position point $\langle i, j\rangle$ **leads** a segment $t_1 \ldots t_k$ if the segment starts at the jth position in the sequence S_i (i.e., $s^i_j = t_1, \ldots, s^i_{j+k-1} = t_k$). The total number of position points is equal to the size of the protein sequence database (i.e., $\sum_{i=1}^n l_i$). Our index structure is built upon the set of all distinct position points in the database.

More specifically, a BASS-tree is a fully *balanced* tree that organizes all position points by recursively grouping together position points that lead *similar* segments in the protein sequence database. The structure of the BASS-tree is in spirit similar to that of the B+-tree (Elmasri and Navathe, 1994) in the sense that all leaf nodes are on the same level and each internal node has a bounded number of children. The fundamental difference between the BASS-tree and the B+-tree is that there exists a total ordering among all children of each node in the B+-tree, which enables both the search and the maintenance to be carried out straightforwardly. Unfortunately, the relationship among amino acids does not support a total ordering of children of an internal node in the BASS-tree, even though the similarity between any two nodes can be assessed (e.g., using some distance function (Durbin et al., 1998, Gusfield, 1997)). We shall address later in more detail that this feature poses a great challenge to the maintenance of the BASS-tree, especially when an overflowed node needs to be split. Figure 1(a) shows a portion of a BASS-tree using the substitution matrix BLOSUM 50 (Durbin et al., 1998). Each node in the BASS-tree is **labeled** with a regular expression[3] that describes the set of *similar* segments in the protein sequence database. Each leaf node contains a collection of position points that lead a set of similar segments described by the regular expression that serves as the **label** of the leaf node. A leaf node is also referred to as a **data node** in contrast to the internal node that holds links to descendant nodes in the BASS-tree. Each data node in Figure 1(a) is represented by a rectangle, which contains a list of entries, each of which stores a position point. As illustrated by a dashed arrow in the figure, the first entry of node X_6 stores the position point $\langle i, 6\rangle$. The **label** of each internal node describes a set of similar segments led by position points indexed through the subtree below. In Figure 1(a), the *label* of node X_5 is **(F+L)(I+V)** which indicates that the set of position points indexed in the subtree lead a set of segments (of length 2) that comply with the regular expression **(F+L)(I+V)**. For instance, the position point $\langle i, 1\rangle$ leads the segment **FI** that satisfies the label of node X_5. Each internal node, X, has a varying but bounded number of children, each of which is labeled with a *specialization* of the regular expression that labels X. Given two regular expressions ψ_1 and ψ_2, ψ_1 is said to be a **specialization** of ψ_2 iff, for every segment σ satisfying ψ_1, either σ also satisfies ψ_2 or a prefix of σ satisfies ψ_2. In this case, ψ_2 is called a **generalization** of ψ_1. For example, both **(A+S)** and **(D)(I+L)** are considered specializations of **(A+D+S+T)**.

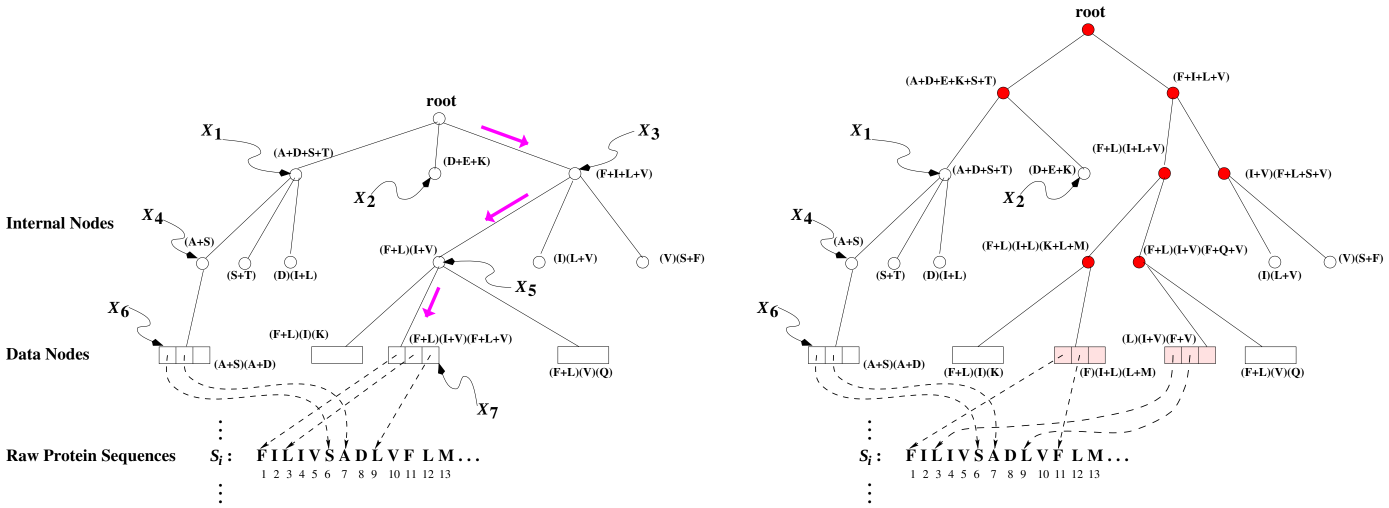
3.1 Data Structures

As shown in Figure 2(a), each internal node consists of a head field and a number of entries, each of which holds the information of a child node. Let M_I be the *maximum* number of children of an internal node. One way to determine the proper value of M_I is to set M_I to the maximum value so that an internal node can still be fit into a storage block (e.g., 8KB), for the sake of experimental efficiency. The head field contains a counter to track the actual number of children. The essential information kept in each entry of the main body includes label of a child and the link to the child node. Figure 2(b) shows the information stored at node X_5 in Figure 1(a). Each data node consists of a head field and a list of data entries. The head field usually maintains the number of entries currently stored in the node. The maximum number of entries that can be stored at each data node is bounded by a parameter M_D whose value can be determined in a similar manner as that of M_I. Each data entry holds a position point. An example of the data node (e.g., node X_7 in Figure 1(a)) is also shown in Figure 2(b).

3.2 Building the BASS-Tree

As we mentioned before, one important feature of the BASS-tree is that position points leading similar segments

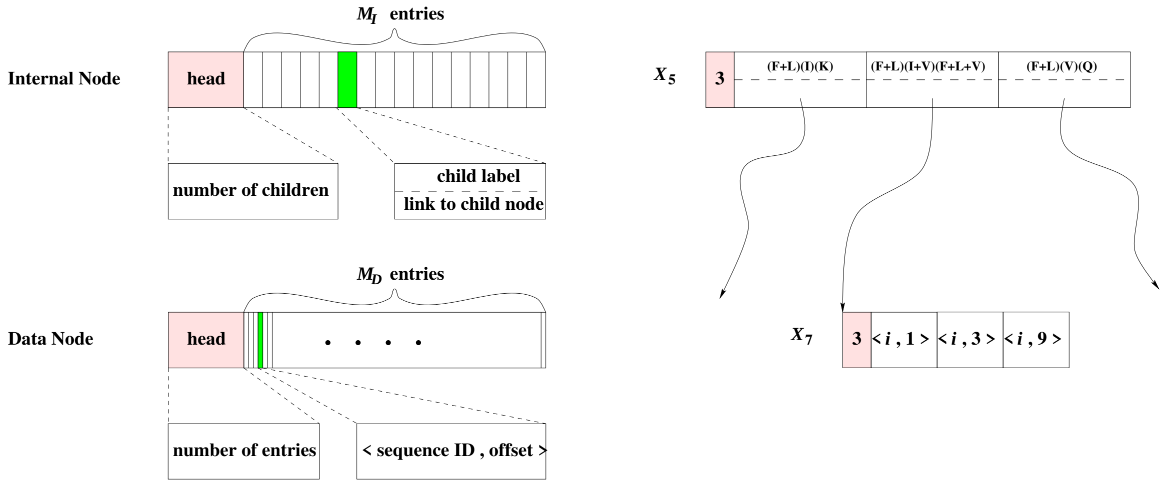
[3] For example, **(A+S+T)(A+S)** is a regular expression of length two, which represents the set of segments whose first position may be either **A**, **S** or **T**, and the second position may take either **A** or **S**.



(a) Part of a BASS-tree after Inserting Position Points <i, 1> ... <i, 10>

(b) Part of a BASS-tree after Inserting the Position Point <i, 11>

Figure 1. An Example of BASS-Tree



(a) Data Structures of the BASS-Tree

(b) Examples of Nodes X5 and X7 in Figure 1

Figure 2. Data Structures of BASS-Tree

will be put close to each other in the BASS-tree so that only a small portion of the BASS-tree needs to be visited and hence a small subset sequences need to be examined during an approximate match query, no matter what the query sequence is. The general procedure to build a BASS-tree on a protein sequence database is to insert each distinct position point into an initially empty tree. At the beginning, the BASS-tree only consists of a root node whose only child is an empty data node. An insertion procedure is invoked to insert each position point into an appropriate data node (determined by examining the segment led by this position point) and to maintain the relevant internal nodes accordingly.

3.2.1 Insertion of a Position Point

Given a BASS-tree, the procedure of inserting a position point $\langle i, j \rangle$ consists of two phases: (1) locating the proper data node and (2) updating the data node and its ascendant internal node(s) if applicable. The routine of locating the desired data node can be done by, beginning from the root, recursively picking the child whose label is most similar to any segment starting from the j position in S_i. If there is a tie among several children, one of them is picked randomly and the procedure continues. For example, in order to locate an appropriate data node for the position point $\langle i, 11 \rangle$ in Figure 1(a), we first compute the similarity between the segment

FLM... and every child of the root. It is easy to see that node X_3 is the most similar one. The same procedure is then invoked again to examine X_3's children, and so on, until the data node (e.g., X_7) is reached. (The path is indicated bold arrows in Figure 1(a).) Once the data node has been located, the process enters the second phase where two scenarios need to be considered.

- *The data node still has empty entry.* This scenario is relatively easy to handle since it does not involve structural change of the BASS-tree. The position point is stored in an empty entry and the head field of the data node is updated to reflect the inclusion of the new data entry. In addition, the node label (physically stored at the parent node) is updated accordingly and this update also propagates up towards the root if necessary.

- *The data node is full.* This is the scenario when we try to insert the position point $\langle i, 11\rangle$ in X_7 in Figure 1(a), assuming that each data node can hold at most 3 entries (i.e., $M_D = 3$). In this case, additional node(s) need to be created in the BASS-tree. A partition procedure is invoked to split this node into two data nodes. All position points in this node will be divided into two portions of roughly equal size, one portion will remain in this data node while the other portion will be stored in a newly created data node. The new position point will then be inserted into the data node whose label is more similar to the segment led by the position point. A new child entry (corresponding to the newly created data node) should be inserted in the parent node of the (original) data node. If the parent node **overflows** as a result of the insertion of a new child entry, a splitting procedure is invoked to split this node into two internal nodes containing roughly same number of child entries. A new child entry is then added in the parent node of the internal node. This split may propagate all the way up to create a new root node and hence a new level of the BASS-tree. In the above example, after the data node X_7 is split into two nodes, the additional child entry causes the internal node X_5 to overflow, assuming that each internal node can have at most 3 children (i.e., $M_i = 3$). This triggers the split of node X_5 into two internal nodes, which happens to cause the overflow of its ancestor node(s). In this example, this chain reaction results in cascading split of all internal nodes along the path up to the root and leads to the growth of the tree height (from 3 to 4). The resulting BASS-tree is shown in Figure 1(b). The shaded nodes (including both data nodes and internal nodes) are those nodes affected in this process.

We shall mention that, in most cases, inserting a position point would not cause any structural change (in terms of node split) and hence can be performed very efficiently. Specifically, the chance that inserting a position point would trigger a node split is $\frac{1}{M_D}$ where M_D is the capacity of the data node and is typically in the range of hundreds to thousands. In addition, the probability that such a data node split may cause an internal node to be split is $\frac{1}{M_I}$ where M_I is the capacity of an internal node and may range up to dozens. The chance that the split propagates to the root and increases the height of the BASS-tree is very rare. As a matter of fact, the height of the BASS-tree grows at a very slow pace ($O(\log(\sum_{i=1}^{n} l_i))$). In practice, a BASS-tree of height 4 or 5 is sufficient to index a reasonably large protein database (e.g., SWISS-PROT).

3.2.2 Node Split

The node split is a very crucial procedure to ensure the quality of the resulting BASS-tree, even though it is invoked only when a node is full. There certainly exist many choices on how to split a node (into two portions). A poor split may result in loss of vital information that plays a deterministic role in screening out unnecessary candidates during an approximate match and significantly prolong the response time. Consider the extreme case (which is also the worst case) where a random partition is employed every time a node is split. Then each position point would have the same probability to be in any data node in the BASS-tree, no matter which segment it leads. During an approximate search, the position points that lead similar segments to the query segment may be scattered instead of concentrating in only a couple of branches. This would force us to examine the entire database.

The optimal partition should be the one that maximize the similarities within each portion resulted from the partition but minimize the similarities across portions. In the real implementation, the partition is performed on the set of entries in the node to be split and the new entry to ensure the optimality of the split. Ideally, we would like that position points leading similar segments should be stored together in the BASS-tree so that only a small number of data nodes need to be retrieved eventually. In addition, the position points should be organized in such a way that the label of each node is as specialized (i.e., containing less variant) and distinct (i.e., having less similarity to others) as possible. In general, the more distinct the label of each node, the more likely, on average, a node (and its subtree) may be eliminated from further investigation at a very early stage during an approximate search, and hence the faster the overall response time.

The problem of generating the optimal partition is in spirit similar to the **min-cut** problem (Cormen et al., 1990, Motwani and Raghavan, 1995). Each entry in the node to be partitioned as well as the new entry can be mapped to a vertex, and there is an edge between two vertices if the corresponding entries have positive similarity (which serves as the weight on the edge). The objective is to partition the graph into two portions, each consists of roughly half of the vertices, and the aggregated weight on the edges across portions is minimized. This can be solved in $O(M^2 \log M)$ time via a randomized algorithm where M is the number of entries considered (Motwani and Raghavan, 1995) (e.g., $M_I + 1$ for internal node and $M_D + 1$ for data node). Due to space limitations, interested readers please refer to Motwani and Raghavan (1995) for the detail algorithm to compute the min_cut.

3.3 Approximate Match with SAM

Once the BASS-tree has been constructed on a protein sequence database, the cost of processing an approximate match of any given segment can be dramatically reduced. One type of commonly asked query is, given a segment $\sigma = t_1 \ldots t_k$ and a similarity threshold min_sim, find the set of segments whose similarities to σ are at least min_sim. A depth-first traversal of the BASS-tree is able to locate all segments that satisfy the criterion. Starting from the root, a recursive procedure is invoked to examine the label of every child and determine the set of children that may hold index to at least one segment potentially possessing sufficient similarity to the query segment. This can be achieved by evaluating the similarity of the query segment to the label of each child node. If it is less than min_sim, we can exclude the node (and its subtree if applicable) from further examination. Otherwise, the node is believed to be relevant to the query and the same procedure described above is invoked on the node to examine its children, if it is an internal node. If it is a data node, a final verification procedure is carried out to check every position point to determine whether the corresponding segment is indeed sufficiently similar to the query segment.

Three scenarios need to be addressed when Algorithm A.5 is employed to search on segment of arbitrary length.

1. The query segment is shorter than the length of the label the data nodes in the BASS-tree. If this is the case, then the set of *internal* nodes whose labels are similar to the query segment will be identified (in the way described above) and entries in their descending data nodes will be retrieved to the $Result$ directly. This step can be performed very efficiently by maintaining, for each internal node, an auxiliary link that points to the physical storage address of the list of data nodes in the subtree. The implementation details can be found in our full paper.

2. The query segment is longer than the length of the label the data nodes in the BASS-tree. The following strategy can be played to take full advantage of the available BASS-tree. Let σ be the query sequence. A set of short segments in σ can be taken to pose similarity queries on the BASS-tree. The results from these queries can then be *merged* to serve as the candidates for the final result. The rationale is that, if a sequence contains similar portion(s) to the query sequence, then it must contain similar portion(s) to a segment in the query sequence.

3. The query sequence contains "wild cards". Then, the query sequence is first decomposed into a list of segments of consecutive symbols. Each segment can be taken to pose queries on the BASS-tree and the results can be merged together.

The (merged) result produced by searching on the BASS-tree is indeed a *superset* of the true result of the similarity query and a final validation process will be carried out to eliminate all false positives and to find the best alignments with the query sequence if required. Each entry in the result is examined against the query sequence to determine the similarity and to find the best alignment. This can be achieved by a simple dynamic programming routine (Gusfield, 1997). Since the number of entries passed to the validation process is usually very small, the computational cost incurred by the validation process is insignificant.

3.4 Implementation Issues

The BASS-tree only needs to be built once on the protein database and can be used to facilitate the process of all subsequent approximate match queries. Structural wise, the BASS-tree is a compact and fully balanced tree structure in the sense that each node is guaranteed to be at least half full and all leaf nodes are on the same level. Consequently, there are at most $\lceil \frac{N}{M_D/2} \rceil$ data nodes and at most $\lceil \frac{\lceil \frac{N}{M_D/2} \rceil - 1}{M_I - 1} \rceil$ internal nodes where $N = \sum_{i=1}^{n} l_i$ is the size of the protein sequence database and M_D and M_I are the capacity of a data node and the maximum fan-out of an internal node, respectively. The size of the BASS-tree is $O(\lceil \frac{N}{M_D/2} \rceil \times M_D + \lceil \frac{\lceil \frac{N}{M_D/2} \rceil - 1}{M_I - 1} \rceil \times M_I) = O(N)$ and the height of the tree is $O(\log_{M_I} \frac{N}{M_D}) = O(\log N)$ since M_I and M_D are usually fixed and can be regarded as constants. This implies that any data node can be reached in at most $O(\log N)$ steps and guarantees the stability of the performance of sequence matching and tree maintenance.

- During an approximate match query, only nodes that have sufficient similarities to the query segment are examined and only the set of sequences that may contain similar segment to the queried one are visited. Let x be the number of data nodes whose labels are similar to the queried segment. Then the total number of nodes visited during the query is bounded by $O(x \log N)$. In most cases, it is much smaller than both the size of the BASS-tree and the size of the protein database, which is crucial to the speed-up of the process. We shall mention that, in practice, the overall response time of an approximate match query highly depends on the number of nodes visited during the procedure. This is mainly because the majority of the BASS-tree (e.g., data nodes and some low level internal nodes) may be stored on the secondary storage space, e.g. disk. Each node occupies a disk block and can be retrieved to main memory at the cost of one disk I/O. Typically, the time consumed by disk I/Os makes up the majority of the processing time. Minimizing the number of nodes visited during a query will minimize the number of disk I/O performed and hence accelerate the response time.

- The insertion of each position point consumes $O(\log N)$ time, no matter whether a node split occurs. Once the BASS-tree has been built, any change to the underlining protein database can be easily incorporated into the tree in an incremental manner. (1) New sequences can be added by inserting each new position point successively into the existing BASS-tree. (2) Deletion of obsolete sequences can be achieved by deleting the set of corresponding position points from the BASS-tree, which can be done via a similar but opposite procedure as the insertion and also takes $O(\log N)$ time for each position point. (3) An update made to existing sequences can be handled as a deletion (of the original sequence) immediately followed by an insertion (of the updated sequence) for each affected position point.

4 Results

In this section, we analyze the performance of the SAM method against two alternatives. The first one is the suffix tree. To best of our knowledge the index structure proposed in Hunt et al. (2001) (a variation of the suffix tree) is the most efficient implementation of the suffix tree. However, this indexing structure is intended for the exact match, which does not consider the amino acid mutation. As a result, to find all portions of sequences in the database that may be a potential mutation of the given pattern, all possible mutations of the pattern have to be checked. The second alternative is BLAST (Altschul et al., 1990). Note that, as a heuristics approach, the BLAST "approximates" the best similarities (or alignments), whereas the SAM method always reports the highest similarities and the optimal alignments. From our experiments, we found that the SAM method is able to produce results of comparable quality to that of the BLAST since their results almost coincide with each other in all tests that we performed.

We evaluate the computational advantages of the SAM method on an IBM-AIX computer with a 333MHz CPU and 128 MB main memory. The SWISS-PROT protein database is used to test the SAM method. The data set consists of 100,304 protein sequences and 36,743,059 amino acids. For the test data set, the BASS-tree index structure occupies 1.8 MB space, 5% of the data size. It takes approximately 2.5 hours to build the index. On the other hand, the suffix tree index (Hunt et al., 2001) takes 8 MB space, which is four times larger than the BASS-tree. As a result, when the data size grows larger, it is more difficult to load the entire suffix tree data structure into the main memory, and leads to longer response time. The time to build the suffix time is over four hours, which is significantly longer than building the BASS-tree.

The most important metric to determine how good an index structure is the response time of answering a query, i.e., finding matches of a given pattern. We query the data set with various patterns. Figure 3(a) shows the query time as a function of the pattern length (number of amino acids in the pattern). The query time of the BASS-tree is less than 20% of that of the BLAST algorithm and less than 10% of that of suffix tree when the pattern is relatively short. When the pattern length increases, the query time (for all three methods) also increases. The query time of the BLAST and the SAM method increases linearly with respect to the pattern length because both scheme uses shorter seeds first to find matches. On the other hand, the query time of the suffix tree increases much faster with respect to the pattern length because it has to look for all possible mutations of the pattern.

Figure 3(b) shows the query time as a function of the number of matches. In this figure, we use patterns with length between 5 and 49 with the mean of 15. When the number of matches increases, the query time of all three schemes also increases. The SAM method maintains a six fold speed up over BLAST and a 15 times speed up over the suffix tree approach.

5 Discussions

There are several interesting issues that are worth further investigation. (1) There exist many scoring matrices. Instead of building a separate BASS-tree for each score matrix, a preprocessing of the set of score matrices can be conducted to partition them into several similar groups. Matrices in a group would be similar to each other. A meta scoring matrix can be constructed for each group by taking the maximum value of each entry. A meta BASS-tree is then constructed on the meta matrix. When a user wants to search a pattern according to a specific scoring matrix, we can use the meta BASS-tree built on the corresponding meta matrix. (2) The potential of additional benefit obtained on processing "all versus all" comparison of a set of sequences using the SAM method may be very significant. As a batch of queries are processed simultaneously, additional performance improvement may be brought by sharing some common operations on traversing the BASS-tree. (3) Due to the advanced biological techniques, rich information on many different aspects of protein sequences have been available today. An interest-

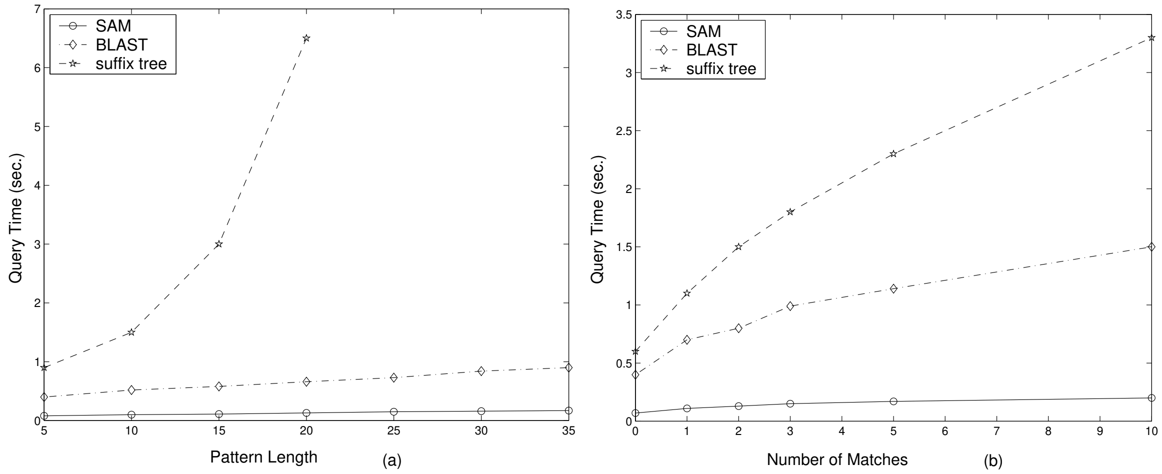


Figure 3. Query Time

ing future research direction is to extend the SAM method to handle more sophisticated data model, e.g., the 3D secondary structure, which is currently under consideration. (4) Another potential application of the SAM method is to index the *gene expression profile* (Billups et al., 2001). A gene expression profile of a drug consists of the expression levels of various genes at a variety of different concentrations of the drug. For each drug, a *permutation vector* is created recording the order in which the genes are turned on by the drug as drug concentrations increase. The similarity between two drugs can be assessed by comparing the gene orders in the corresponding permutation vectors. To facilitate the computation, the set of all permutation vectors can be organized in an orderly manner by a BASS-tree so that, given a new profile, similar profiles can then be retrieved efficiently.

References

Altschul, S., Gish, W., Miller, W., Myers, W., and Lipman, J. (1990) Basic local alignment search tool. *Journal of Molecular Biology*, 215, pp. 403-410.

Altschul, S. and Gish, W. (1996) Local alignment statistics. *Methods in Enzymology*, 266, pp.460-480.

Altschul, S., Madden, T., Schaffer, A., Zhang, J., Zhang, Z., Miller, W., and Lipman, J. (1997) Gaped BLAST and PSI-BLAST: a new generation of protein database search programs. *Nucleic Acids Research*, 25, pp. 3389-3402.

Andersson, A. and Nilsson, S. (1995) Efficient implementation of suffix trees. *Softw. Pract. and Exp.*, 25(2), pp. 129-141.

Baeza-Yates, R., Cunto, W., Manber, U., and Wu, S. (1994) Proximity matching using fixed-queries trees. *Proc. 5th Combinatorial Pattern Matching*, pp. 198-212.

Billups, C., Bar-Or, R., Burnham, J., Gannon, J., Garcia, Y., Kaur, S., Lana, L., and Miller, C. (2001) Final report of the CU-Denver mathematics clinic: Lead compound optimization using gene expression profiling. *Mathematics Clinic Report MC01S001*, University of Colorado at Denver, Department of Mathematics.

Buhler, J. and Tompa, M. (2001) Finding motifs using random projections. *Proc. ACM RECOMB*, pp. 69-76.

Burkhardt, S., Crauser, A., Ferragina, P., Lenhof, H., Rivals, E., and Vingron, M. (1999) *q*-gram based database searching using a suffix array (QUASAR). *Proc. of ACM Recomb*, pp. 77-83.

Cormen, T., Leiserson, C., and Rivest, R. (1990) *Introduction to Algorithms*, The MIT Press.

Durbin, R., Eddy, S., Krogh, A., and Mitchison, G. (1998) *Bioloical Sequence Analysis*, Cambridge University Press.

Elmasri, R. and Navathe, S. (1994) *Fundamentals of Database Systems*, The Benjamin/Cummings Publishing Company.

Farach, M., Ferragina, P., and Muthukrishnan, S. (1998) Overcoming the memory bottleneck in suffix tree construction. *Proc. FOCS*, pp. 174-185.

Ferragina, P. and Grossi, R. (1999) The string B-tree: a new data structure for string search in external memory and its applications. *J. ACM*, 46(2), pp. 236-280.

Floratos, A., Rigoutsos, I., Parida, L., Stolovitzky, G., and Gao, Y. (1999) Sequence homology detection through large scale pattern discovery. *Proc. ACM RECOMB*, pp. 164-173.

Gusfield, D. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*, Cambridge University Press.

Hunt, E., Atkinson, M., and Irving, R. (2001) A database index to large biological sequences. *Proc. of VLDB*, pp. 139-148.

McCreight, E. (1976) A space-economic suffix tree construction algorithm. *J. ACM*, 23(2), pp. 262-272.

Meyers, E. (1994a) Algorithmic advances for searching biosequence databases. *Computational Methods in Genome Research*, pp. 121-135.

Meyers, E. (1994b) A sublinear algorithm for approximate keyword searching. *Algorithmica*, 12, pp. 345-374.

Motwani, R. and Raghavan, P. (1995) *Randomized Algorithms*, Cambridge University Press.

Navarro, G., Sutinen, E., Tanninen, J., and Tarhio, J. (2000) Indexing text with approximate q-grams. *CPM2000*, LNCS 1848, pp. 350-365.

Navarro, G. and Baeza-Yates, R. (2000) A hybrid indexing method for approximate string matching. *J. of Discrete Algorithms*, 1(1), pp. 21-49.

Navarro, G. and Baeza-Yates, R. (2001) Improving an algorithm for approximate pattern matching. *Algorithmica*, 30(4), pp. 473-502.

Navarro, G. and Raffinot, M. (2001) Fast and simple character classes and bounded gaps pattern matching, with application to protein searching. *ACM RECOMB*, 231-240.

Rigoutsos, I. and Floratos, A. (1998) Combinatorial pattern discovery in biological sequences: the TEIRESIAS algorithm. *Bioinformatics*, 14(1), pp. 55-67.

Ukkonen, E. (1992) Approximate string matching with q-grams and maximal matches. *Theor. Comput. Sci.*, 92(1), pp. 191-202.

Ukkonen, E. (1995) On-line construction of suffix trees. *Algorithmica*, 14(3), pp. 249-260.

Yang, J., Wang, W., Xia, Y., and Yu, P. (2002) Index for approximate search on large string databases. *IBM Research Report*.

A Algorithms

Algorithm A.1 *BASS-tree Construction*

BuildBASSTree(S, M_D, M_I, sm)
{ /* $S = \{S_1, \ldots, S_n\}$ is a database of sequences;
M_D is the maximum number of position points that can be held in a data node and is usually an odd integer;
M_I is the maximum number of children that an internal node may have and is usually an odd integer; sm is a substitution matrix; */
$BASS \leftarrow$ NewInternalNode(M_I) /* Initialize the root */
$BASS.ChildrenNum \leftarrow 1$
/* The root has one child which is an empty data node */
$BASS.Child[1] \leftarrow$ NewDataNode(M_D)
for $i \leftarrow 1$ **to** n **do**
    for $j \leftarrow 1$ **to** l_i **do** /* l_i is the length of S_i */
        Insert($BASS$, i, j, M_D, M_I, sm)
        /* Insert the position point $\langle i, j \rangle$ to $BASS$ */
return $BASS$
/* $BASS$ is the BASS-tree built on the protein sequence database S */
}

Algorithm A.2 *Insert a Position Point in a BASS-tree*

Insert($BASS, i, j, M_D, M_I, sm$)
{ /* Insert the position point $\langle i, j \rangle$ to the BASS-tree $BASS$ */
/* Line 1 to 11: Locate the appropriate data node to insert $\langle i, j \rangle$. This is achieved by recursively traversing down the tree to the data node whose label is most similar to the segment lead by $\langle i, j \rangle$ */
1: $T \leftarrow BASS$
2: **while** T is an internal node **do**
3:     $MaxSim \leftarrow -\infty$
4:     $SimChild \leftarrow 0$
5:     **for** $k \leftarrow 1$ **to** $T.ChildrenNum$ **do**
6:         $Sim \leftarrow$ Similarity($T.Child[k].Label, i, j, sm$)
        /* The similarity can be calculated according to Equation 1 */
7:         **if** $MaxSim < Sim$
8:         **then**
9:             $MaxSim \leftarrow Sim$
10:             $SimChild \leftarrow T.Child[k].Link$
11:     $T \leftarrow SimChild$
/* Line 12 to 25: Insert $\langle i, j \rangle$ in T, which may trigger node split */
12: **if** $T.DataNum < M_D$

13: **then** /* T is not full */
14: $T.DataNum \leftarrow T.DataNum + 1$
15: $T.Data[T.DataNum] \leftarrow \langle i, j \rangle$
16: **else** /* T is full */
17: $NewT \leftarrow$ SplitDataNode(T, M_D, sm, i, j)
18: $T \leftarrow T.Parent$
19: **while** $T.ChildrenNum = M_I$ **do**
20: $NewT \leftarrow$ SplitInternalNode(T, M_I, sm, $NewT$)
21: $T \leftarrow T.Parent$
22: $T.ChildrenNum \leftarrow T.ChildrenNum + 1$
23: $T.Child[T.ChildrenNum].Link \leftarrow NewT$
24: $T.Child[T.ChildrenNum].Label$ $\leftarrow$ GenerateLabel($NewT$)
25: UpdateLabel(T)
/* Update the label of T as well as its ancestors in the BASS-tree */
26: **return**
}

Algorithm A.3 *Split a Data Node*

SplitDataNode(T, M_D, sm, i, j)
{ /* Split the data node T into two data nodes of equal size */
/* Line 1 to 7: Map to a graph */
1: **for** $k \leftarrow 1$ **to** M_D **do**
2: $Vertex[k] \leftarrow T.Data[k]$
3: $Vertex[M_D + 1] \leftarrow \langle i, j \rangle$
4: **for** $k \leftarrow 1$ **to** M_D **do**
5: **for** $k' \leftarrow 1$ **to** k **do**
6: $Edge[k][k']$ $\leftarrow$ Similarity($Vertex[k]$, $Vertex[k']$, sm)
/* The similarity can be calculated according to Equation 1 */
7: $Edge[k'][k] \leftarrow Edge[k][k']$
/* Line 8: Calculate the min_cut on the graph. */
8: $\{P_1, P_2\} \leftarrow$ Min_Cut($Vertex$, $Edge$)
/* Line 9 to 15: Store the split data nodes in the BASS-tree properly */
9: **for** $k \leftarrow 1$ **to** $\lceil \frac{M_D+1}{2} \rceil$ **do**
10: $T.Data[k] \leftarrow P_1[k]$
11: $T.DataNum \leftarrow \lceil \frac{M_D+1}{2} \rceil$
12: $NewT \leftarrow$ NewDataNode(M_D)
13: **for** $k \leftarrow 1$ **to** $\lceil \frac{M_D+1}{2} \rceil$ **do**
14: $NewT.Data[k] \leftarrow P_2[k]$
15: $NewT.DataNum \leftarrow \lceil \frac{M_D+1}{2} \rceil$
16: **return** $NewT$
}

Algorithm A.4 *Split an Internal Node*

SplitInternalNode(T, M_I, sm, $NewT$)
{ /* Split the internal node T into two internal nodes of equal size */
/* Line 1 to 8: Map to a graph */
1: **for** $k \leftarrow 1$ **to** M_I **do**
2: $Vertex[k] \leftarrow T.Child[k]$
3: $Vertex[M_I + 1].Link \leftarrow NewT$
4: $Vertex[M_I + 1].Label \leftarrow$ GenerateLabel($NewT$)
5: **for** $k \leftarrow 1$ **to** M_I **do**
6: **for** $k' \leftarrow 1$ **to** k **do**
7: $Edge[k][k'] \leftarrow$ Similarity($Vertex[k].Label$, $Vertex[k'].Label$, sm)
8: $Edge[k'][k] \leftarrow Edge[k][k']$
/* Line 9: Calculate the min_cut on the graph. */
9: $\{P_1, P_2\} \leftarrow$ Min_Cut($Vertex$, $Edge$)
/* Line 10 to 16: Store the split internal nodes in the BASS-tree properly */
10: **for** $k \leftarrow 1$ **to** $\lceil \frac{M_I+1}{2} \rceil$ **do**
11: $T.Child[k] \leftarrow P_1[k]$
12: $T.ChildNum \leftarrow \lceil \frac{M_I+1}{2} \rceil$
13: $NewT \leftarrow$ NewInternalNode(M_I)
14: **for** $k \leftarrow 1$ **to** $\lceil \frac{M_I+1}{2} \rceil$ **do**
15: $NewT.Child[k] \leftarrow P_2[k]$
16: $NewT.ChildNum \leftarrow \lceil \frac{M_I+1}{2} \rceil$
17: **return** $NewT$
}

Algorithm A.5 *Approximate Match*

ApproximateMatch(σ, T, min_sim, sm)
{ /* Search the set of position points that lead segments whose similarity to σ is at lease min_sim */
1: $Result \leftarrow \emptyset$
2: **if** T is a data node
3: **then**
4: **for** $k \leftarrow 1$ **to** $T.DataNum$ **do**
5: $Result \leftarrow Result \cup \{T.Data[k]\}$
6: **return** $Result$
7: **else** /* T is an internal node */
8: **for** $k \leftarrow 1$ **to** $T.ChildrenNum$ **do**
9: **if** Similarity($\sigma, T.Child[k].Label, sm$) $\geq min\_sim$
10: **then** $Result \leftarrow Result \cup$ ApproximateMatch ($\sigma, T.Child[k].Link, min\_sim, sm$)
11: **return** $Result$
/* $Result$ holds all position points that lead similar segments to σ */
}