ACADEMIC
World Academic Union

ISSN 1746-7659, England, UK
Journal of Information and Computing Science
Vol. 3, No. 4, 2008, *pp*. 243-257

An edit operation-based approach to approximate string matching in large DNA databases

Jiun-Rung Chen and Ye-In Chang [+]

Dept. of Computer Science and Engineering, National Sun Yat-Sen University Kaohsiung, Taiwan

(*Received March 14, 2008, accepted August 20, 2008*)

Abstract. In DNA related research, due to various environment conditions, *mutations* occur very often, where a mutation is defined as a heritable change in the DNA sequence. Therefore, *approximate string matching* is applied to answer those queries which find mutations. The problem of approximate string matching is that given a user specified parameter, k, we want to find where the substrings, which could have k errors at most as compared to the query sequence, occur in the database sequences. In this paper, we make use of a new index structure to support the proposed method for approximate string matching. In the proposed index structure, *EII*, we map each overlapping q-gram of the database sequence into an index key, and record occurring positions of the q-gram in the corresponding index entry. In the proposed method, *EOB*, we first generate all possible mutations for each gram in the query sequence. Then, by utilizing information recorded in the EII structure, we check both local order (*i.e.*, the order of characters in a gram) and global order (*i.e.*, the order of grams in an interval) of these mutations. The final answers could be determined directly without applying dynamic programming which is used in traditional filter methods for approximate string matching. From the experiment results, we show that our method could outperform the ($k + s$) q-samples filter, a well-known method for approximate string matching, in terms of the processing time with various conditions for short query sequences.

Keywords: databases, approximate string matching, DNA, mutation, similarity search

1. Introduction

Because of the Human Genome Initiative, an international research program for the creation of detailed genetic and physical maps of the human genome, enormous quantities of genome data, *e.g.*, DNA and protein sequences, are generated [7]. DNA sequences, holding the code of life of every living organism, could be considered as strings over an alphabet of four characters, {A, C, G, T}, called bases [10][28]. DNA sequences could be very long. For example, a human genome (DNA) contains around 3Gbp (giga base pairs). Searching patterns in the databases of DNA sequences is usually the first and a crucial step in DNA related research [6][11][18].

String matching methods are usually applied when searching in the databases. The domain of string matching could be separated into two parts, exact string matching and approximate string matching [8]. For exact string matching, we want to find where the query sequence occurs in the database sequences.

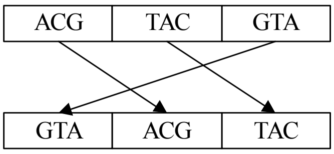


Fig. 1: An example of local order and global order

For approximate string matching, given a user specified parameter k, we want to find where the sequences, which could have k errors at most as compared to the query sequence, occur in the database sequences [14]. In the DNA related research, due to exposure to ultra violet radiation or the other environment condition, mutations occur very often, where a mutation is defined as a heritable change in the

[+] Corresponding author. *Tel*.: +886-7-525 2000 (ext. 4334); *fax*: +886-7-525 4301.
E-mail address: changyi@cse.nsysu.edu.tw.

Published by World Academic Press, World Academic Union

DNA sequence and is caused by a faulty replication process [5]. There are several kinds of errors which may occur in the replication process [17]: (1) Replacement: one character of the original sequence is replaced by another character, for example, ACGT and ACCT; (2) Insertion: one character is inserted into the original sequence, for example, ACGT and ACGAT; (3) Deletion: one character of the original sequence is deleted, for example, ACGT and ACT. These errors are kinds of edit operations in the string matching problem. Therefore, approximate string matching is applied to answer those queries which find mutations.

The genomic databases, GenBank, the DNA Databank of Japan, and the European Molecular Biology Laboratory database (EMBL), are used to assist molecular biologists to determine the biochemical function and the chemical structure of query strings, and to investigate the evolutionary history of organisms [27]. There are several principal database search methods used to compute pairwise comparisons between a candidate query sequence and each of the sequences stored within a database, e.g., Smith-Waterman, FASTA, and BLAST. The Smith-Waterman method [20] uses dynamic programming to compute the optimal pairwise similarity alignments. It computes all values of a two-dimensional array with size (m ∗ n) to determine which areas in the database sequence matches the query sequence, where m and n are the lengths of the query sequence and the database sequence, respectively. However, in DNA databases, the length of the database sequence may be up to several millions. Computing all values of such a big array is almost impractical [16]. Although not as optimal as the Smith-Waterman method, FASTA [12] and BLAST [1] methods provide a trade-off between comparison accuracy versus execution time. However, these methods may still run inefficiently and need large amounts of memory [13].

On the other hand, filter methods are quite new and currently very active for approximate string matching. In practice, filter methods are the fastest ones [9]. The basic idea of the filter methods is to filter out those substrings which are impossible to be the answers in the database sequence. For those substrings which pass the check of the filter methods (called candidates), they will be verified with the dynamic programming approach to determine whether they are truly the final answers. To find candidates, these filter methods usually check whether there are enough q-grams of the query sequence occurring within an interval of the database sequence, where a q-gram consists of q continuous characters in a sequence. If there are enough q-grams occurring within an interval, this interval will be a candidate. For filter methods, considering local order and global order of these q-grams is very important. Local order is the order of characters in a q-gram, while global order is the order of q-grams in an interval. For those q-grams within two sequences, they may maintain local order but no global order. For example, in Fig. 1, the 3-grams in these two sequences maintain local order, while they do not maintain global order.

Among the filter methods, the counting filter [15] considers neither local order nor global order. Although other methods, e.g., the q-gram filter [25], LET & SET filters [3], the l-tuple filter [4], and the h-samples filter [24], consider local order, they do not consider global order. So far, only the series of the (k + s) q-samples [16][21][22][23] considers the global order. Therefore, the (k + s) h-samples filter generates candidates more accurately than other filter methods, since it generates less number of candidates than other filter methods. However, it still has several disadvantages. The (k + s) h-samples filter still needs to apply dynamic programming to verify those generated candidates. When the length of the database sequence is very long (e.g., a DNA sequence), even though k is small, dynamic programming still needs a lot of time. Moreover, the (k + s) q-samples filter builds the index based on the query dynamically. When the query changes, it needs to rebuild the index. It also takes the query time to build the index, which increases the response time for a query.

Therefore, in this paper, to avoid those disadvantages mentioned above, we make use of a new index structure to support the proposed method for approximate string matching. In the proposed index structure, EII, we map each overlapping q-gram of the database sequence into an index key. The occurring positions of each gram are recorded in the corresponding index entry. Therefore, we could efficiently find the occurring positions of any q-gram by its index key. The EII structure is also pre-built for the database sequences, instead of being built for each query at the query time in those previous filter methods. In the proposed method, EOB, given a query sequence, P, we first generate all possible mutations for each q-gram in P with up to k errors. Then, by utilizing information recorded in the EII structure, we efficiently check local order and global order of the mutations of q-grams in P, and determine the answers directly. That is, different from those filter methods which generate candidates and verify them by dynamic programming, we utilize the EII structure to help us find the answers. Therefore, we could save the time for verifying those candidates with the dynamic programming approach. From the experiment results, we show that for short query sequences, our method could outperform the (k + s) q-samples filter in terms of the processing time with various conditions.

The rest of this paper is organized as follows. In Section 2, we give a survey of the (k + s) q-samples filter. In Section 3, we present the proposed index structure and the searching method. In Section 4, we discuss the performance of the proposed method. Finally, in Section 5, we give the conclusion.

2. The (*k* + *s*) *q*-Samples Filter

Sutinen and Tarhio have proposed several filter methods for approximate string matching [16][21][22][23]. These methods focus on ($k + s$) consecutive q-samples, and thus we call these methods "the ($k + s$) q-samples filter" in this paper. The ($k + s$) q-samples filter generalizes the filter of [24], and improves its filtering efficiency. This is the first filter which takes into account the relative positions of the pattern pieces that match in the text, while other previous filters match pieces of the pattern in any order. The generalization is to force s q-grams of the pattern to be matched, not just one. The pieces must conserve their relative ordering in the pattern, and must not be more than k characters away from their correct positions; otherwise, the number of errors will be larger than k. This method is illustrated in Fig. 2.

In this case, the sampling step is reduced to $h = \lfloor (m - k - q + 1) / (k + s) \rfloor$. The reason for this reduction is that to ensure s pieces matched, we need to cut the pattern into ($k + s$) pieces. The pattern is divided into ($k + s$) pieces, and a hash set is created for each piece so that the pieces are forced not to be too far away from their correct positions. The set contains the q-grams of the piece and some neighboring ones, too, because the sample can be slightly misaligned. At the search time, instead of a single q-sample, they consider text windows of consecutive sequences of ($k + s$) q-samples. Each of these q-samples is searched in the corresponding set. If at least s q-grams are found, the area of T $[(j - 1) - m'h - 2k - q + 2, (j - 1) + m - (m' - 1)h + k - q]$ is verified, and variable m' is updated to ($k + s$). In Fig. 2, it uses an array, M, with size m' to implement the Shift Add approach [2], to compute the sum of matches for the ($k + s$) consecutive samples. This is a sort of Hamming Distance, and the authors resort to an efficient method for that distance to process the text.

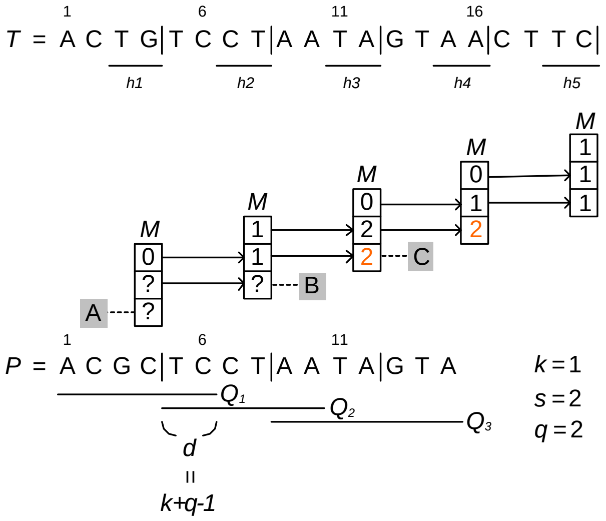


Fig. 2: The ($k + s$) q-samples filter

3. The Proposed Index Structure and Method

In this section, first, we present the proposed index structure, *EII*. Next, we present the proposed method, *EOB*, for approximate string matching in DNA databases.

3.1. The EII Structure

Although the inverted index [28] is a simple approach which does not need large storage space as compared to the suffix tree, it may lose some information at the end of the target sequence. Therefore, in this subsection, based on a revised version of the inverted index, we present the *EII* (Encoded Inverted Index) structure for efficiently indexing DNA sequences, which could avoid those missing cases. Similar to the inverted index, the EII structure records a postings list (*i.e.*, a list of occurring positions) for each overlapping q-gram in DNA sequences. However, to avoid missing cases, we append ($q - 1$) “\$” at the end of each DNA sequence. Fig. 3 shows an example of the EII structure with $q = 3$. In this example, we will append 2 (= 3 − 1) “\$” at the end of the DNA sequence, as shown in Fig. 3-(a). Therefore, not only the original 3-grams, *i.e.*, “ACG”, “CGA”, “GAC”, “ACG”, and “CGT”, but also two more 3-grams, “GT\$” and “T\$\$”, will be recorded in the EII structure. By appending “\$”, we also record the occurring positions of “GT” and “T”, which will be missed in the original inverted index. To efficiently find the searched grams, each searched gram in the EII structure is encoded into an index key and stored in an index array (called *IA*) in an ascending order, instead of being directly stored as a string in the original inverted index. The encoding rule is {A → 0, C → 1, G → 2, T → 3, \$ → 4}. For example, gram “ACG” is encoded into $(012)_5$ (*i.e.*, $0 * 52 + 1* 51 + 2* 50 = 7$ in the decimal form). The postings list of a q-gram will be recorded in the corresponding entry of *IA*, as shown in Fig. 3-(b). For example, $IA[(012)_5] = IA[7] = \{0, 3\}$ means that gram “ACG” occurs at positions 0 and 3 of the DNA sequence. Therefore, the task of searching q-grams by string comparison in original inverted indexes becomes an efficient array access in the EII structure. The total size of *IA* is 5^q, since each q-gram consists of some characters of {A, C, G, T, \$}.

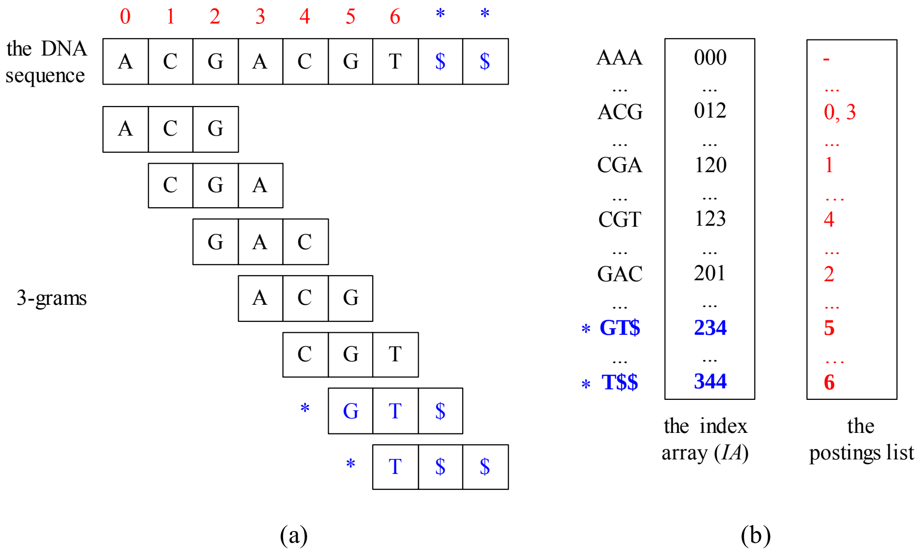


Fig. 3: An Example of the EII structure for sequence “ACGACGT” and $q = 3$: (a) overlapping 3-grams; (b) the EII structure.

3.2. The EOB Method

In this subsection, we present the *EOB* (Edit Operation-Based) method for approximate string matching in DNA databases. The basic idea of the EOB method is that for each query sequence, we separate it into non-overlapping q-grams, where q is the parameter used in the EII structure. In order to avoid an exhaustive process, instead of processing the entire query sequence at a time, we process only one q-gram to obtain a local answer in each round, where a local answer stores a set of occurring positions of those substrings in the database sequence which have the same local order as the given q-gram. We will recursively process all q-grams in the query sequence. After processing all q-grams, the global answer is derived from local answers found in each round, where a global answer maintains the same global order of the whole q-grams (*i.e.*, all of the local answers) as the query sequence.

Fig. 4 shows the EOB method, which has the following 4 main steps:

(1) Fetch one q-gram from the query sequence, *QuerySeq*. If the length of *QuerySeq* is less than q, we append “#” to *QuerySeq*, such that the length of *QuerySeq* equals q, where “#” means that it can be replaced by any character of {A, C, G, T, \$}. For example, assume that q is 3 and *QuerySeq* is “AC”. By searching “AC#” from the EII structure, we could obtain all occurring positions of “AC” in the database sequence. (Note that since we will remove characters from *QuerySeq* in Steps 3 and 4 in each round, its length may be less than q in the last round.)

(2) Generate all possible cases of mutations according to the number of errors which could occur in this gram.

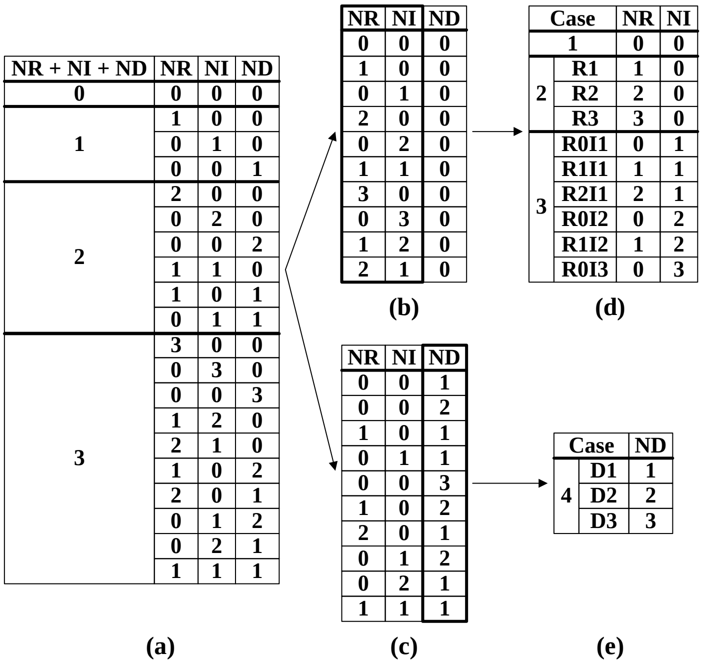
(3) Consider this q-gram and its mutations with Replacement and Insertion operations to derive a local answer. If there is no local answer found in this round, we do not need to process the rest of QuerySeq and could terminate the process early.

(4) Consider the mutations with Deletion operations.

These steps will be recursively applied to process all *q*-grams in the query sequence, and finally the global answer will be gradually derived from local answers. The details of Steps 2, 3, and 4 will be described in the following subsections.

```
procedure EOB(QuerySeq, k, TempAns, RoundNum);
begin
    if (|QuerySeq| < q) then append (q − |QuerySeq|) "#" to QuerySeq; // Step 1
    CurrentGram := FirstQGram(QuerySeq);
    MinKQ := Min(k, q);                                              // Step 2
    CombinAll := MathCombin(MinKQ);
    CombinRI := FetchRI(CombinAll);
    CombinD := FetchD(CombinAll);
    ConsiderMutRI(CurrentGram, CombinRI, TempAns, RoundNum);    // Step 3
    ConsiderMutD(CurrentGram, CombinD, TempAns, RoundNum);      // Step 4
end;
```

Fig. 4: The EOB method



NR + NI + ND	NR	NI	ND
0	0	0	0
1	1	0	0
	0	1	0
	0	0	1
2	2	0	0
	0	2	0
	0	0	2
	1	1	0
	1	0	1
	0	1	1
3	3	0	0
	0	3	0
	0	0	3
	1	2	0
	2	1	0
	1	0	2
	2	0	1
	0	1	2
	0	2	1
	1	1	1

(a)

NR	NI	ND
0	0	0
1	0	0
0	1	0
2	0	0
0	2	0
1	1	0
3	0	0
0	3	0
1	2	0
2	1	0

(b)

NR	NI	ND
0	0	1
0	0	2
1	0	1
0	1	1
0	0	3
1	0	2
2	0	1
0	1	2
0	2	1
1	1	1

(c)

Case		NR	NI
1		0	0
2	R1	1	0
	R2	2	0
	R3	3	0
3	R0I1	0	1
	R1I1	1	1
	R2I1	2	1
	R0I2	0	2
	R1I2	1	2
	R0I3	0	3

(d)

Case		ND
4	D1	1
	D2	2
	D3	3

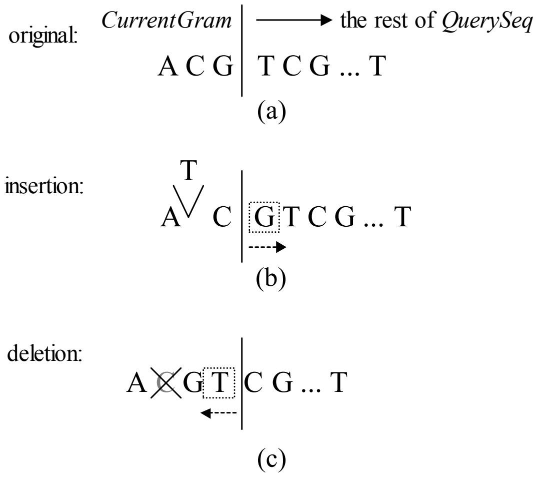
(e)

Fig. 5: All possible cases of mutations for *MinKQ* = 3 (i.e., $NR + NI + ND \leq 3$): (a) set *CombinAll*; (b) the subset of *CombinAll* with $ND = 0$; (c) the subset of *CombinAll* with $ND \neq 0$; (d) set *CombinRI*; (e) set *CombinD*.

3.3. Step 2

In this step, we generate all possible cases of mutations of *CurrentGram*. This is because we will search the occurring positions of these possible mutations from the EII structure in the following steps. The possible number of errors in a *q*-gram is the minimum value of *k* and *q* (denoted as *MinKQ*). That is, we consider at most *q* errors for each gram, even $k > q$. For the remaining $(k - q)$ errors, they will be considered in the following grams, if $k > q$. Fig. 5 shows all possible cases for *MinKQ* = 3, where *NR*, *NI*, and *ND* are the numbers of Replacement, Insertion, and Deletion operations, respectively. Since the possible number of errors in a *q*-gram is *MinKQ* at most, we have $NR + NI + ND \leq MinKQ$. Fig. 5-(a) shows the set of all possible solutions (denoted as *CombinAll*) for $NR + NI + ND \leq 3$.

Then, in order to simplify the problem, we consider cases of mutations without Deletion and with Deletion operations separately. Fig. 6 shows an example of Insertion and Deletion operations in the EOB algorithm, where Fig. 6-(a) shows the original *q*-gram before applying any operations. Basically, when applying any number of Replacement operations, it will not affect the length of the current q-gram for



(a)

(b)

(c)

Fig. 6: An example of Insertion and Deletion operations for q = 3: (a) the original gram; (b) the gram after insertion; (c) the gram after deletion.

data search in the EII structure. For w Insertion operations ($w \leq q$), we could simply “push” the last w characters of the current gram to the front of the next gram, to keep the length of the current gram equal to q, as shown in Fig. 6-(b). These pushed characters will be processed in the next round (which will be described in detail later).

However, for Deletion operations, they will result in the decrease of the length of the current gram. If we directly “pull” those characters which are originally in the next gram to the current gram (as shown in Fig. 6-(c)), it will cause one problem: all kinds of mutations for those pulled characters should also be considered, which results in a very complex method for solving this case. For example, in Fig. 6-(c), for the pulled character, ‘T’, we still need to consider those mutations of applying Replacement, Insertion, and Deletion operations on ‘T’, in addition to the original remaining gram, “AG”. Therefore, to simplify the process of the Deletion operations, instead of pulling characters from the next gram, we push the deleted results of *CurrentGram* to the front of the next gram, and process them in the next round. This process is different from the processes of Replacement and Insertion operations. Therefore, we consider cases of mutations without Deletion and with Deletion operations separately. (We will explain why this consideration will not result in any missing case later.)

For all cases of mutations stored in set *CombinAll* (as shown in Fig. 5-(a)), we separate them into two subsets: one is with $ND = 0$ and the other is with $ND \neq 0$, as shown in Fig. 5-(b) and Fig. 5-(c), respectively. These two subsets could be further simplified to two sets, *CombinRI* and *CombinD*, as shown in Fig. 5-(d) and Fig. 5-(e), respectively. Set *CombinRI* stores the cases with Replacement and Insertion operations, *i.e.*, Case 1 ($NI = NR = 0$), Case 2 ($NI = 0$, $1 \leq NR \leq MinKQ$), and Case 3 ($NI \geq 1$, $1 \leq NR + NI \leq MinKQ$). Set *CombinD* stores the case with Deletion operations, *i.e.*, Case 4 ($1 \leq ND \leq MinKQ$)).

3.4. Step 3

Now, we generate mutations of *CurrentGram* with Replacement and Insertion operations, and search their occurring positions from the EII structure to obtain a local answer. We apply procedure *ConsiderMutRI* shown in Fig. 7 to do this process. In procedure *ConsiderMutRI*, we first fetch one pair, (*NR*, *NI*), from *CombinRI*, and process it according to the conditions of *NR* and *NI*.

```
procedure ConsiderMutRI(CurrentGram, CombinRI, TempAns, RoundNum);
begin
    for each (NR,NI) ∈ CombinRI do
    begin

        if ((NI = 0) and (NR = 0)) then                    // Case 1
            NewTempAns := CheckAns(CurrentGram, TempAns, RoundNum);
        else if (NI = 0) then                               // Case 2
        begin
            ChooseSet := ChooseR(CurrentGram, NR);
            MutSet := ReplaceACGT(ChooseSet);
            for each MS ∈ MutSet do
            NewTempAns := NewTempAns_CheckAns(MS, TempAns, RoundNum);
        end
        else                                                // Case 3
        begin
            PreChars := PrefixChars(CurrentGram, $q − NI$);
            ChooseSet := ChooseRI(PreChars, NR, NI);
            MutSet := ReplaceACGT(ChooseSet);
            for each MS ∈ MutSet do
                NewTempAns := NewTempAns_CheckAns(MS, TempAns, RoundNum);
        end;

        NewQuerySeq := RemoveChars(QuerySeq, $q − NI$);  // remove the first ($q − NI$) characters
        NewK := $k − NR − NI$;
        if (NewQuerySeq = ““) then
            FinalAns := FinalAns_NewTempAns
        else // NewQuerySeq contains any character
            if (NewTempAns _= $\varphi$ ) then
               EOB(NewQuerySeq, NewK, NewTempAns, RoundNum + 1);
    end;
end;
```

Fig. 7: Procedure *ConsiderMutRI*

For Case 1 ($NI = NR = 0$), it considers the original gram, *i.e.*, *CurrentGram*. Therefore, we directly search the occurring positions of *CurrentGram* from the EII structure by calling function *CheckAns*, which is shown in Fig. 8. Function *CheckAns* not only searches the occurring positions of *CurrentGram*, but also checks whether these positions could meaningfully follow those positions stored in *TempAns*, where *TempAns* is the temporary global answer generated from the previous round. (Since we separate *QuerySeq* into many q-grams and find their postings lists independently, we need to ensure the correct global order of those matched substrings in the DNA sequence by applying such checks.) A position, *PL*, is said to meaningfully follow another position, *TA*, if $PL = TA + (RoundNum - 1) * q$, where *RoundNum* is the number of rounds processed so far. For example, assume that *QuerySeq* = "ACGTTT". The first gram, "ACG", has been found occurring at position 5 in the first round. Then, gram "TTT" is said to meaningfully follow "ACG", if "TTT" is found occurring at position $(5 + (2 - 1) * 3) = 8$ in the second round. In function *CheckAns*, if *RoundNum* is 1, *i.e.*, the first round, we directly return the postings list of *CurrentGram* as the function result. The reason is that *CurrentGram* is the first gram of the query sequence. The result of function *CheckAns* is stored in *NewTempAns*, *i.e.*, the new *TempAns* in the next round. After the first round, we will check whether the occurring positions of *CurrentGram* could meaningfully follow those positions in *TempAns*. For those positions in *TempAns* which are found to be followed, they are returned as the function result.

```
function CheckAns(CurrentGram, TempAns, RoundNum) : set;
begin
  PostList := GetPost(IA, CurrentGram);           // get the postings list of CurrentGram from IA
  if (RoundNum = 1) then CheckAns := PostList; // the first round
  for each TA ∈ TempAns do
     for each PL ∈ PostList do
        if ($PL = TA + (RoundNum - 1) * q$) then   // PL could follow TA
        begin
           append TA into PassedPos;
           break;
        end;
  CheckAns := PassedPos;
end;
```

Fig. 8: Function *CheckAns*

Case	NR	ChooseR*("ACG",* NR*)*
2-R1	*1*	**CG, A*G, AC**
2-R2	*2*	*A**, *C*, **G*
2-R3	*3*	***

*: the replaced character

Fig. 9: An example of the results of function *ChooseR*

ACA, ACC, ACG, ACT, AC$

Fig. 10: An example of the result of function *ReplaceACGT* for "AC*"

Next, we consider Case 2 (line 7 of Fig. 7), which is the case of mutations with Replacement operations only ($NI = 0$, $1 \leq NR \leq MinKQ$). To generate all possible mutations for this case, we choose *NR* characters from *CurrentGram* to do Replacement operations. The result of this choosing process, which is the same as that of C(q, *NR)* in combination mathematics, is stored in *ChooseSet*. Fig. 9 shows an example of the results of function *ChooseR* with NR = 1, 2, and 3 (*i.e.*, Cases 2-R1, 2-R2, and 2-R3 in Figure 5-(d)), where symbol "*" means the chosen characters. Then, we call function *ReplaceACGT* to replace each "*" in *ChooseSet* with any character of {A, C, G, T, $} to generate all possible mutations, and store them in *MutSet*. Fig. 10 shows an example of the result of function *ReplaceACGT*. (Note that we could utilize a simple technique to avoid generating a mutation the same as the original gram.) Finally, each mutation in *MutSet* is processed in the same way as the processing of *CurrentGram* in Case 1. That is, we call function *CheckAns* and determine *NewTempAns*.

For Case 3, we consider those mutations with Insertion operations and with/without Replacement operations ($NI \geq 1$, $1 \leq (NR + NI) \leq MinKQ$). In this case, if we insert *NI* characters to *CurrentGram*, we need to push the last *NI* characters of *CurrentGram* to the front of the next gram, to keep the length of these mutations equal to q, as shown in Fig. 11. Those pushed characters will be processed in the next round. That is, in Case 3, among the q characters of a mutation, *NI* characters are newly inserted characters, and ($q - NI$) characters are the original first ($q - NI$) characters of *CurrentGram*. Therefore, we first store the first ($q - NI$) characters of *CurrentGram* in *PreChars*. Then, function *ChooseRI* is applied to choose *NR* characters from *PreChars* to do Replacement operations, and choose *NI* positions from q positions to put the inserted characters. The number of possible mutations will be

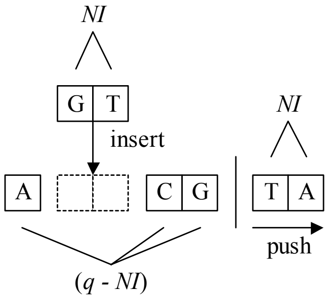


Fig. 11: The illustration for Insertion operations

Case	NR	NI	PreChars	ChooseRI*(*PreChars, NR, NI*)*
3-R0I1	*0*	*1*	*AC*	*%AC, A%C, AC%*
3-R1I1	*1*	*1*	*AC*	*%*C, %A*, *%C, A%*, *C%, A*%*
3-R2I1	*2*	*1*	*AC*	*%**, *%*, **%*
3-R0I2	*0*	*2*	*A*	*A%%, %A%, %%A*
3-R1I2	*1*	*2*	*A*	**%%, %*%, %%**
3-R0I3	*0*	*3*		*%%%*

*: the replaced character %: the inserted character

Fig. 12: The result of function *ChooseRI* for gram "ACG" with $0 \leq NR \leq 3$ and $1 \leq NI \leq 3$

(C(q – *NI*, *NR*) * *C*(q, *NI*)). Fig. 12 shows an example of all possible results of function *ChooseRI* for gram "ACG" with 0 ≤ *NR* ≤ 3 and 1 ≤ *NI* ≤ 3, where symbol "%" means the chosen positions for the inserted characters, and symbol "*" means the chosen characters to be replaced. (Note that each case is corresponding to one tuple of Case 3 shown in Fig. 5-(d).) In this example, for Case 3-R1I1, first, we have *PreChars* = "AC", since *NI* is 1. Then, function *ChooseRI* chooses 1 character from "AC" to be replaced, *i.e.*, "*C" and "A*", and inserts 1 character into any position of "*C" and "A*". That is, the possible results are "%*C", "*%C", "*C%", "%A*", "A%*", and "A*%", where "%" represents the inserted character. The number of possible results is (C(2, 1) * C(3, 1)) = 6. For those "*" and "%", they are replaced with {A, C, G, T, $} to generate all possible mutations. Finally, we call function *CheckAns* to check each mutation and determine *NewTempAns*.

Up to this point, we have introduced those cases without Deletion operations. Then, to recursively process the uncompleted part of the query sequence, *QuerySeq*, we need to update *QuerySeq* and the number of allowed errors, k. For *QuerySeq*, we remove its first (q − *NI*) characters, and store the result in *NewQuerySeq* (line 22 of Fig. 7). By doing such removal, the last *NI* characters of *CurrentGram* will be retained in (or "pushed" to) *NewQuerySeq*, as shown in Fig. 13. For k, we decrease its value by *NR* and *NI*. Then, if *NewQuerySeq* does not contain any character, it means that we have completely processed the query sequence, and those positions stored in *NewTempAns* are parts of the final answer (*i.e.*, the global answer). Otherwise, if there is at least one answer in *NewTempAns*, we recursively apply procedure *EOB* with new parameters *NewQuerySeq*, *NewK*, *NewTempAns*, and the new round number, (*RoundNum* + 1).

3.5. Step 4

In Step 4, we consider each of those *ND* Deletion operations stored in *CombinD*, *i.e.*, Case 4. We call procedure *ConsiderMutD* to process this case, which is shown in Fig. 14. In this case, we choose *ND* characters from *CurrentGram* to be deleted, and directly push the deleting results to form *NewQuerySeq*. That is, *NewQuerySeq* consists of two parts: (1) the deleting results of *CurrentGram*, and (2) the rest of *QuerySeq* after removing its first q-gram. Therefore, those pushed characters will be processed in the next round.

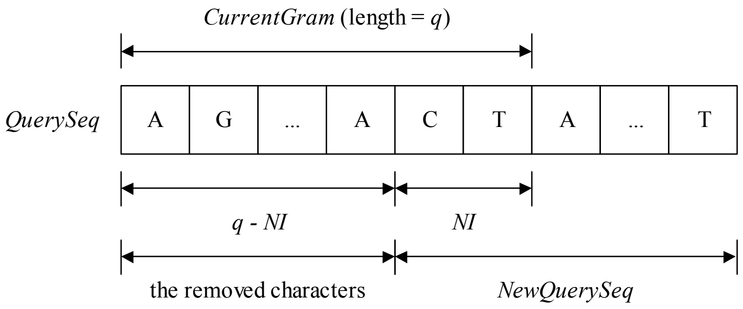


Fig. 13: The illustration of the number of removed characters

```
procedure ConsiderMutD(CurrentGram, CombinD, TempAns, RoundNum);
begin
  for each ND ∈ CombinD do                         // Case 4
  begin
    MutSet := ChooseD(CurrentGram, ND);
    for each MS ∈ MutSet do
    begin
      NewQuerySeq := MS + RemoveChars(QuerySeq, q); // concatenate MS and the rest of QuerySeq
      NewK := k −ND;
      EOB(NewQuerySeq, NewK, TempAns, RoundNum);
    end;
  end;
end;
```

Fig. 14: Procedure *ConsiderMutD*

DOI https://doi.org/2024-JICS-22764 | Generated on 2025-04-21 00:15:08

We apply function *ChooseD* to do the choosing process, where function *ChooseD* returns a set of possible deleting results with *ND* Deletion operations for *CurrentGram*. Fig. 15 shows an example of possible results with *ND* = 1, 2, and 3, *i.e.*, Cases 4-D1, 4-D2, 4-D3 in Fig. 5-(e), respectively. The number of possible results is C(*q*, *ND*). The results are stored in *MutSet*. Then, for each deleted result stored in MutSet, MS, we will combine it with the rest of *QuerySeq* after removing the first gram (line 8 of Fig. 14), to form the new query sequence, *NewQuerySeq*. Fig. 16 shows the illustration of Deletion operations. Assume that *QuerySeq* is "ACGTA" and *CurrentGram* is "ACG". When considering Case 4-D1, we combine the mutations of "ACG" with 1 Deletion operations, *i.e.*, "CG", "AG", and "AC", with the rest of *QuerySeq* after removing the first 3-gram, *i.e.*, "TA". Therefore, we have "CGTA", "ACTA", and "AGTA" as *NewQuerySeq*. After generating new query sequences, we decrease the value of *k* by *ND*. Finally, for each *NewQuerySeq*, we recursively apply procedure *EOB* with parameters *NewQuerySeq* (the new one), *NewK* (the new one), *TempAns*, and *RoundNum*.

Now, we explain the reason why we will not miss any case when we consider mutations with and without Deletion operations separately. Up to this point, we have discussed those cases of mutations with Replacement operations only (*i.e.*, Case 2), with Replacement and Insertion operations (*i.e.*, Case 3), and with Deletion operations (*i.e.*, Case 4). The other cases that we have not discussed yet are cases of mutations with the combinations of Deletion and any/both of Replacement and Insertion operations. We call such cases as the mixed cases. Fig. 17 shows the mixed cases for *MinKQ* = *k* = 3, where Fig. 17-(a) shows the original cases of mutations with Deletion operations (which are the same as Fig. 5-(c)), and Fig.

Case	NR	ChooseD*("ACG",* ND*)*
4-R1	*1*	*AC, AG, CG*
4-R2	*2*	*A, C, G*
4-R3	*3*	

Fig. 15: The results of function *ChooseD* for gram "ACG"

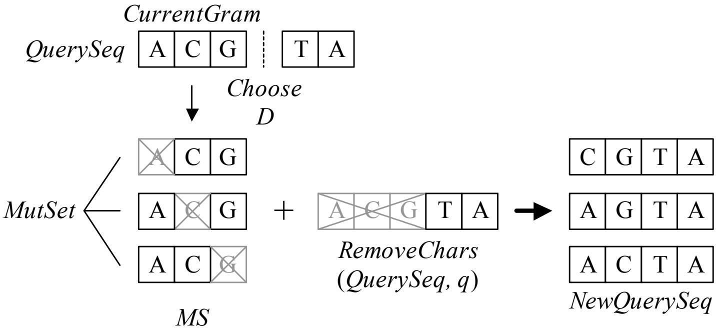


Fig. 16: The illustration of Deletion operations

	NR	*NI*	*ND*	Case
	0	0	1	4-D1
	0	0	2	4-D2
*	1	0	1	
*	0	1	1	
	0	0	3	4-D3
*	1	0	2	
*	2	0	1	
*	0	1	2	
*	0	2	1	
*	1	1	1	

(a)

		the current round		the next round	
NR	*NI*	*ND*	*NewK* (= *k* - *ND*)	*k*	Case
1	0	1	2	2	2-R1
0	1	1	2	2	3-R0I1
1	0	2	1	1	2-R1
2	0	1	2	2	2-R2
0	1	2	1	1	3-R0I1
0	2	1	2	2	3-R0I2
1	1	1	2	2	3-R1I1

(b)

Fig. 17: The mixed cases of mutations for *MinKQ* = *k* = 3: (a) the original cases; (b) the mixed cases and their equivalent cases in the next round after processing Deletion operations.

17-(b) shows the mixed cases and their equivalent cases in the next round after processing Deletion operations. In the EOB method, for each mixed case, there exists an equivalent case (Case 2 or Case 3) with the updated k in the next round after processing Case 4. That is, these mixed cases will be considered within two successive rounds: one round for considering Case 4 and updating k, and the next round for considering Cases 2 or 3 with updated k, as shown in Figure 17-(b). For example, when we consider the case of $NR = NI = ND = 1$, it is considered as follows: in the first round, we consider Case 4-D1, and update k ($= 3 - 1 = 2$); in the next round, we consider Case 3-R1I1 ($NR = NI = 1$ in Case 3) with the updated k ($= 2$). Therefore, we also consider the mixed cases, and do not miss any case.

Case	*Name*	*Description*	*Length (base pairs)*
1	*NT_035113*	*Homo sapiens chromosome 11 genomic contig*	*1102759*
2	*NT_113796*	*Homo sapiens chromosome 1 genomic contig*	*201709*

Fig. 18: Real DNA sequences used in our experiments

4. Performance

In this section, we study the performance of the proposed EOB method for approximate string matching by experiments on real biological data. We will make a comparison with the ($k + s$) q-samples filter [16][21][22][23] in terms of the processing time.

4.1. Experiment Environment

In our performance study, we use the real DNA sequences downloaded from NCBI (http://www.ncbi.nlm.nih.gov) as our experiment data. Fig. 18 shows the details of these chosen DNA sequences. The DNA sequence of Case 1 is a long sequence, which contains over one million base pairs. The DNA sequence of Case 2 is a short sequence, which contains two hundred thousand base pairs. In the experiment on approximate string matching, the input data is a query sequence and the number of allowed errors, k, and the output result will be the matching positions for this query in the DNA sequence. We will compare the processing time of the EOB method with the total time of processing the ($k + s$) q-samples filter to find candidates and processing dynamic programming to verify those candidates. (Note that the time for building the EII structure in our approach is very short. For the long DNA sequence of Case 1, it only needs about 500 milliseconds to read from the file and build the index. Therefore, we only measure the execution time of both methods after the DNA sequence has been stored in the memory.) The parameter, s, which is used in the ($k + s$) q-samples filter, is set to 2, as mentioned in [21]. The parameter, q, which is used to build the proposed EII structure, is set to 6 here. (We will discuss the effect on the processing time with different values of q later.) Our experiments are performed on a Celeron machine with one CPU clock rate of 2.66 GHz, 736 MB of main memory, running Windows XP Professional with SP2 version, and coded in Java.

4.2. Experiment Results on Approximate String Matching

In this subsection, we show the experiment results on different conditions. These conditions include varied error levels, varied lengths of query sequences, and varied lengths of DNA sequences.

First, we use the DNA sequence of Case 1 to compare the performance of the EOB method with that of the ($k + s$) q-samples filter, based on different *error levels*. The error level, e, is defined as $e = k / m$, where k is the number of allowed errors and m is the length of the query sequence. Most of the filter methods will lose their filtration power at error levels over 30% [17]. Furthermore, the optimal error level for the filter methods seems to be between 0% and 20% [21]. Therefore, this experiment is based on the error levels from 0% to 20%. The query sequence, whose length is 20, is randomly generated from the DNA sequence of Case 1, and we randomly add k ($= e * m$) errors to this query sequence. That is, we randomly choose k positions and apply one of Replacement, Insertion, or Deletion operations to each chosen position. Fig. 19 shows the comparison of the processing time between the EOB method and the ($k + s$) q-samples filter. From this figure, we observe that the EOB method needs less processing time than the ($k + s$) q-samples filter. This is because the ($k + s$) q-samples filter generates many candidates which are false positives, and these candidates also need to be verified by dynamic programming. In the EOB method, we do not find candidates and verify them. Those found positions in the EOB method are the final answers for the query. Therefore, the EOB method outperforms the ($k + s$) q-samples filter in terms of the processing time.

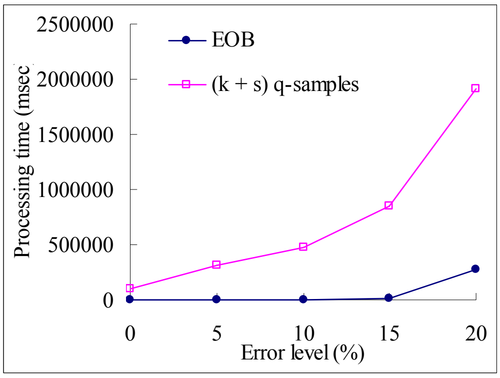


Fig. 19: A comparison of the processing time based on different error levels (Case 1)

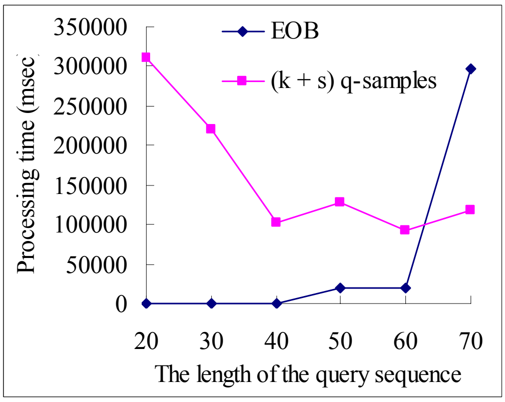


Fig. 20: A comparison of the processing time based on different lengths of the query sequence (Case 1)

Second, we use the DNA sequence of Case 1 to compare the performance of the EOB method with that of the $(k + s)$ q-samples filter, based on different lengths of the query sequence. The error level is set to 5% here, and query sequences are randomly generated from the DNA sequence with different lengths. Fig. 20 shows the comparison of the processing time between the $(k + s)$ q-samples filter and the EOB method. From this figure, we observe that the EOB method needs less processing time than the $(k + s)$ q-samples filter, when the length of the query sequence is less than 70. When the length of the query sequence is larger than 70, the proposed method needs to consider too many mutations, which results in the decrease of the performance. Therefore, from this figure, we show that the proposed method is more suitable than the $(k + s)$ q-samples filter for short query sequences, while the $(k + s)$ q-samples filter is more suitable for long query sequences.

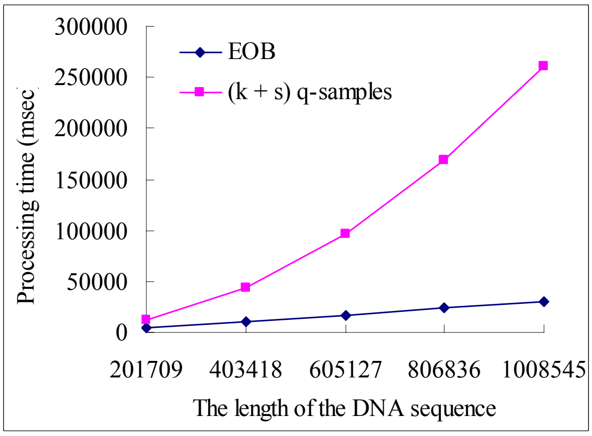


Fig. 21: A comparison of the processing time based on different lengths of DNA sequences (Case 2)

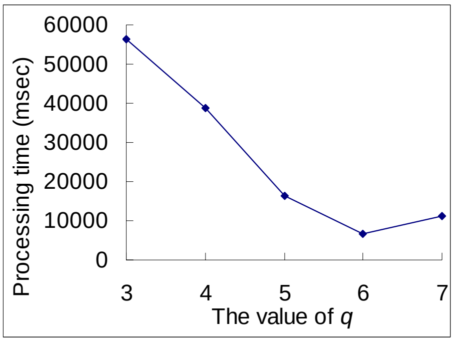


Fig. 22: The processing time of the EOB algorithm based on different values of q

Third, we use the DNA sequence of Case 2 to compare the performance of the EOB method with that of the $(k + s)$ q-samples filter, based on different lengths of DNA sequences. We expand the DNA sequence of Case 2 to double size, triple size, and so on, by repeatedly appending this DNA sequence to itself. The error level is set to 10% here, and the query sequence (whose length is 30) is randomly generated from the DNA sequence. Fig. 21 shows the comparison of the processing time between the $(k + s)$ q-samples filter and the EOB method. From this figure, we show that the EOB method could provide better performance than the $(k + s)$ q-samples filter in terms of the processing time, no matter what length the DNA sequence is. This is because as the length of the DNA sequence increases, the number of candidates in the $(k + s)$ q-samples filter also increases. Therefore, the $(k + s)$ q-samples filter needs to spend much time to verify these candidates. Obviously, as the length of the DNA sequence increases, the processing time of the EOB method increases linearly. This is because we build the EII structure in advance and utilize it to answer the query. This index structure is very helpful, when the length of the DNA sequence is long. Therefore, the EOB method could provide better performance than the $(k + s)$ q-samples filter in terms of the processing time, no matter what length the DNA sequence is.

4.3. Experiment on Different Values of *q*

In this subsection, we discuss the effect on the processing time with different values of q, where q is the parameter which determines the length of each gram for building the EII structure. We use the DNA sequence of Case 2 as the test database. The error level is set to 5% here, and the query sequence (whose length is 60) is randomly generated from the DNA sequence. Fig. 22 shows the processing time of the EOB method based on different values of q. We observe that as the value of q increases, the processing time decreases. This is because in the EOB method, for a query sequence with length m, we need to recursively process it by about (m / q) rounds. The larger the value of q is, the less the processing rounds we need. However, in each round, as the value of q increases, the number of mutations that we need to consider also increases. Therefore, we could observe that the processing time for $q = 7$ is longer than the processing time for $q = 6$. Moreover, Fig. 23 shows the size of the EII structure based on different values of q. As the value of q increases, the size of the EII structure increases, since the size is equal to 5^q. If the value of q is too large, it may result in running out of memory. Therefore, in those previous experiments, the value of q is set to 6, which is acceptable in both of the processing time and the size of the used memory space.

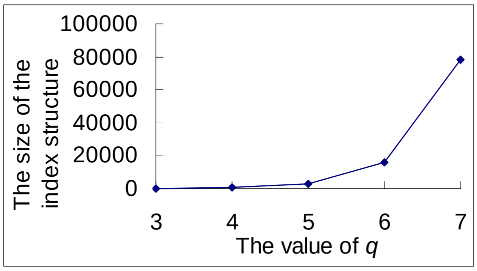


Fig. 23: The size of the EII structure based on different values of q

5. Conclusion

Searching patterns for approximate string matching in DNA databases is a crucial step in the DNA related research. In this paper, we have proposed a new index structure and a new method for approximate string matching. First, we have proposed the EII index structure, which makes use of the mapping technique to map each *q*-gram into an index key, and records the occurring positions of this *q*-gram in the corresponding index entry. Therefore, we could efficiently find the occurring positions of any q-gram by its index key. Moreover, the EII index structure avoids missing cases which may occur in the inverted index. Next, based on the EII structure, we have proposed the EOB method to answer the queries for approximate string matching. Different from the filter methods, we utilize the pre-built EII structure to help us check local order and global order of the related mutations for a query sequence, and find the answers directly without applying dynamic programming. From the experiment results, we have shown that our method could outperform the $(k + s)$ *q*-samples filter in terms of the processing time for short query sequences. How to extend the proposed method to parallel processing is the possible future work.

6. Acknowledgements

This research was supported in part by the National Science Council of Republic of China under Grant No. NSC-95-2221-E-110-079-MY2. The authors also like to thank “Aim for Top University Plan” project of NSYSU and Ministry of Education, Taiwan, for partially supporting the research.

7. References

[1] S. Altschul, W. Gish, W. Miller, E. Myers, and D. Lipman. Basic Local Alignment Search Tool. *Journal of Molecular Biology*. 1990, **215**(3): 403-410.

[2] R. Baeza-Yates and G. Gonnet. A New Approach to Text Searching. *Communications of the ACM* . 1992, **35**(10): 74-82.

[3] W. Chang and E. Lawler. Sublinear Approximate String Matching and Biological Applications. *Algorithmica.* 1994, **12**(4): 327-344.

[4] W. Chang and T. Marr. Approximate String Matching and Local Similarity. *Proc. of the 5th Annual Symp. on Combinatorial Pattern Matching*. pp. 259-273. 1994.

[5] E. C. Friedberg, G. C. Walker, and W. Siede. *DNA Repair and Mutagenesis. American Society Microbiology*, 1995.

[6] L. Gravano, P. Ipeirotis, H. Jagadish, N. Koudas, S. Muthukrishnan, L. Pietarinen, and D. Srivastava. Using q-Grams in a DBMS for Approximate String Processing. *IEEE Data Engineering Bulletin*. 2001, **24**(4): 28-34.

[7] J. L. Houle, W. Cadigan, S. Henry, A. Pinnamaneni, and S. Lundahl. Database Mining in the Human Genome Initiative. http://www.biodatabases.com/whitepaper01.html.

[8] H. Hyyro, Y. Pinzon, and A. Shinohara. Fast Bit-Vector Algorithms for Approximate String Matching Under Indel Distance. *Proc. of the 31st Annual Conf. on Current Trends in Theory and Practice of Informatics*. pp. 380-384, 2005.

[9] J. Karkkainen and J. C. Na. Faster Filters for Approximate String Matching. *Proc. of Workshop on Algorithm Engineering and Experiments*. pp. 1-7, 2007.

[10] M. S. Kim, K. Y. Whang, J. G. Lee, and M. J. Lee. n-Gram/2L: A Space and Time Efficient Two-Level n-Gram Inverted Index Structure. *Proc. of the 31st Int. Conf. on Very Large Databases*. pp. 325-336, 2005.

[11] W. C. Kim, S. Park, J. I. Won, S. W. Kim, and J. H. Yoon. An Efficient DNA Sequence Searching Method Using Position Specific Weighting Scheme. *Journal of Information Science.* 2006, **32**(2): 176-190.

[12] D. J. Lipman and W. R. Pearson. Rapid and Sensitive Protein Similarity Searches. *Science*. 1985, **227**(4693): 1435-1441.

[13] B. Ma, J. Tromp, and M. Li. PatternHunter: Faster and More Sensitive Homology Search. *Bioinformatics*. 2002, **8**(3): 440-445.

[14] A. Mazeika, M. H. Bohlen, N. Koudas, and D. Srivastava. Estimating the Selectivity of Approximate String Queries. *ACM Trans. on Database Systems*. 2007, **32**(2): 1-40.

[15] G. Navarro. Multiple Approximate String Matching by Counting. *Proc. of the 4th South American Workshop on String Processing*. pp. 95-111, 1997.

[16] G. Navarro, E. Sutinen, J. Tanninen, and J. Tarhio. Indexing Text with Approximate q-grams, *Proc. of the 11th Annual Symp. on Combinatorial Pattern Matching*. pp. 350-363, 2000.

[17] G. Navarro. A Guided Tour to Approximate String Matching. *ACM Computing Surveys*. 2001, **33**(1): 31-88.

[18] Z. Ning, A. J. Cox, and J. C. Mullikin. SSAHA: A Fast Search Method for Large DNA Databases. *Genome*

Research. 2001, **11**(10): 1725-1729.

[19] K. R. Rasmussen, J. Stoye, and E. W. Myers. Efficient q-Gram Filters for Finding All ε-Matches over a Given Length. *Journal of Computational Biology*. 2006, **13**(2): 296-308.

[20] T. F. Smith and M. S. Waterman. Identification of Common Molecular Subsequences. *Journal of Molecular Biolog.* 1995, **147**(1): 195-197.

[21] E. Sutinen and J. Tarhio, On Using q-Gram Locations in Approximate String Matching. *Proc. of the 3th Annual European Symp. on Algorithms*. pp. 327-340, 1995.

[22] E. Sutinen and J. Tarhio. Filtration with q-Samples in Approximate String Matching. *Proc. of the 7th Annual Symp. on Combinatorial Pattern Matching*. pp. 50-63, 1996.

[23] E. Sutinen and J. Tarhio. Approximate String Matching with Ordered q-Grams. *Nordic Journal of Computing*. 2004, **11**(4): 321-343.

[24] T. Takaoka. Approximate Pattern Matching with Samples. *Proc. of the 5th Int. Symp. on Algorithms and Computation*. pp. 234-242, 1994.

[25] E. Ukkonen. Approximate String Matching with q-Grams and Maximal Matches. *Theoretical Computer Science* 1992, **92**(1): 191-211.

[26] M. S. Waterman. Sequence Alignments in the Neighborhood of the Optimum with General Application to Dynamic Programming. *Proc. of National Academy of Sciences*. pp. 3123-3124, 1983.

[27] H. E. Williams and J. Zobel. Indexing Nucleotide Databases for Fast Query Evaluation. *Proc. of Int. Conf. on Extending Database Technology*. pp. 275-288, 1996.

[28] H. E. Williams and J. Zobel. Indexing and Retrieval for Genomic Databases. *IEEE Trans. on Knowledge and Data Eng*. 2002, **14**(1): pp. 63-78.