Fast Filter-and-Refine Algorithms for Subsequence Selection

Beng Chin Ooi[1] Hwee Hwa Pang[2] Hao Wang[1] Limsoon Wong[2] Cui Yu [1]

[1]Department Computer Science
National University of Singapore
3 Science Dr 2, Singapore 117543
{ooibc,wanghao,yucui}@comp.nus.edu.sg

[2]Lab for Information Technology
21, Heng Mui Keng Terrace
Singapore 119613
{hhpang,limsoon}@lit.org.sg

Abstract

Large sequence databases, such as protein, DNA and gene sequences in biology, are becoming increasingly common. An important operation on a sequence database is approximate subsequence matching, where all subsequences that are within some distance from a given query string are retrieved. This paper proposes a filter-and-refine algorithm that enables efficient approximate subsequence matching in large DNA sequence databases. It employs a bitmap indexing structure to condense and encode each data sequence into a shorter index sequence. During query processing, the bitmap index is used to filter out most of the irrelevant subsequences, and false positives are removed in the final refinement step. Analytical and experimental studies show that the proposed strategy is capable of reducing response time substantially while incurring only a small space overhead.

1. Introduction

Large sequence databases, such as protein, DNA and gene sequences in biology, are becoming increasingly common. In such applications, users need to elicit from a sequence database those subsequences that match certain query strings. For instance, a biologist may want to retrieve all known gene sequences that contain certain segments of nucleotides.

While subsequence matching capability may be programmed into individual applications, it would be desirable for the DBMS to offer this capability so as to improve programmers' productivity and system efficiency.

While modern database systems provide facilities to store data sequences, there is a paucity of support for sequence manipulation. For example, many systems have a *text* data type for variable-length strings and support some form of keyword indexing such as signature files. In conventional database systems, there are substring matching operators such as LIKE. Alternatively, data sequences can be stored in external files where system utilities like $grep$ and $fgrep$ can operate on them. However, these approaches allow only exact keyword or substring matching, but not the sophisticated approximate subsequence matching required by the applications described above.

Approximate subsequence matching can be defined as an operation that takes as input an edit distance [23] $EditDist$ and a query string $*Q_1 * Q_2 * .. * Q_m*$, where $*$ is a variable length don't care (VLDC) segment, and each Q_i is a segment of at least one data element and possibly some fixed length don't care characters. As output, the operation returns all subsequences $*D_1 * D_2 * .. * D_m*$ in the database that each can be transformed into the query string by replacing at most $EditDist$ characters, after an optimal substitution for the VLDCs. With this definition, substring matching can be viewed as a special case of subsequence matching where m = 1 [17].

In this paper, we propose a two-step filter-and-refine query processing strategy for subsequence matching, in the context of DNA sequences in particular. We introduce *BIS*, a bitmap indexing scheme for adding approximate subsequence matching capability to database systems. The most attractive feature of the scheme is that it can speed up queries very significantly while incurring only the space overhead of a small fraction of the data size. The data structure, *BIS*, uses a hash function to encode and reduce each data sequence in a collection to a shorter index sequence. At runtime, the index sequences are used to efficiently filter out most of the subsequences that do not match a submitted query string. A very desirable characteristic of *BIS* is that the false drop rate (the number of subsequences that filter through even though they do not match the query) declines exponentially as the query length increases. Two notable objectives of *BIS* are (1) reducing the space complexity of the index, and (2) improving the performance of index processing. For the purpose of performance tuning and evaluation, a cost model of the scheme that estimates the achievable response time savings is also presented. We have implemented the processing strategy as part of a proteomic application system, and also as a stand alone system to study its efficacy.

The remainder of the paper is organized as follows. Section 2 describes a motivating example. Section 3 introduces the basic algorithm and Section 4 presents its extended form. Section 5 shows the derivation of a cost model for *BIS*, which is verified through the experiments reported in Section 6. The section also analyzes the behavior of *BIS* and studies its efficiency. A discussion of related work appears in the Section 7. Finally, Section 8 concludes the paper.

2. A Motivating Genetic Sequence Application

In this section, we shall describe an application in proteomics. The primary goal in proteomics is to discover what are the functions of proteins. An important idea in this process is the use of protein "motif". A protein motif is in essence a signature for an associated protein function. If a particular motif is detected in a protein, then the protein has some likelihood of possessing the function associated with that motif. In order to properly exploit protein motifs, three items are necessary: (a) a database of known protein motifs, (b) a tool to automatically discover protein motifs, and (c) a tool to scan protein databases for motifs obtained from (a) and (b). One of the most common type of queries asked by molecular biologists has the form "Which sequences in a protein database has the function (i.e., motif) that I am interested in?", and needs to be answered in real time.

To further motivate our work, consider the polyprotein of the denguevirus [14] which is known to have helicase activity [8]. However, the Swiss Prositescan [2, 3], the most widely used software for motif checking, reports no helicase activity for this protein in question. There are only two possible explanations. The first possibility is that the PROSITE motif collection used by Prositescan does not contain helicase motifs. The other possibility is that PROSITE does contain helicase motifs, but they are derived from organisms that are too distantly related to denguevirus.

Actually, PROSITE contains the helicase motif `[GSAH].[LIVMF]{3}DE[ALIV]H[NECR]`. This motif is to be read as follows: The first residue is any one of G, S, A, H; the second residue is a "don't care", i.e., a FLDC of length 1; the next three residues are any of L, I, V, M, F; the next residue must be D; the one after that must be E; the next residue is any of A, L, I, V; this is then followed by a H, and the final residue is any of N, E, C, R. The actual helicase site in denguevirus polyproteins is `NLIIMDEAHF`, which disagrees with that of PROSITE at the two flanking residues. This explains the failure of Prositescan to detect the helicase site in denguevirus polyproteins.

The proposed algorithm underlying a PROSITE application must support matching modulo Fixed Length Don't Care (FLDC), character classes, and edit distance, and must be able to rapidly report that denguevirus polyproteins have helicase sites with 2 mutations on the flanking residues of the PROSITE helicase motif.

Besides the PROSITE application, we have also developed several other proteomics applications, including: 1) $ScanSeq$ that searches segments of sequences satisfying a given motif; 2) $Signature$ that discovers conserved motifs in amino acid sequences; 3) $Duplicate$ that looks for repeats in amino acid sequences; 4) $DNADuplicate$ that finds tandem and inverted repeats in DNA sequences.

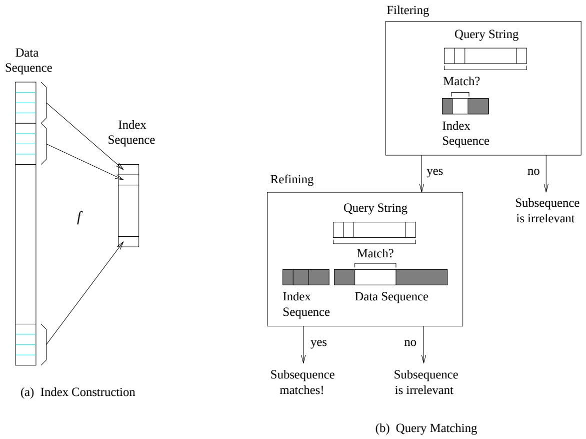
These applications are currently being used by biologists in their work. Taking advantage of the filter-and-refine utility that is introduced in this paper, the developers were able to quickly complete the applications to the users' requirements.

3. The Basic Algorithm

When the database is populated, each sequence is encoded into a smaller-size bitmap index. During query processing, this bitmap index is used at the filtering step to efficiently eliminate most of the subsequences that do not match a given query string. At the refinement step, each matching sequence is further checked using real data sequence to remove false positives. Figure 1 illustrates the process. The idea behind *BIS* is similar to that of signature file schemes, where compact signatures are used to filter out data objects that are irrelevant to a query. Unlike work in that area which focused on sophisticated signature extraction and storage structures for the signature files [28], however, *BIS* has a simple and efficient index generation and storage mechanism. Rather, our contribution is a scheme for using a single, compact bitmap index to quickly find subsequences that match arbitrary-length patterns in an adhoc querying environment.

In constructing the index, a data element is mapped via a hash function into an index element that is smaller in size. In practice, a data element could be a character, a short integer, a long integer, a float, or a double float field, while an index element could range from one bit to a few bytes. Figure 1 shows a hash function f that maps a multi-bit data element into a single index bit. By applying the same hash function to every data element in a sequence and to each sequence in the database, we produce a smaller bitmap index that can be retrieved and matched quickly during query retrieval.

When a query string is presented, the filtering step uses *BIS* to evaluate the index representation of each sequence in turn to isolate its subsequences that might satisfy the query string. Only the fraction of subsequences that "drops" through the index filter will have their data representation



(a) Index Construction

(b) Query Matching

Figure 1. Filtering and Refining

Notation	Meaning	Default
N	Index element	
N_{bits}	Number of bits in an index element	1
D	Data element	
D_{bits}	Number of bits in a data element	8
$NumSeq$	Number of data sequences	1
$SeqLen$	Avg. number of elements per data sequence	512 million
$QLen$	Number of query elements	16

Table 1. Algorithm Parameters

checked for an actual match at the refinement step; the remaining subsequences can be disqualified immediately. By selecting the hash function judiciously, most of the irrelevant subsequences can be eliminated through the filtering step. Since the bitmap index is more compact than the database, the savings from *sequential read* of fewer data subsequences and comparing fewer data subsequences is expected to outweigh the overhead of searching the index. The notations, which will be explained as they are used, are summarized in Table 1.

3.1. BIS Index Construction

The first step in index construction is to define a hash function

$$f : D \rightarrow N$$

to map each element d in the database to an index element n. For *BIS* to be effective, N_{bits}, the number of bits that make up an index element, has to be smaller than D_{bits}, the number of bits in a data element. This means that several data values will map to the same index value, hence several different data subsequences could be represented by the same index subsequence. Consequently, false drops may occur. If the false drop rate is too high, the overhead of examining the subsequences at the refinement step makes the whole processing strategy ineffective.

To ensure the filtering power of *BIS*, we would like the maximum number of data subsequences that can be represented by any index subsequence to be as small as possible. This happens when every index value is equally likely to occur, i.e., when N follows a uniform distribution. To derive such a hash function, we first find the frequency distribution of D by data sampling, then partition D into intervals with equal frequency, and finally assign each interval to an index value. The philosophy is somewhat similar to the bit strings in high-dimensional database indexing in [25].

To illustrate, suppose D is an unsigned char (1 byte) that is uniformly distributed between 0 and 255, and N is a single bit. If the hash function f maps data values from 0 to 171 to 0, and data values in [172, 255] to 1, the index value 0 will be 3 times as likely to occur as 1. In the worst case, this allows $\frac{3}{4} \times \frac{3}{4} = \frac{9}{16}$ of the subsequences to drop through the index filter for a 2-character query string. In contrast, if the hash function splits D into equal halves, only $\frac{1}{2} \times \frac{1}{2} = \frac{1}{4}$ of the subsequences are expected to filter through against any 2-character query string.

While the above illustration shows that N should be uniformly distributed if possible, D is *not* required to be uniformly distributed as well. Indeed, such a requirement would prove unrealistic as there are often some correlation among the elements in real-life data sequences. By analyzing the distribution of the data elements, whatever that may be, it is always possible to define a hash function f that produces a uniform index distribution during index construction. The challenge really is to make *BIS* robust against updates that shift the data distribution, and consequently the index distribution, at runtime. Indeed, our performance study indicates that the proposed strategy does not deteriorate appreciably unless N becomes highly skewed.

Having determined a hash function f, we then apply it to the elements of every sequence in the database to produce a bitmap index. The index preserves the relative position of the data elements. This property is pivotal in the runtime performance of *BIS*, as locating the data representation of those subsequences that filter through becomes straightforward. There is no need for any pointers between the index and data files, nor any computation overhead. The positional relationship also simplifies updates greatly.

Since the index construction process involves only a quick scan through the database, and possibly an earlier scan to determine the distribution of D, the process has a time complexity of O(DBSize) and requires only two I/O buffers, one for the database and one for the index. In terms of storage overhead, the index is $\frac{N_{bits}}{D_{bits}}$ the size of the database. Depending on whether D is a character, a short integer, an integer, a float or a double float, D could be 8, 16, 32, or 64 bits long. As for N, we restrict it to be a divisor or a multiple of a byte in our implementation to avoid grappling with index elements that straddle two bytes. We will show in the next section that this does not diminish the usefulness of *BIS* in practice.

Example 1. Suppose D is an unsigned character that is uniformly distributed in [0, 255], and N is 1 bit in size. We choose a hash function that maps [0, 127] to 0, and [128, 255] to 1; this hash function can be implemented very efficiently by a single comparison operation, or by extracting the leftmost bit in the binary representation of D. Given a data sequence [75 3 95 189 165 106 229 239   8 222 122 236 200 146 75 33   ...], the hash function would produce the index sequence [0 0 0 1 1 0 1 1   0 1 0 1 1 1 0 0   ...].

3.2. Filtering Step

Having introduced the index construction process, we shall explain how the filtering step uses the *BIS* index to speed up the search for subsequences that satisfy a submitted query string. We shall begin with the scenario where N is one or more bytes in size; sub-byte index elements necessitate bit manipulations that will be described shortly.

Given a query string of $QLen$ data elements, the filtering algorithm first applies the hash function f to derive the corresponding bitmap index representation. This bitmap representation of the query is then matched against each sequence in the index in turn. The filtering algorithm is designed to iterate over every sequence in the collection. In each iteration, the algorithm maintains pos, the position in the current sequence, and a state array $state[1..q]$. If and only if $state[i]$ = TRUE, then the last i index elements in the sequence match the first i index elements in the query string. The algorithm marches through each element in the index in turn; there is no backtracking. After reading a new index element n from the sequence, pos is advanced and the state array is updated as follows:

$$state[i] := \begin{cases} \text{(query[1] = n)} & \text{if } i = 1 \\ \text{(query[}i\text{] = n) AND state[}i-1\text{]} & \text{if } i > 1 \end{cases} \quad (1)$$

where $query[i]$ is the ith index element of the query string. Thus, whenever $state[QLen]$ = TRUE, the subsequence of length $QLen$ ending at pos "drops" through the index filter and its data representation is tested for a match. The algorithm is given below.

```
 Filter-and-Refine Subsequence Selection Algorithm

/* definition */
NumSeq = number of seq. in the database;
DSeq[i][j] = element at position j of data seq. i;
DIdx[i][j]= element at position j of data index i;
SeqLen[i] = number of elements in data seq./index i;
QSeq[i] = element at position i of query seq.;
QIdx[i] = element at position i of query index;
QLen = number of elements in query seq./index;

for i = 1 to NumSeq { /* filtering step */
 for j = 1 to QLen
   s[j] = FALSE;
 for j = 1 to SeqLen[i] {
   for k = QLen downto 2
     s[k] = ((QIdx[k] == DIdx[i][j]) AND s[k-1]);
   s[1] = (QIdx[1] == DIdx[i][j]);
   if (s[QLen] == TRUE)   /* refinement step */
     if (DSeq[i][(j-QLen+1)..j] == QSeq[1..QLen])
       output("Match at pos %d of seq. %d",j,i);
  }
}
```

Let us now focus on the case where N is less than a byte, the smallest unit that is directly addressable. Since we had restricted N_{bits} to be a divisor of a byte, the number of index elements per byte $\frac{BYTE}{N_{bits}}$ is an integer. We note that there are $\frac{BYTE}{N_{bits}}$ possible ways in which a matching index subsequence could be aligned with respect to the byte boundaries. In all but one of the alignments, the leading index elements of the subsequence only partially occupy a byte; the left side of the byte contains index elements belonging to other subsequences that must be masked out. The same is true of the trailing index elements of the subsequence, except that here the irrelevant index elements are located at the right side of the byte. To handle this situation, we need to modify the filtering sequence matching algorithm slightly.

We re-define $query[i]$ to be a byte in which the jth position from the right contains the $(i-j+1)$th index element of the query string. pos is now incremented by $\frac{BYTE}{N_{bits}}$ after each new index byte. Furthermore, the state array has $QLen + \frac{BYTE}{N_{bits}} - 1$ entries, and is updated as follows:

$$state[i] := \begin{cases} \text{(query[i] = Right(n, i))} \\ \quad \text{if } 1 \le i \le min(QLen, \frac{BYTE}{N_{bits}}) \\ \text{(query[i] = Left(Right(n, i), } QLen + \frac{BYTE}{N_{bits}} \text{ - i))} \\ \quad \text{if } QLen < i \le \frac{BYTE}{N_{bits}} \\ \text{(query[i] = n) AND state[i - } \frac{BYTE}{N_{bits}}\text{]} \\ \quad \text{if } \frac{BYTE}{N_{bits}} < i \le QLen \\ \text{(query[i] = Left(n, } QLen + \frac{BYTE}{N_{bits}} \text{ - i)) AND} \\ \qquad \text{state[i - } \frac{BYTE}{N_{bits}}\text{]} \\ \quad \text{if } max(QLen, \frac{BYTE}{N_{bits}}) < i < QLen + \frac{BYTE}{N_{bits}} \end{cases} \quad (2)$$

where $Left(n,i)$ and $Right(n,i)$ mask out the bits in n except for the i index elements from the left and right, respectively. The two functions are implemented efficiently by pre-computing a bit mask for each state entry. Whenever

$state[i]$ = TRUE for some $QLen \leq i < QLen + \frac{BYTE}{N_{bits}}$, and $pos + QLen > i$, the subsequence of length $QLen$ ending at position $(pos + QLen - i)$ drops through the index filter.

While the modified scheme is more complicated, it can deliver significant speed-ups. The reason is that each comparison operation matches $query[i]$ against $\frac{BYTE}{N_{bits}}$ index elements *concurrently* now, rather than only one at a time.

Now we shall prove the correctness of the algorithm.

Theorem 1. The *BIS* algorithm is correct, returning all matching subsequences and no false matches.

Proof: Two cases to consider: *Case 1.* Suppose that a subsequence a satisfies a query string q. This implies that both query strong and subsequence contain the same data representation. Therefore, $a[i] = q[i] \; \forall 1 \leq i \leq QLen$. It follows that $f(a[i]) = f(q[i])$ where f is the hash function, and that a and q have the same index representation too. Consequently, a will pass the filtering step and the subsequent refinement step.

Case 2. Suppose subsequence a does not match q. This implies $a[i] \neq q[i]$ for some $1 \leq i \leq QLen$, but $f(a[i]) = f(q[i])$. Since a drops through the index filter, its data representation will be examined. At this stage, *BIS* will discover that the ith data element is different and reject a.

We therefore conclude that the *BIS* algorithm is correct – all matching subsequences are returned, and there are no false matches. □

4. Extended Subsequence Matching

While the basic algorithm works only with query strings that comprise an arbitrary number of data elements, it can be extended easily to handle more general queries. Here, we extend it to support the following queries:

1. *Fixed length don't cares* (FLDC). Besides data elements, a query string may also include one or more FLDCs that are meant to match any data value. When updating the state array, the index elements that correspond to the FLDCs are simply masked out.

2. *Variable length don't cares* (VLDC). A VLDC separating the data elements in a query is supposed to match any string of contiguous elements in a subsequence. Unlike FLDCs, the introduction of VLDCs allows matching subsequences to be longer than the query string. In our implementation, we first extract from a query string those segments of contiguous data elements and FLDCs that are separated by the VLDCs. Next, we treat each segment as a separate query and find its set of matching subsequences. Finally, we return all combinations of subsequences in which the matching subsequence for segment i precede the matching subsequence for segment $i + 1$.

3. *Edit distance.* Instead of returning only subsequences that match a query string exactly, an application may require all subsequences that contain up to a certain number of mismatches; i.e., all subsequences that are within a certain edit distance $EditDist$ from the query. To accommodate this, we need only re-define the state array so $state[i]$ counts the total number of mismatches between the first i index elements of the query and the last i index elements in the current sequence. Whenever $state[i] \leq EditDist$ for any $i \geq QLen$, the subsequence ending at position $(pos + QLen - i)$ is deemed to have passed the filtering step.

4. *Alternative characters.* A user may need to specify a set of candidate characters for some position in the query string, rather than stating one single character or an FLDC. If all of the candidate characters hash to the same index value, the filtering step proceeds as per the single-character case; only subsequences that filter through need to be matched against the candidate set. If the candidate characters hash to different index values, they are treated as an FLDC in the filtering step, i.e., the index elements corresponding to that position are masked out when updating the state array. Since the presence of alternative characters does not require special handling in the filtering step, we shall not address it further.

5. Cost Model

To study the effectiveness and efficiency of the processing strategy, we develop a cost model. The purpose of the model is three-fold. First, when constructing an index for a sequence database, the model can provide the best setting for N_{bits}, the number of bits needed for each index element. Second, the model can help a database system to decide whether *BIS* will speed up a particular query. Third, the model can estimate the processing time if *BIS* is employed.

For reference purposes, we also present the cost of applying the Baeza-Yates and Gonnet algorithm [1] to the data sequences directly (denoted by BYG; a brief description of BYG algorithm is given in Section 7). Besides the algorithm parameters in Table 1, we shall also make use of the cost parameters in Table 2.

5.1. Cost of BYG

The BYG algorithm essentially marches through each data element in the sequence collection and compares it with the $QLen$ entries in the state array. This incurs the cost of retrieving the data element ($C_{I/O}$[1]), the cost of looping

[1] Obviously, we do not issue a disk I/O for each data element. Rather, data are retrieved in large blocks to reduce I/O costs. For this reason, $C_{I/O}$

Notation	Meaning
$C_{I/O}$	Avg. I/O time for fetching a data element in CPU cycles[1]. Default=75
C_{loop}	# of CPU cycles to loop through each state entry. Default=5
C_{state}	# of CPU cycles to compare data element with an entry in the query array and store the result in a state entry. Default=18
C_{string}	# of CPU cycles to match elements of two data sequences. Default=11.
C_{cache}	# of CPU cycles to cache interim results. Default=11
R_{BYG}	Response time of Baeza-Yates & Gonnet algorithm
R_{BIS}	Response time of *BIS* algorithm
$Speedup$	Response time speed-up, = $\frac{R_{BYG}}{R_{BIS}}$

Table 2. Cost Parameters

through the state array ($C_{loop} \times QLen$), and the cost of comparing the data element with $query[i]$, $1 \leq i \leq QLen$; see Formula (1). The expected cost of comparing $query[i]$ is $C_{state}/|D|^{i-1}$ where $|D|$ is the data alphabet size, as this comparison is done only if $state[i-1]$ = TRUE, i.e., all of the previous $i-1$ data elements matched. Since $|D| > 1$, the cost diminishes quickly as i increases. For simplicity, we will include only the first-order term. Therefore, the response time for matching a sequence collection is

$$R_{BYG} = NumSeq \times SeqLen \times (C_{I/O} + C_{loop} \times QLen + C_{state}) \tag{3}$$

This equation suggests that the I/O cost for retrieving and the CPU cost for testing each data element is independent of the number of bytes it contains. While this is not true in practice, in an efficient implementation the costs increase only very marginally as D goes from one to several bytes. We will demonstrate in the next section that dropping D_{bits} is acceptable.

5.2. Cost of *BIS* Processing Strategy

The proposed algorithm consists of two steps: (1) searching the index representation to weed out irrelevant subsequences at the filtering step, and (2) examining the data representation of subsequences at the refinement step. If each index element N occupies one or more bytes, the filtering cost alone will match R_{BYG}. Therefore N has to be smaller than a byte for the index *BIS* to be effective.

The computation of the filtering cost is similar to that of R_{BYG}. There are altogether $NumSeq \times SeqLen \times \frac{N_{bits}}{BYTE}$ bytes in the index. In processing each byte, there is an I/O cost ($C_{I/O}$), the cost of looping through the state array ($C_{loop} \times (QLen + \frac{BYTE}{N_{bits}} - 1)$), the cost of matching with $query[i]$, and the cost of recording subsequences that filter through (C_{cache}). According to Formula (2), we have to evaluate $Query[i]$ for at least $1 \leq i \leq \frac{BYTE}{N_{bits}}$. Hence, the filtering cost is

$$R_{filter} = NumSeq \times SeqLen \times \frac{N_{bits}}{BYTE} \times (C_{I/O} + C_{loop} \times (QLen + \frac{BYTE}{N_{bits}} - 1) + C_{state}\frac{BYTE}{N_{bits}} + C_{cache}) \tag{4}$$

Next, we consider the cost of matching the subsequences that filter through. These are expected to make up only $2^{-N_{bits} \times QLen}$ of the $NumSeq \times (SeqLen - QLen + 1)$ subsequences, as the hash function f is chosen so that N is (roughly) uniformly distributed between 0 and 1. For each of these subsequences, there is an I/O cost, the cost of looping through each element in the subsequence, and the cost of comparing with the query string. Here, we are fetching an entire subsequence all at once rather than element by element, so the I/O cost does not appreciate significantly with $QLen$. This is why the I/O cost is only $C_{I/O}$. Also, C_{string}, the cost of comparing with the query string, is lower than C_{state} because we are not updating the state array here. The refinement cost is, therefore,

$$R_{match} = \frac{NumSeq \times (SeqLen - QLen + 1)}{2^{N_{bits} \times QLen}} \times (C_{I/O} + C_{loop} \times QLen + C_{string}) \tag{5}$$

and the response time of *BIS* is

$$R_{BIS} = R_{filter} + R_{match} \tag{6}$$

5.3. Extended Subsequence Matching

Besides query strings consisting of contiguous data elements, the cost models presented above can also capture the cost of the extended queries. Only a few modifications are necessary.

1. *Fixed length don't cares* (FLDC). For a query string with length $QLen$ that includes i FLDCs, the response time of the BYG algorithm remains as in Equation (3). In the case of *BIS*, the filtering cost (Equation (4)) is not affected, but the cost of checking subsequences that filter through (Equation (5)) becomes:

   $$R_{match} = \frac{NumSeq \times (SeqLen - QLen + 1)}{2^{N_{bits} \times (QLen - i)}} \times (C_{I/O} + C_{loop} \times (QLen - i) + C_{string}) \tag{7}$$

   to account for the larger number of potentially matching subsequences.

is obtained by dividing the time needed to fetch a block, by the number of data elements in the block. The default of 75 CPU cycles is based on a block size of 8 KBytes.

Parameter	**Meaning**	**Default**
$NumSeq$	# of data seq. in the database	1
$SeqLen$	Avg. # of elements per data seq.	512 million
D_{dist}	Distribution of data element	uniform
D_{bits}	# of bits in a data element	8
N_{bits}	# of bits in an index element	1
$QLen$	# of query elements	16

Table 3. Experiment Parameters

2. *Variable length don't cares* (VLDC). As explained in Section 3.4, a query string with VLDCs is processed by combining the subsequences that match the segments separated by VLDCs. As the combination cost is relatively low, the overall cost is approximately the sum of the segment refinement costs. This applies to both R_{BYG} and R_{BIS}.

3. *Edit distance.* If all subsequences that are within an edit distance i from a query are needed, Equation (3) becomes

$$\begin{aligned} R_{BYG} = {} & NumSeq \times SeqLen \times \\ & (C_{I/O} + C_{loop} \times QLen + C_{state}(1+i)) \end{aligned} \tag{8}$$

because we now need to evaluate i more elements to disqualify an irrelevant subsequence. Similarly, Equation (4) is changed to

$$\begin{aligned} R_{filter} = {} & NumSeq \times SeqLen \times \tfrac{N_{bits}}{BYTE} \times \\ & (C_{I/O} + C_{loop} \times (QLen + \tfrac{BYTE}{N_{bits}} - 1) + \\ & C_{state}(\tfrac{BYTE}{N_{bits}} + i) + C_{cache}) \end{aligned} \tag{9}$$

In the case of R_{match}, the fraction of subsequences that filter through increases by a factor of $\sum_{j=1}^{i} \mathbf{C}_j^{QLen}$. The reason is that there are $\mathbf{C}_j^{QLen}$ different ways in which a subsequence can differ from the query string and yet have an edit distance of j. Accordingly,

$$\begin{aligned} R_{match} = {} & NumSeq \times (SeqLen - QLen + 1) \times \\ & min(1, \tfrac{\sum_{j=1}^{i} \mathbf{C}_j^{QLen}}{2^{N_{bits} \times QLen}}) \times \\ & (C_{I/O} + C_{loop} \times QLen + C_{string}(1+i)) \end{aligned} \tag{10}$$

6. Experimental Study

To study the performance of the proposed strategy, we have implemented the strategy *BIS*, and Baeza-Yates and Gonnet's algorithm [1], in the C language. The experiment platform consists of Sun UltraSparc 170 machines, each equipped with a 167 MHz CPU, 64 MB of memory, and a 2.1 GB Fast and Wide SCSI 2 hard disk. We have isolated and timed relevant portions of the implementation on this platform to obtain settings for the various cost parameters; the settings are listed in Table 2. For example, the *average* I/O time for fetching a data element, $C_{I/O}$, is set to 75 CPU cycles, or 450 nsec on a 167 MHz CPU (i.e., about 3.7 msec for each 8 KByte block).

The performance metrics that will be used to present the results are response time and speed-up, defined as R_{BYG}/R_{BIS}. For every experiment, we run 100 queries and average their results. Each query string is composed from randomly picked portions of one of the data sequences, so the query elements have the same distribution as the data elements. The parameters for the experiments are summarized in Table 3.

Apart from analytical experiments, the proposed algorithm was implemented as part of the system described in Section 2, which is being used by molecular research scientists for supporting their work.

6.1. Effect of N_{bits}

Before we construct an *BIS* index, we first need to determine a setting for N_{bits}, the number of bits per index element. We note that a smaller N_{bits} lowers filtering cost (Equation (4)) but raises R_{match} (Equation (5)). The key factor that decides whether the net effect is beneficial is the query length $QLen$. Since both R_{BYG} and R_{BIS} increase with $QLen$, we want to configure *BIS* for large $QLen$'s. With a large $QLen$, the number of subsequences that filter through is small regardless of N_{bits}. Consequently, filtering cost dominates, leading us to conclude that N_{bits} should be as small as possible, i.e., 1 bit.

To verify the above conclusion, we first build a database with $NumSeq = 1$, $SeqLen$ = 512 million elements, D following a uniform distribution and D_{bits} = 16. Different indices are then created for N_{bits} = 1, 2, 4, and 8. Finally, query strings of the form $q_1 q_2 .. q_{16}$ are generated and run against each of the indices, and against the database directly.

Figures 3 and 4 plot the average response times and the speed-ups, respectively, produced by both the experiment and the cost models. The experiment results agree with the estimations from Equations (3) and (6). Moreover, the results confirm that performance worsens as N_{bits} increases. We shall therefore set N_{bits} to 1 from now on.

6.2. Effect of Query Length

Having determined the best N_{bits} setting for index construction, we now need a criterion for deciding when *BIS* will speed up query processing. Referring to Equations (3),

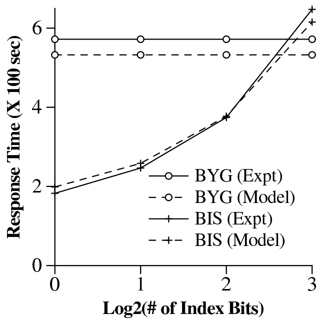


Figure 3: Response vs. N_{bits}

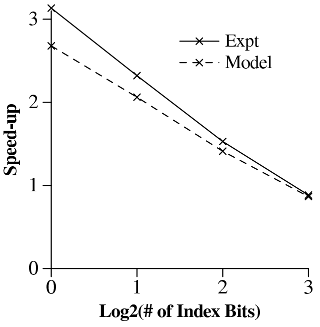


Figure 4: Speed-up vs. N_{bits}

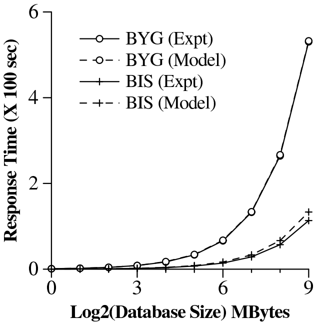


Figure 7: Response vs. DB Size

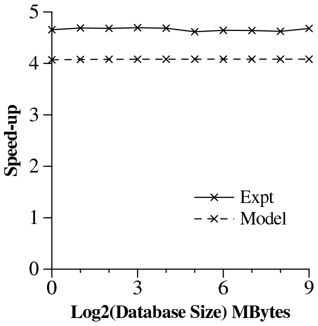


Figure 8: Speed-up vs. DB Size

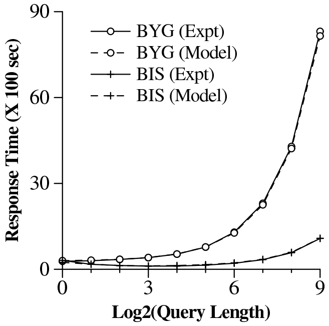


Figure 5: Response vs. Query Length

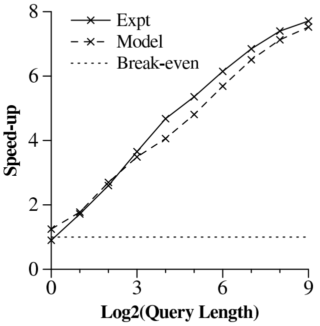


Figure 6: Speed-up vs. Query Length

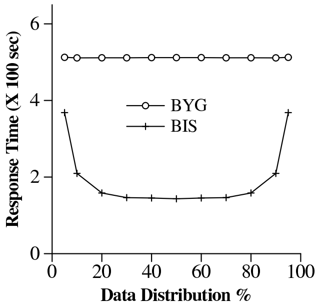


Figure 9: Response vs. Index Dist.

(4) and (5), we note that $NumSeq$ and $SeqLen$ get cancelled out in Speed-up = R_{BYG} / R_{BIS} when $SeqLen >> QLen$, leaving $QLen$ as the only variable. In other words, whether *BIS* is beneficial depends only on the length of the query string.

In this experiment, we set D_{bits} to 8, N_{bits} to 1, and vary $QLen$. The rest of the parameter settings remain as before. The resulting response times and speed-ups are plotted in Figures 5 and 6.

While it may not be obvious in Figure 4 because the x-axis represents $log_2 QLen$ rather than $QLen$, the time taken by the BYG algorithm increases linearly with $QLen$. This agrees with Equation (3). As for *BIS*, its response time reduces initially as savings from fewer subsequences filtering through dominate rising filtering cost. However, eventually filtering cost prevails, and the response time of *BIS* embarks on a gradual uptrend. Nevertheless, overall there is a net savings over BYG. As Figure 6 shows, *BIS* breaks even at $QLen$ = 2, and the speed-up rises roughly linearly with $log_2 QLen$. Again, the experiment results corroborate the cost models.

6.3. Effect of Data Size

In this experiment, we want to verify if the performance gain of the proposed strategy is sustainable as the database scales up. Theoretically, they should, as Speed-up = R_{BYG}/R_{BIS} is independent of $NumSeq$ and $SeqLen$.

For this experiment, we fix the query length at 16, $NumSeq$ at 1, and vary $SeqLen$. We create databases with $SeqLen$ ranging from one to 512 million. The results, given in Figures 7 and 8, confirm that response time increases linearly, and that the speed-up achieved remain constant, exactly as predicted by the cost models.

6.4. Effect of Data Distribution

While gene and protein collections are fairly static, it is important for any processing strategy to be dynamic and perform well under different data distributions with different skewness. It is therefore necessary to see how *BIS* copes with updates that shift the data and, consequently, index distributions at runtime. This experiment is intended to profile any adverse effect that skewed index distributions might have on the performance of *BIS*.

Here, we build a database with $NumSeq = 1$, $SeqLen = 512$ million, D following a uniform distribution and $D_{bits} = 8$. Various skewed index distributions are generated via a family of hash functions that map x% of the data values to 0, and the rest to 1. Having done that, we then run queries with $QLen = 16$ against each of the indices and against the database directly.

Figure 9 plots the response time against the filter element distribution x. The figure shows that the performance of *BIS* remains stable as x swings from 20% to 80%. As the filter element distribution becomes even more skewed, however,

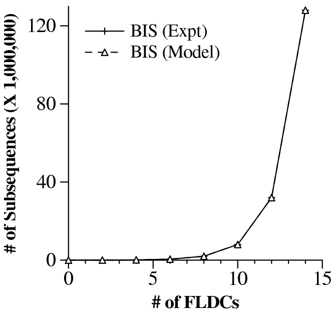


Figure 10: # Subsequences vs. FLDCs

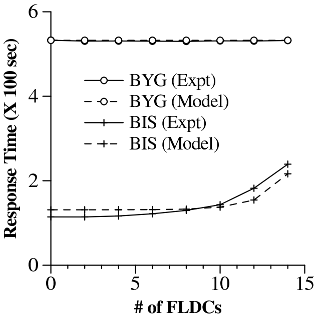


Figure 11: Response vs. FLDCs

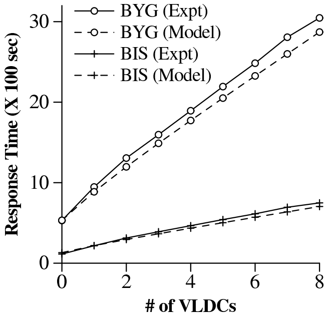


Figure 12: Response vs. VLDCs

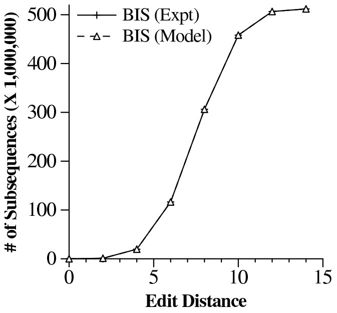


Figure 13: # Subsequences vs. Edit Dist.

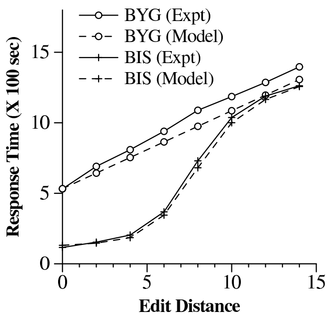


Figure 14: Response vs. Edit Dist.

BIS deteriorates rapidly, though it remains beneficial even at $x = 5\%$ and $x = 95\%$. Nevertheless, when the distribution becomes so skewed, the filter should be re-constructed with an updated hash function to restore *BIS*'s effectiveness.

6.5. Effect of Fixed Length Don't Cares Segments

In this experiment, we set $NumSeq = 1$, $SeqLen =$ 512 million, $D_{bits} = 8$, $N_{bits} = 1$ (uniformly distributed) and $QLen = 16$. Moreover, we introduce FLDCs at randomly selected positions in the query strings.

As Figure 10 indicates, the average number of subsequences that pass *BIS*'s filtering step in the experiment matches the cost model's prediction almost exactly. The number increases very slowly initially as we introduce more FLDCs. Consequently, *BIS*'s response time remains almost unchanged until the number of data elements in the query strings becomes less than six, as shown in Figure 11. This is yet another confirmation that filtering cost (Equation (4)) dominates refinement cost (Equation (7)) unless there are very few data elements in the query string. Even then, *BIS* is still faster than BYG, which is not affected by the FLDCs.

6.6. Effect of Variable Length Don't Cares Segments

Besides FLDCs, another kind of extended subsequence matching operations involves variable length don't cares (VLDC) in the query strings. Such queries are processed by matching the component segments separately and then combining the results. Consequently, the overall response time is the sum of the individual segment's processing time. This is confirmed in Figure 12, which is obtained with the same parameter settings as in the previous experiment, except that there are no FLDCs.

6.7. Effect of Edit Distance

The third kind of extended subsequence matching operations that we have implemented is support for edit distances. Using the same parameter settings as the previous experiment (minus the VLDCs), we run *BIS* and BYG with different edit distances. The experiment results, together with the estimations from Equations (8)-(10), are given in Figures 13 and 14. The number of subsequences that filter through in the experiment (plotted in Figure 13) again matches those obtained from Equation (10). As for response time, the agreement is not as good as before – the experiment shows *BIS* outperforms BYG up to an edit distance of 14, whereas the cost models indicate only 10. Nevertheless, both *BIS* and the cost models are useful for edit distances that are low relative to the query length.

Experimental results show that *BIS*'s speed-up improves with $QLen$, and *BIS* remains advantageous for larger edit distances if the query string is longer. For example, at $QLen = 32$, *BIS* outperforms BYG even when the edit distance reaches 30.

6.8. Effect of D_{bits}

In deriving the cost models, we have omitted the impact of D_{bits} on the ground that, in an efficient implementation, both CPU and I/O costs increase only marginally as D goes from one to several bytes. To verify this, we ran several experiments to investigate the impact of D_{bits} at different $QLen$ and N_{bits} settings. Figures 15 and 16 give the response times produced by BYG and *BIS* in one of these experiments, where $QLen = 16$ and $N_{bits} = 1$. As the figures show, the variations introduced by higher D_{bits}'s have

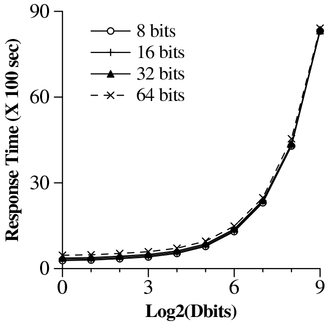


Figure 15: BYG vs. D_{bits}

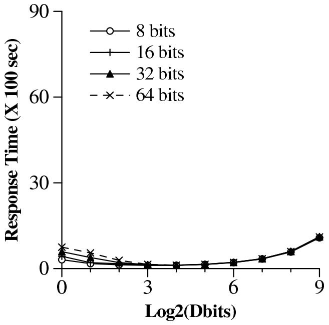


Figure 16: *BIS* vs. D_{bits}

little effect on either the efficacy of *BIS* or the accuracy of the cost models.

7 Related Work

Many algorithms have been proposed to address the problem of approximate subsequence matching. In this section, we shall discuss work related to our proposal, and highlight the differences between them.

The general approach to subsequence matching is to evaluate every sequence in response to a query, by constructing for it an automaton or a dynamic-programming table. Algorithms of such nature include the classical Knuth-Morris-Pratt (KMP) [9] and Boyer-Moore (DM) [5] algorithms, and algorithms proposed in [16, 19, 10, 1, 27]. If these algorithms are applied directly to the data sequences, the processing time may be unacceptably long, especially if the sequence database is large. However, they can be employed in conjunction with *BIS* to examine its index sequences. An earlier work related to our own was carried out by Karp and Rabin [7]. In [7], the authors presented a scheme in which each *subsequence* is mapped to an index number. Subsequently, rather than the longer, original subsequences, the index numbers are employed to search for matching subsequences. This scheme is similar to *BIS* in using a filter to weed out most of the irrelevant subsequences. However, there is a fundamental difference: Since the subsequences to consider are not defined until the query is known (as they need to have the same length), Karp and Rabin's scheme is not suitable for pre-indexing in database systems that must process ad-hoc queries. In contrast, *BIS* encodes and maps each *data element* to an index element during index construction. The processing of subsequence matching is performed dynamically at runtime, at which point the bitmap index representations are *dynamically* defined from the index that correspond to the data elements in the subsequences. Hence, *BIS* offers greater flexibility as it is not tailored to a particular query.

BYG [1] is a well-known algorithm for string matching with mismatches, which consists in representing the state of the search by a bit number and, at each step, in performing a number of bit operations. The algorithm searches a pattern in a text (without errors) by parallelizing the generation of a non-deterministic finite automation that looks for the patterns. For a search pattern of length m, and a text of length n, the automation has $m + 1$ states. The algorithm first builds a table B which for each alphabet character c stores a bit mask $B[c] = b_m..b_i$. The mask in $B[c]$ has the bit b_i in one if and only if the ith character in the pattern is equal to *c*. The state of the search in kept in a machine word $D = d_m..d_1$, where d_i is one whenever $P_{1..i}$ matches the end of the text read up to now (i.e. the ith state). A match is reported whenever $d_m = 1$. D is set to 1^m originally, and for each new text character T_j, D is updated using the formula *D' = ((D* $>>$ 1*)* $|$ *10*$^{m-1}$*)* & *B[T*$_i$*]*$))$, where $>>$ is the bitwise shift, $|$ is the bitwise OR, and & is the bitwise AND operation. For patterns longer than the computer word, the algorithm uses $\lceil$*m/w*$\rceil$ words (w is the length of a computer word). The algorithm achieves *O(mn/w)* worst-case time, *O(m/w (m*+$|\Sigma|$)) preprocessing time, and *O(m/w* $|\Sigma|$) extra space, where Σ denotes the alphabet.

An alternative approach to subsequence matching is to build a suffix tree [24, 12] index for the sequence database in advance, and to search the index rather than the actual data sequences at runtime. Solutions based on this approach include those reported in [11, 20, 22]. Suffix trees suffer from a number of performance problems.

For a sequence database consisting of l elements, each with a size of D_{bits} bits, a suffix tree index may need up to l leaf nodes and $l - 1$ internal nodes [17]. Since each node stores its corresponding starting and ending positions in a data sequence, together with a child pointer and a sibling pointer, the suffix tree could reach 32 times the database size with typical 4-byte fields. Moreover, many nodes need to be traversed for long query strings, while the presence of *don't care* segments blows up the number of branches that need to be searched, all of which lead to significant performance degradation. Finally, update is expensive as it affects both the structure, and the starting and ending positions recorded in multiple leaf nodes.

In [6], Faloutsos et al took a different approach. They proposed an algorithm that extracts the features in a window as it slides over the data sequence, thus transforming the sequence into a trail in a multi-dimensional feature space. The trail is then divided into sub-trails that are represented by their minimum bounding rectangles, and are indexed using traditional spatial access methods like the R^*-tree [4]. The algorithm is designed for sequences of continuous numbers, where the metric is the average Euclidean distance between a data element and the corresponding query element. In contrast, our algorithm targets sequences of discrete symbols, where the concern is whether a data element has the same value as the corresponding query element. For example, the subsequences ADC and AEC have different distances from ABC under Faloutsos' algorithm, whereas both are equally good/bad matches under our algorithm.

Recently, bitmap indexing has been used increasingly in other applications such as data warehouse query processing [26] and attribute based query processing [13] [15]. In [25], the VA-file (vector approximate file) which uses short bit strings to represent attribute values to index high-dimensional databases for similarity search. The performance gain due to shorter compact presentation and linear scan of the transformed vector file makes it a simple and yet one of most efficient high-dimensional indexes so far. Although the applications are different, the design philosophy is similar: they are designed to reduce index space overhead while improving query performance.

In [29, 30] Altschul et al. proposed the BLAST technique to find local similarities. BLAST, the most popular string matching tool for biologists, runs in two phases. In the first phase, all the substrings of the query of some pre-specified length (typically between 3 and 11) are searched in the database for an exact match. In the second phase, all the matches obtained in the first phase are extended in both directions until the similarity between the two substrings falls below some threshold. This technique keeps a pointer to the starting locations of all possible substrings of the pre-specified length in the database to speedup the first phase. Therefore, the space requirement of BLAST is more than the size of the database. Furthermore, BLAST does not find a similar substring to the whole query string, only similarities between the query substrings and the database substrings.

8 Conclusion

In this paper, we propose a filter-and-refine two-step processing strategy for approximate subsequence matching in large sequence databases. It employs *BIS* to map a data element to an index element that is smaller in size. The hash function is applied to every data element in a sequence and to each sequence in the database to produce an index. When a query string is presented, the filtering step evaluates the index representation of all the subsequences to isolate those that might be relevant. Only this fraction of subsequences need to have their data representation tested for a match at the refine step. Since the index is more compact than the database, doing so is expected to shorten the response time while saving on index storage space.

To evaluate performance, we have developed a cost model. The model enables us to determine the best size for an index element during *BIS* index construction. It also helps a DBMS to decide when to exploit *BIS* for query processing. Finally, the model can estimate the achievable response time savings over searching the database directly.

Extensive experiments were conducted using both synthetic and genetic sequence databases. The results of these experiments agree closely with the estimations of the cost model. More importantly, the experiments consistently confirm that *BIS* is space efficient, that it significantly reduces response time, and that it scales up with the database. For example, by constructing an index that is $\frac{1}{8}$ the size of the database, we have achieved more than 5 times speed-ups for query strings that are longer than 100 data elements.

We are currently conducting experiments and analysis using the large genomic databases, such as GenBank [31, 32] which contains approximately 1.3 billion bases, to develop the confidence in these early, but intriguing results. We also plan to improve the algorithm by compressing the bitmap index: frequent substrings in the sequences and queries can be encoded into much shorter index elements, and the filtering step can be speed up by the quick lookup of the frequent substrings in the index.

References

[1] R. Baeza-Yates, G.H. Gonnet, "A New Approach to Text Searching", *Communications of the ACM, Vol. 35, No. 10*, pp 74-82, October 1992.

[2] A. Bairoch, P. Bucher, "PROSITE: Recent Development", *Nucleic Acids Research, Vol. 22, No. 5*, pp 3583–3589, 1994.

[3] A. Bairoch, P. Bucher, K. Hofmann, "The PROSITE database, its status in 1997", *Nucleic Acids Research, Vol. 25, No. 1*, pp 217–221, January 1997.

[4] N. Beckmann, H.P. Kriegel, R. Schneider, B. Seeger, "The R^*-tree: An Efficient and Robust Access Method for Points and Rectangles", *Proc. of the ACM SIGMOD Conf.*, pp 322-331, May 1990.

[5] R.S. Boyer, J.S. Moore, "A Fast String Searching Algorithm", *Communications of the ACM, Vol. 20, No. 10*, pp 762-772, October 1977.

[6] C. Faloutsos, M. Ranganathan, Y. Manolopoulos, "Fast Subsequence Matching in Time-Series Databases", *Proc. of the ACM SIGMOD Conf.*, pp 419-429, May 1994.

[7] R.M. Karp, M.O. Rabin, "Efficient Randomized Pattern-Matching Algorithms", *IBM Journal of Research and Development, Vol. 31, No. 2*, pp 249-260, March 1987.

[8] G. Kadare and A.-L. Haenni, "Virus-encoded RNA helicases", *Journal of Virology, Vol. 71, No. 4*, pp 2583-2590, April 1997.

[9] D.E. Knuth, J.H. Morris, V.R. Pratt "Fast Pattern Matching in Strings", *SIAM Journal on Computing, Vol. 6, No. 2*, pp 323-350, June 1977.

[10] G.M. Landau, U. Vishkin, “Efficient String Matching in the Presence of Errors”, *Proc. of the 26th IEEE Symp. on Foundations of Computer Science*, pp 126-136, October 1985.

[11] G.M. Landau, U. Vishkin, “Fast Parallel and Serial Approximate String Matching”, *Journal of Algorithms, Vol. 10, No. 2*, pp 157-169, June 1989.

[12] E.M. McCreight, “A Space-Economical Suffix Tree Construction Algorithm”, *Journal of the ACM, Vol. 23, No. 2*, pp 262-272, April 1976.

[13] P. O’Neil, D. Quass, “Improved Query Performance with Variant Indexes”, *Proc. of the ACM SIGMOD Conf.*, pp 38-49, 1997.

[14] K. Osatomi, H. Sumiyoshi, “Complete Nucleotide Sequence of Dengue Type 3 Virus Genome RNA”, *Virology, Vol. 176, No. 2*, pp 643-647, 1990.

[15] D. Rinfret, P. O’Neil, E. O’Neil, “Bit-Sliced Index Arithmetic”, *Proc. of the ACM SIGMOD Conf.*, 2001.

[16] P.H. Sellers, “The Theory and Computation of Evolutionary Distances: Pattern Recognition”, *Journal of Algorithms, Vol. 1, No. 4*, pp 359-373, December 1980.

[17] G.A. Stephen, *String Searching Algorithms*, Lecture Notes Series on Computing and Problems, World Scientific, 1994.

[18] K.S. Trivedi, *Probability and Statistics with Reliability, Queuing, and Computer Science Applications*, Prentice-Hall, Inc., pp 132, 1982.

[19] E. Ukkonen, “Finding Approximate Patterns in Strings”, *Journal of Algorithms, Vol. 6, No. 1*, pp 132-137, March 1985.

[20] E. Ukkonen, D. Wood, “Approximate String Matching with Suffix Automata”, *Algorithmica, Vol. 10, No. 5*, pp 353-364, November 1993.

[21] E. Ukkonen, “On-line Construction of Suffix Trees”, *Algorithmica, Vol. 14, No. 3*, pp 249-260, September 1995.

[22] J.T.L. Wang, G.W. Chirn, T.G. Marr, B. Shapiro, D. Shasha, K. Zhang, “Combinatorial Pattern Discovery for Scientific Data: Some Preliminary Results”, *Proc. of the ACM SIGMOD Conf.*, pp 115-125, May 1994.

[23] R.A. Wagner, M.J. Fischer, “The String-to-String Correction Problem”, *Journal of the ACM, Vol. 21, No. 1*, pp 168-173, January 1974.

[24] P. Weiner, “Linear Pattern Matching Algorithms”, *Proc. of the IEEE 14th Annual Symposium on Switching and Automata Theory*, pp 1-11, 1973.

[25] R. Weber and H. Schek and S. Blott, “A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces”. *Proc. 24th International Conference on Very Large Data Bases*, pp 194-205, 1998.

[26] M.-C. Wu, “Query optimization for selections using bitmaps”, *Proc. of the ACM SIGMOD Conf.*, pp 227-238, 1999.

[27] S. Wu, U. Manber, “Fast Text Searching Allowing Errors”, *Communications of the ACM, Vol. 35, No. 10*, pp 83-91, October 1992.

[28] P. Zezula, F. Rabitti, P. Tiberio, “Dynamic Partitioning of Signature Files”, *ACM Trans. on Information Systems, Vol. 9, No. 4*, pp 336-369, October 1991.

[29] Altschul, S.F., Gish, W., Miller, W., Myers, E.W., Lipman, D.J., “Basic Local Alignment Search Tool”, *Journal of Molecular Biology, Vol 215*, pp 403-410, 1990.

[30] Altschul, S.F., Madden, T.L., Schffer, A.A., Zhang, J., Zhang, Z., Miller, W., Lipman, D.J., “Gapped BLAST and PSI-BLAST: A New Generation of Protein Database Search Programs”, *Nucleic Acids Research, Vol 25*, pp 3389-3402, 1997.

[31] Benson DA, Karsch-Mizrachi I, Lipman DJ, Ostell J, Rapp BA, Wheeler DL, “GenBank”, *Nucleic Acids Research, Vol 28, No. 1*, pp 15-18, 2000.

[32] http://www.ncbi.nlm.nih.gov/Genbank/index.html