A Hash Trie Filter Method for Approximate String Matching in Genomic Databases

Ye-In Chang, Jiun-Rung Chen, and Min-Tze Hsu

Dept. of Computer Science and Engineering
National Sun Yat-Sen University
Kaohsiung, Taiwan
Republic of China
{E-mail: changyi@cse.nsysu.edu.tw}

Abstract

In genomic databases, approximate string matching with k errors is often applied when searching genomic sequences, where k errors can be caused by substitution, insertion, or deletion operations. In this paper, we propose a new method, the *hash trie filter*, to efficiently support approximate string matching in genomic databases. First, we build a hash trie for indexing the genomic sequence stored in a database in advance. Then, we utilize an efficient technique to find the ordered subpatterns in the sequence, which could reduce the number of candidates by pruning some unreasonable matching positions. Moreover, our method will dynamically decide the number of ordered matching grams, resulting in the increase of precision. The simulation results show that the hash trie filter outperforms the well-known $(k + s)$ q-samples filter in terms of the response time, the number of verified candidates, and the precision, under different lengths of the query patterns and different error levels.

(*Keywords*: Approximate string matching, filter, genomic database, global order, local order)

1 Introduction

Bioinformatics has received increased publicity over the past few years, in large part due to its importance to the Human Genome Project. With the increment of genomic sequences, numerous large databases hold these DNA and protein sequences [8]. DNA sequences hold the code of life for every living organism. The primary structure of DNA is represented as strings, which are composed of four basic molecules called *nucleotides*. Each nucleotide contains phosphate, sugar, and one of the following four bases: *Adenine* (A), *Guanine* (G), *Cytosine* (C), and *Thymine* (T) [9, 10, 28]. DNA sequences could be very long. For example, the human genome contains about 3×10^9 base pairs (bps). Genomic sequence databases, like GenBank, EMBL, are widely used by molecular biologists for homology searching. Because of the long length of each genomic sequence and the increase of the number of genomic sequences, the importance of efficient searching methods for answering queries grows [13].

String matching methods are usually applied when searching in the databases. The domain of string matching could be separated into two parts, *exact matching* and *approximate string matching* [11]. Given a text sequence, exact matching is to find the positions of all its substrings which exactly match the query pattern. Approximate string matching (*ASM*) is to find the positions of all its substrings which differ from the query pattern in at most k errors [5, 16].

In the DNA related research, *mutations* occur very often, where a mutation is defined as a heritable change in the DNA sequence, caused by a faulty replication process [7]. These errors in replication occur often due to exposure to ultra violet radiation or other environment conditions. There are several kinds of point mutations [19]: (1) Substitution: a change of one nucleotide in a DNA sequence, for example, AC$\underline{\text{G}}$T and AC$\underline{\text{C}}$T; (2) Insertion: an addition of one or more nucleotides to a DNA sequence, for example, ACGT and ACG$\underline{\text{A}}$T; (3) Deletion: a removal of one or more nucleotides from a DNA sequence, for example, AC$\underline{\text{G}}$T and ACT. The problem of finding these mutations could be solved by ASM, since a DNA sequence could be thought as a traditional string composed of characters {A, C, G, T} in the string matching problem.

For methods of ASM, FASTA [14] and BLAST [1] are two principal heuristic methods.

They compute suboptimal pairwise similarity comparisons [8]. Although they are very efficient, they may not find the optimal answers, and may need large amounts of memory [15]. For methods of ASM which aim to find the optimal answers, they can be classified into four main approaches [19]: dynamic programming, automaton, bit-parallelism, and filters.

The most basic category is dynamic programming [21]. 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, for DNA sequences, the length of a sequence may be up to several millions. Computing all values of such a big array is almost impractical [20].

The second category is the automaton approach [26]. This approach models the search with a nondeterministic automaton (NFA). However, for this approach, there is a time and space exponential dependence on the length of the query sequence and the number of allowed errors, which limits its practicality [19].

The third category is the bit-parallelism approach [17]. This approach is based on exploiting the parallelism of the computer when it works on bits [19]. The basic idea is to "parallelize" another approach, *e.g.*, dynamic programming or automaton, by using bits. This approach could cut down the number of needed operations when performing methods by a factor of at most w, where w is the number of bits in a computer word, *i.e.*, $w = 32$ or $w = 64$ in current computer architectures [19].

The fourth category is the filter approach. Methods based on this approach are currently very active for ASM [3, 4, 18, 22, 25, 27]. They are based on finding fast methods to discard large areas of the text that cannot match, and applying another method to verify the candidates, *e.g.*, the simple dynamic programming method. Filter methods address only the average case, and their major interest is the potential of methods not to inspect all text characters. The major theoretical achievement is a method with the average cost $O(n(k + \log_\sigma m)/m)$, which was proven optimal, where n is the length of a text sequence, k is the number of allowed errors, and m is the length of a query pattern. In practice, filter methods are the fastest ones [6, 12].

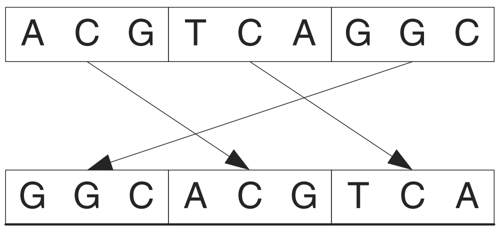


Figure 1: An example of different global order of grams between two sequences

Most of filter methods deal with q-grams of the pattern, *i.e.*, continuous substrings with length q. The idea is that whenever an approximate match occurs, it has to resemble the original pattern. This resemble is reflected by the existence of the same q-grams both in the pattern and in its approximate match [22]. According to the consideration of the local/global order of q-grams of the pattern, filter methods could be classified into two types: one ensures the local order of characters within a gram or subpattern, and the other one ensures the global order of grams or subpatterns within a sequence. Figure 1 shows the concept of global order. Most of those previous filter methods consider only the local order. So far, only the series of the $(k + s)$ q-samples filter [20, 22, 23, 24] considers both the local order and the global order, which could improve the precision of filtration.

Although the $(k + s)$ q-samples filter considers both local and global order to improve the precision of filtration, it still has some disadvantages. The $(k + s)$ q-samples filter cannot handle the situations when the length of a query pattern is short. Moreover, when the length of a query pattern is too long, it will lead to a large value of h according to its formula, where h is the distance of sampling. However, the length of a sample is fixed to $q = \log_4 |P|$, which would neglect the non-sampled area within h. Figure 2 shows an example of this situation, where only the sampled characters, *e.g.*, $h_1 =$"CTA" in this example, appear in piece Q_1 of the query pattern. The other characters before h_1 within area h of text X, *i.e.*, "ACTGTC", do not appear in piece Q_1. Furthermore, in the $(k+s)$ q-samples filter, to be a candidate for verification, the least number of ordered subpatterns, s, seems to be fixed to 2. In fact, it may not be strict enough to decide a candidate. Let $X[a \ldots b]$ mean the substring of text X from position a to position b. In Figure 2, there are two ordered subpatterns, "CTA" and "TCC", occurring in the text, where k (the number of allowed errors) is 1, s (the least number of ordered subpatterns) is 2, and $q = 3$. That is,

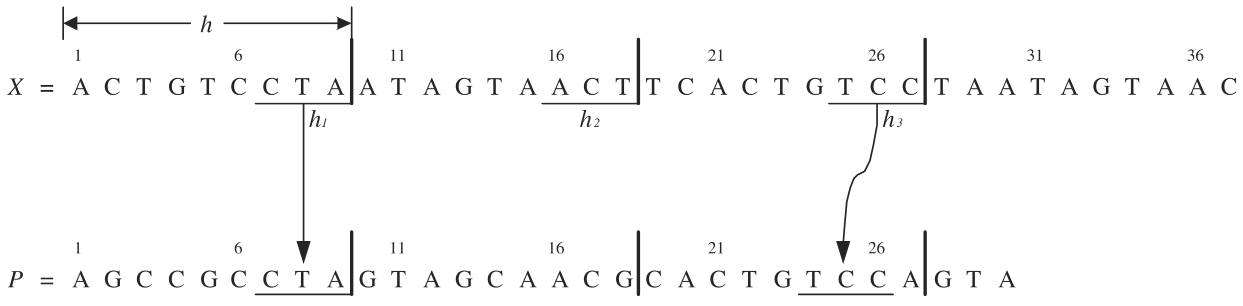

h
1
6
11
16
21
26
31
36
X = A C T G T C C T A A T A G T A A C T T C A C T G T C C T A A T A G T A A C
h1
h2
h3
1
6
11
16
21
26
P = A G C C G C C T A G T A G C A A C G C A C T G T C C A G T A

Figure 2: A bad case of the $(k + s)$ q-samples filter

the number of matched grams, 2, is equal to the value of s. Therefore, the $(k+s)$ q-samples filter will pass $X[1\ldots37]$ as a candidate to be verified. However, it is obvious that query pattern P can not be approximately matched with any substring of text X within $1(=k)$ edit operation.

To keep the advantage of the $(k + s)$ q-samples filter, *i.e.*, checking the global order, and improve its disadvantages, in this paper, we propose a new method, the *hash trie filter*. In our method, first, we construct a hash trie for indexing text X in advance. Next, we propose an efficient method to find the occurring positions of subpatterns of one query pattern in text X. To improve the precision, our method not only ensures that the global order of passed subpatterns in a candidate is correct, but also dynamically decides the value of s according to the length of the query pattern (m) and the number of allowed errors (k). From the simulation study, we show that the response time of the hash trie filter is shorter than that of the $(k + s)$ q-samples filter. The number of processing candidates of the hash trie filter is also fewer than that of the $(k + s)$ q-samples filter. Furthermore, the precision of the hash trie filter is higher than that of the $(k + s)$ q-samples filter.

The rest of the paper is organized as follows. Section 2 gives a survey of several filter methods. Section 3 presents the proposed hash trie index structure and the method for approximate string matching. In Section 4, we study the performance of the hash trie filter and compare it with the $(k + s)$ q-samples filter. Finally, Section 5 states the conclusion.

2 Filter Methods for ASM

The idea of filter methods is to filter the text and quickly discard the text areas that are not matched. In practice, filtering methods are the fastest ones. All of them, however, are limited in their applicability by the error level. They also need a non-filter method to check the potential matches [19]. In this subsection, we describe three well-known filter methods, including the counting filter [18], the q-gram filter [27], and the $(k+s)$ q-samples filter [20, 22, 23, 24].

2.1 The Counting Filter

The idea of the counting filter [18] is based on the simple fact that inside any match with at most k errors, there must be at least $(m-k)$ characters belonging to the pattern. The filter does not care about the order of these characters, *i.e.*, the local order.

The search method first builds a counting table for the query pattern of length m, to count the number of occurrences of each character in this pattern. Then, it slides a window of length m over the text, and keeps counting the number of window characters that belong to the pattern. This is easily done with a table where for each character W, the table stores a counter of W in the pattern. The counter is increased when a W enters the window, and is decreased when it leaves the window. Then, we evaluate the difference value of two counters of each character between the counting tables of the query pattern and the current window. If the difference value is not larger than k, the current window is verified with P by dynamic programming.

Figure 3 shows the counting table of query $P =$ "GTTAGC". Figure 4 shows the counting tables of windows in text $T =$"AACGGTCCCTTAGGC". Figure 5 shows the process of the counting filter. It passes the substrings $X[1,8]$, $X[2,9]$, $X[8,15]$, $X[9,15]$, $X[10,15]$ to be verified, where $X[a,b]$ means the substring of X with the index from a to b. Since this filter only cares about the counting number of each character in a window, it considers neither local order nor global order.

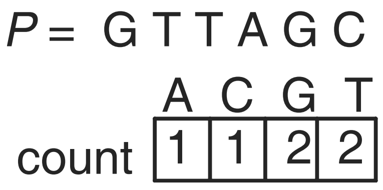


Figure 3: The counting table of query $P =$“GTTAGC”

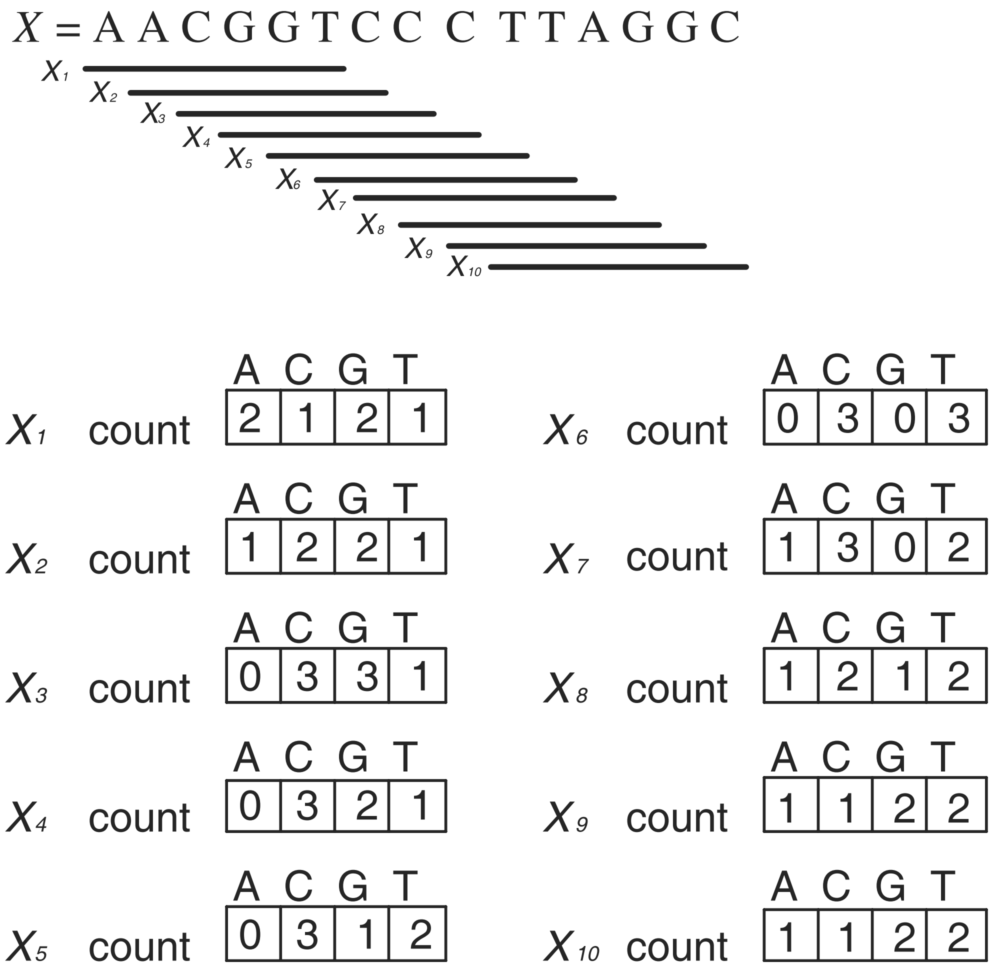


Figure 4: The counting tables of windows in text $X =$ “AACGGTCCCTTAGGC”

		A	C	G	T			A	C	G	T		
X_1	count	2	1	2	1	-	P count	1	1	2	2	= 1+0+0+1=2	<= k = 2 Pass X[1,8] to be verified!!
X_2	count	1	2	2	1	-	P count	1	1	2	2	= 0+1+0+1=2	<= k = 2 Pass X[2,9] to be verified!!
X_3	count	0	3	3	1	-	P count	1	1	2	2	= 1+2+1+1=4	> k = 2 Filter!!
X_4	count	0	3	2	1	-	P count	1	1	2	2	= 1+2+0+1=4	> k = 2 Filter!!
X_5	count	0	3	1	2	-	P count	1	1	2	2	= 1+2+1+0=4	> k = 2 Filter!!
X_6	count	0	3	0	3	-	P count	1	1	2	2	= 1+2+2+1=6	> k = 2 Filter!!
X_7	count	1	3	0	2	-	P count	1	1	2	2	= 0+2+2+0=4	> k = 2 Filter!!
X_8	count	1	2	1	2	-	P count	1	1	2	2	= 0+1+1+0=2	<= k = 2 Pass X[8,15] to be verified!!
X_9	count	1	1	2	2	-	P count	1	1	2	2	= 0+0+0+0=0	<= k = 2 Pass X[9,15] to be verified!!
X_{10}	count	1	1	2	2	-	P count	1	1	2	2	= 0+0+0+0=0	<= k = 2 Pass X[10,15] to be verified!!

Figure 5: The process of the counting filter with text X and query P

2.2 The q-gram Filter

Ukkonen proposed a filter method which references to "q-grams" for searching [27]. A q-gram is a substring of length q. A filter was proposed based on counting the number of q-grams shared between the pattern and a text window. A pattern of length m has $(m-q+1)$ overlapping q-grams. Each error can alter q q-grams, and therefore, $(m - q + 1 - kq)$ q-grams must appear in any occurrence. Figure 6 illustrates an example of one passed text window. With the formula mentioned above, we calculate the number of q-grams of the window of text X which must appear in query P (or *vice versa*). Since $(m - q + 1 - kq) = (8 - 2 + 1 - 2 * 2) = 3$, it means that to be a candidate, three q-grams of the window of text X must appear in query P. In this example, there are three 2-grams matched. Therefore, we can pass substring $X[1, 10]$ to be verified with P by dynamic programming. This filter considers the local order of characters in a gram, while it does not consider the global order of grams in a window.

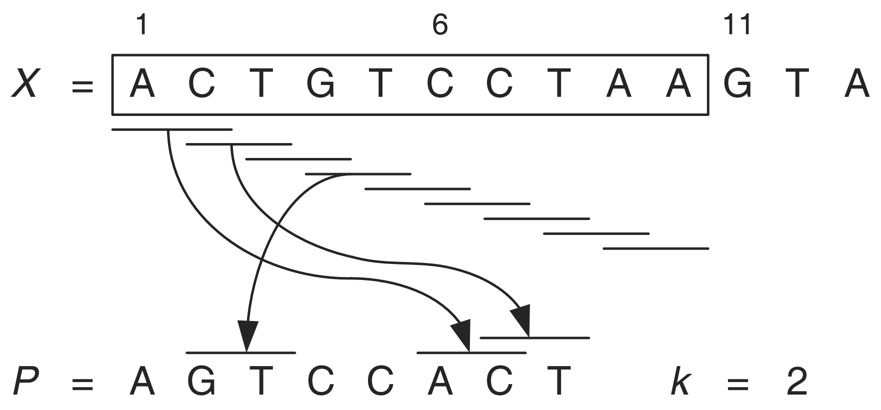


Figure 6: The q-gram filter

2.3 The $(k + s)$ q-Samples Filter

Sutinen and Tarhio proposed several filter methods for approximate string matching [20, 22, 23, 24]. 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 [25], 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 (*i.e.*, global order), and must not be more than k characters away from their correct positions; otherwise, the number of errors will be larger than k.

In this case, the sampling step is reduced to $h = \lfloor (m - k - q + 1)/(k + s) \rfloor$, where m is the length of the query pattern. The reason for this reduction is that to ensure s pieces being 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 one single q-sample, this filter considers the text windows of consecutive sequences of $(k+s)$ q-samples. Each of these q-samples is searched in the corresponding set. If there exist at least s of these q-samples matched, the area of $X[((j-1)-(k+s)h-2k-q+2)\ldots((j-1)+m-(k+s-1)h+k-q)]$ will be verified with dynamic programming, where j is the end position of the last q-sample. To efficiently compute the sum of matches for the $(k+s)$ consecutive samples, this filter utilizes an array to implement the Shift_Add approach [2].

Take Figure 2 as an example. X is the text sequence and P is the query pattern. The length of P, *i.e.*, m, is 30. Assume $k = 1$ (the number of allowed errors), $q = 3$ (the length of a subpattern), and $s = 2$ (the least number of matched subpatterns). Therefore, the sampling distance, h, is evaluated by $h = \lfloor (m - k - q + 1)/(k + s) \rfloor = \lfloor 27/3 \rfloor = 9$. That is, for text X, we pick one q-sample every 9 characters. If at least s $(= 2)$ of q-samples of text X match q-grams of query P in order, this area of text X is verified with query P by dynamic programming. Therefore, this filter considers both the local order of characters in a gram and the global order of grams within one area.

Note that for the $(k + s)$ q-samples filter, s seems to be set to a fixed value, 2. In the part of concluding remarks of [22], the authors mentioned that they had tried simulating their method for $s > 2$. Although the efficiency of their method increases as the value of s increases, the efficiency of the Shift_Add approach decreases, which limits the improvement of performance of this filter. Therefore, in fact, the authors seemed to focus on only $(k+2)$ consecutive samples, *i.e.*, $s = 2$.

3 A Hash Trie Filter Approach for ASM

In this section, first, we present a data structure, the hash trie, for efficiently find the occurring positions of q-grams split from sequences of a genomic database. Next, we present a new query processing method for ASM.

3.1 The Construction of a Hash Trie

In this section, we describe how to construct the hash trie for the given text, X. A *trie* is a tree with the following properties: (1) each edge contains a symbol of the alphabet; (2) no two child edges of one node contain the same symbol. The length of a gram in the hash trie, q, is determined by the formula $q = \lceil \log_\sigma n \rceil$ [22], where σ is the size of the alphabet and n is the length of X. Note that we let $\sigma = 4$ for the DNA sequences, while we will let $\sigma = 20$ for the proteomic sequences. Based on this formula, the size of a hash trie is proportional to the logarithm of the length of X. Then, we divide the overlapped

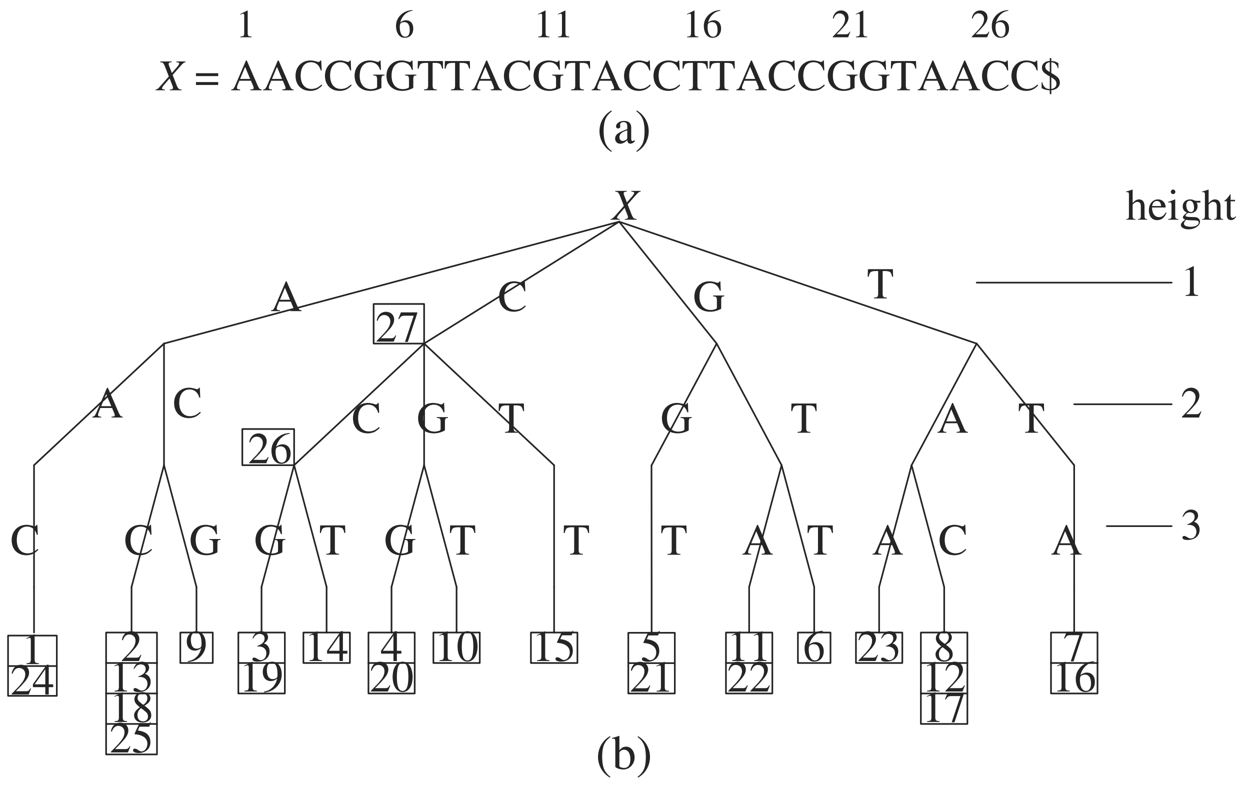


Figure 7: The hash trie with $q = 3$ in our example: (a) the given text, X; (b) the related hash trie

q-grams from the genomic sequence. For example, for $X =$"AACC" and $q = 3$, we have two overlapped grams, "AAC" and "ACC". Note that the last y characters of X (*e.g.*, "CC" and "C" in the previous example) will also be split, where $1 \leq y \leq (q - 1)$. The reason is that we will adjust the length of a gram according to the length of the query pattern, m, and the number of allowed errors, k, when a query is inquired. In order not to miss any information, we need to maintain the positions of grams occurring in the end of X, where lengths of these grams are all smaller than q. For the given text, X, we will traverse its grams in the hash trie (HT) one character by one character, and store the occurring positions of each gram under the traversing path. For example, for 3-grams (*i.e.*, $q = 3$) split from text X shown in 7-(a), Figure 7-(b) shows the resulting hash trie.

The breadth of a node of the hash trie is $\sigma = 4$, since the alphabet is $\Sigma = \{$A,C,G,T$\}$. The height of hash trie is q, because the length of a gram is q. Therefore, the hash trie has σ^q leave nodes. Initially, without storing any position, the hash trie is a complete trie.

```
Procedure $Search(P, HT)$;
begin
    $m$ := the length of $P$;
    $q'$ := $\lceil \log_\sigma m \rceil$;                    /* Step 1 */
    $NumSP$ := $\lfloor (m/q') \rfloor$;
    $s$ := $NumSP - k$;
    while ($s < w$) {
        $q'$ := $q' - 1$;
        $NumSP$ := $\lfloor (m/q') \rfloor$;
        $s$ := $NumSP - k$;
    }
    $lastLen$ := $m$ mod $q'$;
    $ConstructPT(q', P, PT)$;                  /* Step 2 */
   $PruneStartPosition(PT, PrunedSPT)$;/*Step 3*/
    $ComputeEndPosition(PrunedSPT, EndPT)$;
    $PruneEndPosition(EndPT, PrunedEPT)$;
    $MapToRM(PrunedEPT, RM)$;        /* Step 4 */
    $Find\_s\_WithSW(RM, ListOfDPRange)$;
    $VerifyWithDP(ListOfDPRange)$; /* Step 5 */
end;
```

Figure 8: Procedure $Search(P, HT)$

3.2 Query and Search

In the previous subsection, we construct a hash trie (HT) for the given text sequence, X. Then, when one user inquires a query pattern, P, we apply procedure $Search(P, HT)$ shown in Figure 8 for searching P. The related parameters are shown in Table 1.

3.2.1 Step 1: Deciding Length q' of a Subpattern

When a query is inquired, we decide the length of a subpattern, q', for this query pattern, according to its length, m, and the number of allowed errors, k. (Note that the value of q' used in this step can be different from the value of q used in the construction of the hash trie.) In the $(k + s)$ q-samples filter [20, 22, 23, 24], the length of q' is decided by formula $q' = \lceil \log_\sigma m \rceil$, where σ is the size of the alphabet (*i.e.*, 4), and m is the size of query pattern P. However, q' is restricted by formulas $h = \lfloor (m-k-q+1)/(k+s) \rfloor$ and $k/m < 1/\log_\sigma m$. Furthermore, the value of s seems to be fixed to 2 in the $(k+s)$ q-samples filter. Therefore,

Table 1: The meaning of parameters used in procedure *Search*

parameter	description
q'	the length of a subpattern for the given query pattern, P
$NumSP$	the number of subpatterns
s	the least number of ordered subpatterns of a candidate
w	the threshold of the minimal number of subpatterns which occur in X
$lastLen$	the length of those last characters of P whose length is shorter than q'
PT	the table which records the positions of each subpattern P_i occurring in text X
$PrunedSPT$	the table after pruning unreasonable start positions from table PT
$EndPT$	the table after computing the related end positions for table $PrunedSPT$
$PrunedEPT$	the table after pruning unreasonable end positions from table $EndPT$
RM	the array mapped from the reasonable end positions of subpatterns in table $PrunedEPT$
$ListOfDPRanges$	the list used to store the ranges of candidates

the usability is restricted. In other words, the $(k+s)$ q-samples filter cannot handle the situation when the length of a query pattern is too short. For example, when m is 10 and the number of allowed errors, k, is 4, we have $q' = \lceil \log_\sigma m \rceil = \lceil \log_4 10 \rceil = 2$ and $h = \lfloor (m-k-q+1)/(k+s) \rfloor = \lfloor (10-4-2+1)/(4+2) \rfloor = \lfloor 5/6 \rfloor = 0$. Since h is 0, the $(k+s)$ q-samples filter will not take any sample. That is, it will not search the database.

In the hash trie filter, when a query is inquired, first, we compute length q' of a subpattern by using the formula the same as that of the $(k+s)$ q-samples filter, *i.e.*, $q' = \lceil \log_4 m \rceil$, where m is the length of query pattern P. Note that since we let $q = \lceil \log_4 n \rceil$ for text X (where $n = |X|$) and $m \leq n$, the condition $(q' \leq q)$ is always satisfied, where q' is the length of a subpattern of the query and q is the length of a gram in the hash trie. In this case, the subpattern with length q' will always be found, if it exists in the hash trie. Then, our method will dynamically decide the number of exactly-matching subpatterns (*i.e.*, s) which satisfy the global order, according to the number of allowed errors, k. We let $NumSP = \lfloor m/q' \rfloor$ and $s = NumSP - k$. When the value of $NumSP$ increases or the value of k decreases, the value of s will increase, and the precision will also increase. If the value of s is smaller than a threshold, w, it means that the precision of filtration may be too low. In this case, we will reduce the length of one gram, *i.e.*, q', by 1 to increase the value of $NumSP$.

Table 2 shows a comparison of the values of s between the $(k+s)$ q-samples filter and the proposed hash trie filter under $m = 17$ and various values of k, where w is 2. From this table, we show that in the hash trie filter, when $k = 1$, the value of s can increase to 4. Moreover, in the $(k+s)$ q-samples filter, when the value of k increases, the value of h may decrease to a value smaller than q', which will result in splitting the grams overlappedly. Figure 9-(a) shows this case, while Figure 9-(b) shows the case of non-overlapped grams when $h \geq q'$. We can see that the number of grams in Figure 9-(a) is larger than that in Figure 9-(b). That will lead to too many grams and result in too many candidates, if s is fixed to 2 in the $(k+s)$ q-samples filter.

3.2.2 Step 2: Constructing Table PT for the Subpatterns

After computing the value of q', next, we apply procedure *ConstructPT*(q', P, PT) to split query pattern P into several non-overlapped grams, and traverse them in the hash

Table 2: A comparison of the values of s in the $(k+s)$ q-samples filter and the hash trie filter under $m = 17$ and $w = 2$

m	k	q'	$(k+s)$ q-samples		hash trie
			h	s	s
	1		4	2	4
17	2	3	3	2	3
	3		2	2	2

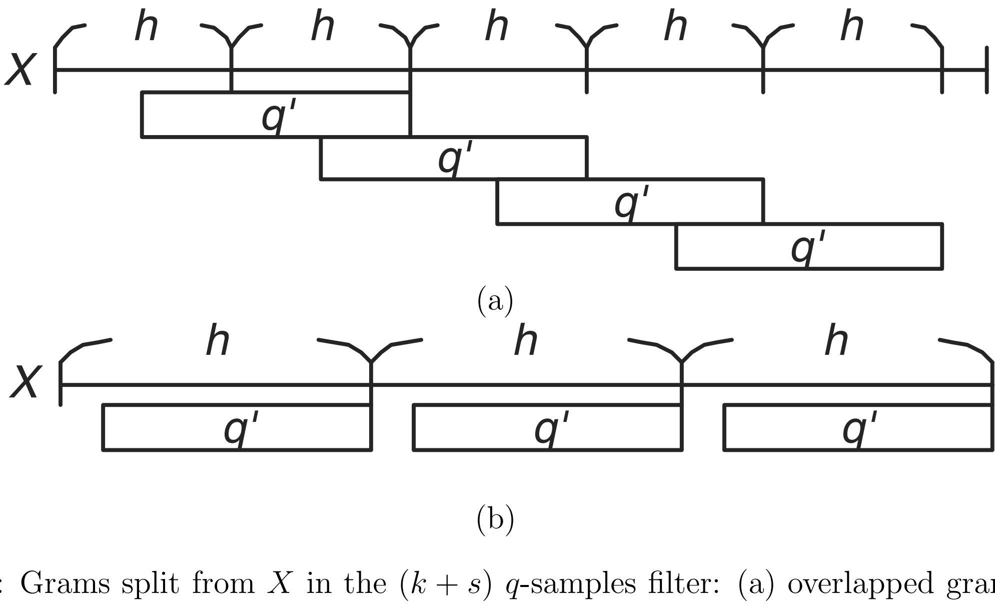


Figure 9: Grams split from X in the $(k+s)$ q-samples filter: (a) overlapped grams when $h < q'$; (b) non-overlapped grams when $h \geq q'$

$$P = \underline{C\ C\ G}\ \underline{G\ T\ T}\ \underline{A\ C\ G}\ \underline{C\ C\ T}\ \underline{T\ A\ C}\ G\ G\ \$$$

P	C C G	G T T	A C G	C C T	T A C
P_i	P_1	P_2	P_3	P_4	P_5

Figure 10: Splitting query pattern P into 3-grams (subpatterns)

PT

gram	positions
P_1	(3 , 19)
P_2	(6)
P_3	(9)
P_4	(14)
P_5	(8, 12 , 17)

Figure 11: Table PT for storing the positions of subpattern P_i in text X

trie, HT. Moreover, we will generate a position table (PT) which stores the positions of each subpattern P_i of P.

Figure 10 shows an example of how query pattern P is split into non-overlapped grams. Assume that m is 17 and k is 3. Since the length of a gram, q', is equal to $\lceil \log_4 17 \rceil = 3$, we split query pattern P into 5 ($= NumSP$) non-overlapped subpatterns with length 3, *i.e.*, $\{P_1, P_2, P_3, P_4, P_5\}$. The remaining part whose length is less than q' ("GG" in this example) is ignored at this point, and will be considered in Step 4. Moreover, its length is stored in $lastLen$. In this example, the value of $lastLen$ is 2.

After query pattern P is split into several non-over-lapped subpatterns, these subpatterns are traversed in hash trie HT. After traversing, we get the positions of each gram of query pattern P occurring in text X. In our example, after splitting query pattern P into 5 non-overlapped subpatterns $\{P_1, P_2, P_3, P_4, P_5\}$, we traverse them in hash trie HT of text X to get all positions of these subpatterns occurring in text X. Figure 11 shows table PT which stores the positions of each subpattern P_i occurring in the hash trie shown in Figure 7-(b).

3.2.3 Step 3: The Pruning Step

After finding all positions of each subpattern P_i occurring in text X, we will prune the unreasonable start/end positions. Note that a start or end position, denoted by $z1$ or $z2$ respectively, is unreasonable, if (1) $z1 < (1 - k)$ or (2) $z2 > (n + k)$.

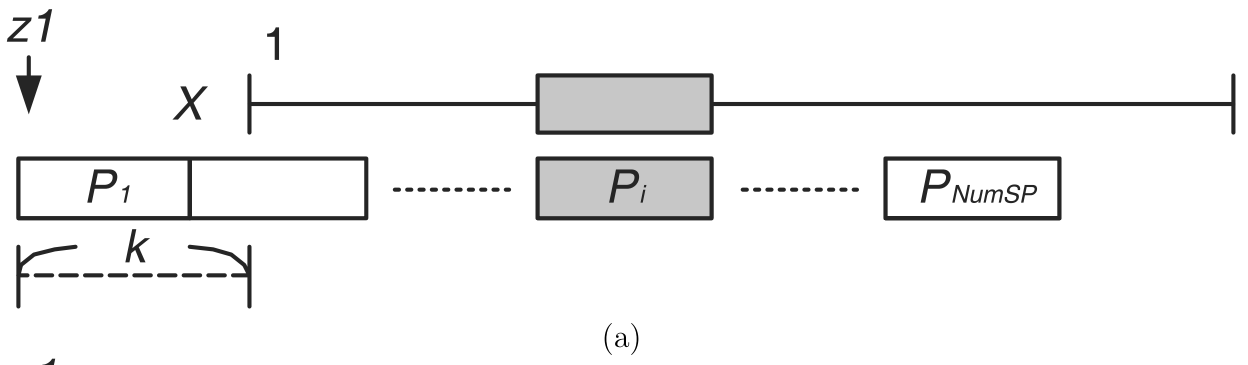
From now on, each subpattern P_i of query pattern P will be treated individually. For Case 1, we consider the unreasonable start positions. We apply procedure $PruneStartPosition(PT,$ $PrunedSPT)$ to prune these unreasonable start positions from table PT, and store the result in table $PrunedSPT$.

Let $PL_i = PT[P_i]$ denote a list of positions stored in table PT, *i.e.*, a list of positions of subpattern P_i occurring in text X. Let $PL_i[j]$ denote the j-th position stored in PL_i. Note that we allow k errors between query pattern P and text X. Assume that all k errors occur in front of each position of PL_i, which results in the smallest reasonable start position. For each matching subpattern P_i, the start position $z1$ of its first gram in text X is

$$z1 = PL_i[j] - q' * (i - 1), \tag{1}$$

where $j \in 1 \dots |PL_i|$ and q' is the length of a subpattern. The idea is that when P_i occurs at one position (*i.e.*, $PL_i[j]$), P_1 can only occur at some position, (*i.e.*, $z1$). In this case, we use the start position of P_1 (*i.e.*, $z1$) to decide whether this start position is reasonable. The value of $z1$ should not be smaller than $(1 - k)$; otherwise, this substring of text X can not become query P within k edit operations. That is, if $z1 \geq (1 - k)$, $z1$ is a reasonable start position; otherwise, $z1$ is a unreasonable start position. Therefore, we can prune those unreasonable positions whose $z1$ are smaller than $(1 - k)$.

Figure 12 shows examples in which both of query patterns are split into $NumSP$ subpatterns. Figure 12-(a) shows an example of a reasonable start position, $z1$, for subpattern P_i, where $z1 = (1 - k)$. That is, when P_i occurs at one position, the start position of the first subpattern, P_1, will occur at position $(1-k)$. This is a reasonable start position, since we can delete the first k characters of P to match P with one substring of X. On the other hand, Figure 12-(b) shows an example of an unreasonable start position, $z1$, for subpattern P_i, where $z1 < (1 - k)$. That is, when P_i occurs at one position, the start position of the first subpattern, P_1, will occur at a position less than $(1 - k)$. In this case, we can not



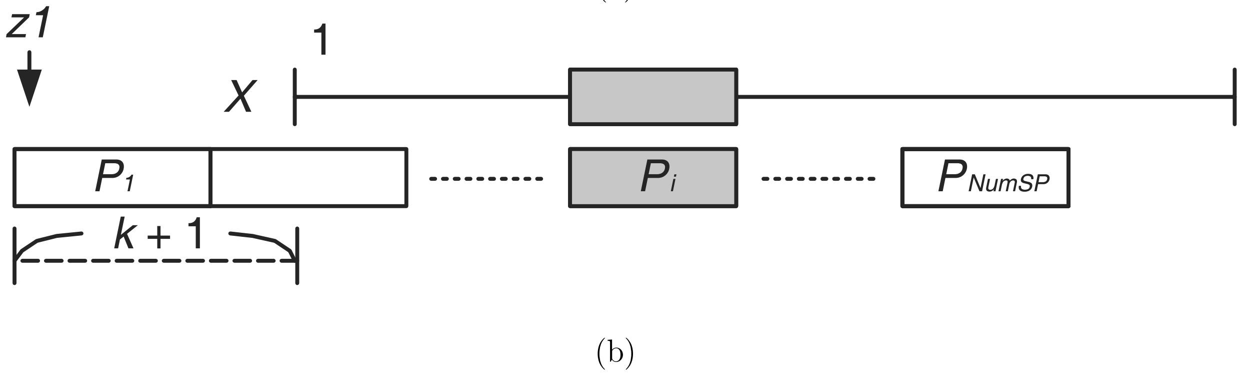


Figure 12: Two possible start positions $z1$ of a substring of text X: (a) the reasonable start position; (b) the unreasonable start position

match P with one substring of X within k edit operations. Therefore, when $z1 < (1-k)$, we say that such a start position is unreasonable.

Figure 13-(a) shows position table PT in our example, where symbol "×" means the unreasonable position which should be pruned. Figure 13-(b) shows the result of pruning the unreasonable positions of each subpattern P_i of query pattern P, where position 8 of subpattern P_5 is pruned. For subpatterns $P_1 \ldots P_5$, we compute their start position $z1$ by equation (1). Therefore, the first position of PL_5 in table PT shown in Figure 13-(a), *i.e.*, $PL_5[1] = 8$, is pruned, since its start position $z1 = PL_5[1] - 3 * 4 = 8 - 12 = -4 < (1-k) = -2$. That is, when P_5 occurs at position 8, P_1 must occur at position -4, which is an unreasonable start position.

Similarly, for Case 2, we consider the unreasonable end positions. Before this pruning step, we apply procedure $ComputeEndPosition(PrunedSPT, EndPT)$ to compute the maximal related end positions for positions stored in table $PrunedSPT$, and store the result in table $EndPT$. Next, we apply procedure
$PruneEndPosition(EndPT, PrunedEPT)$ to prune

PT

gram	positions
P_1	(3 , 19)
P_2	(6)
P_3	(9)
P_4	(14)
P_5	(~~8~~, 12 , 17)

(a)

PrunedSPT

gram	positions
P_1	(3 , 19)
P_2	(6)
P_3	(9)
P_4	(14)
P_5	(12 , 17)

(b)

Figure 13: The case of pruning the unreasonable start positions: (a) the original position table, PT (b) table $PrunedSPT$ after pruning unreasonable start positions

the unreasonable end positions stored in table $EndPT$, and store the result in table $PrunedEPT$.

Let $PL_i = PrunedSPT[P_i]$ denote a list of positions stored in table $PrunedSPT$. Assume that all k errors are caused by insertion occurring after each position of PL_i, which results in the maximal end position. The maximal related end position, $z2$, should be

$$z2 = PL_i[j] + q' * (NumSP - i + 1) - 1 + k, \qquad (2)$$

where $j \in 1 \ldots |PL_i|$. The value of $z2$ should not be larger than $(n + k)$; otherwise, this substring of text T can not become query P within k edit operations. That is, if $z2 \leq (n + k)$, $z2$ is a reasonable end position; otherwise, $z2$ is a unreasonable end position.

Figure 14 shows examples in which both of query patterns are split into $NumSP$ subpatterns. Figure 14-(a) shows an example of a reasonable maximal end position, $z2$, for subpattern P_i, where $z2 = (n + k)$. That is, when P_i occurs at one position, the maximal end position of the last subpattern, P_{NumSP}, will occur at position $(n + k)$. This is a reasonable start position, since we can delete the last k characters of P to match P with one substring of X. On the other hand, Figure 14-(b) shows an example of an unreasonable maximal end position, $z2$, for subpattern P_i, where $z2 > (n + k)$. That is, when P_i occurs at one position, the end position of the last subpattern, P_{NumSP}, will occur at a position larger than $(n + k)$. In this case, we can not match P with one substring of X within k edit

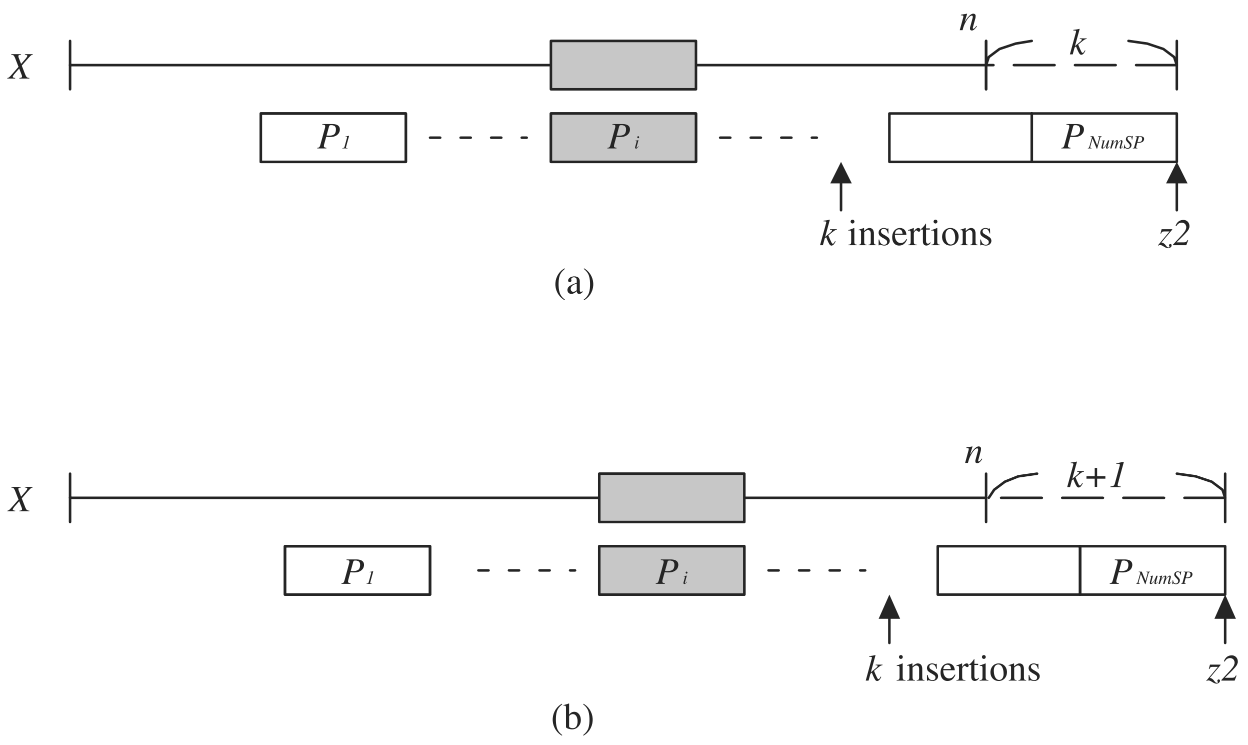


Figure 14: The situations for end position $z2$: (a) the reasonable end position; (b) the unreasonable end position

operations. Therefore, when $z2 > (n+k)$, we say that such an end position is unreasonable.

Figure 15-(a) shows table $PrunedSPT$ in our example, and Figure 15-(b) shows the resulting table by computing all related end positions $z2$ for the occurring positions of each subpattern P_i according to equation (2). For example, for subpattern P_2, it occurs at position 6. Therefore, the end position of the last subpattern, P_{NumSP}, is $z2 = 6 + 3 * (5 - 2 + 1) - 1 + 3 = 20$, recorded in the tuple of P_2 of table $EndPT$ shown in Figure 15-(b). Then, we can prune those unreasonable positions whose end positions $z2$ are larger than $(n + k)$. Figure 16-(a) shows the related end position table, $EndPT$ (the same as that in Figure 15-(b)), where symbol "×" means the unreasonable position which should be pruned. In this example, position 36 of subpattern P_1 in table $EndPT$ is pruned, since we have $z2 = 36 > (n + k) = 27 + 3 = 30$. Figure 16-(b) shows the resulting table after pruning the unreasonable end position.

PrunedSPT

gram	positions
P_1	(3 , 19)
P_2	(6)
P_3	(9)
P_4	(14)
P_5	(12 , 17)

(a)

EndPT

gram	related end
P_1	(20 , 36)
P_2	(20)
P_3	(20)
P_4	(22)
P_5	(17 , 22)

(b)

Figure 15: The case of computing the related end positions: (a) the reasonable start position table, $PrunedSPT$; (b) the related end position table, $EndPT$

EndPT

gram	related end
P_1	(20 , ~~36~~)
P_2	(20)
P_3	(20)
P_4	(22)
P_5	(17 , 22)

(a)

PrunedEPT

gram	related end
P_1	(20)
P_2	(20)
P_3	(20)
P_4	(22**)
P_5	(17* , 22)

(b)

Figure 16: The case of pruning the unreasonable end positions: (a) the related end position table, $EndPT$; (b) the reasonable end position table, $PrunedEPT$

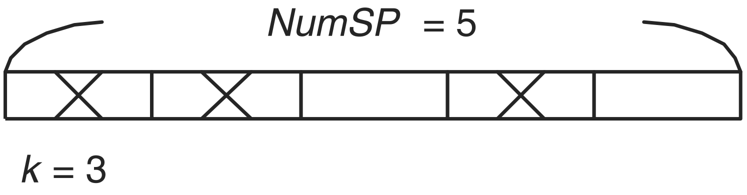


Figure 17: A candidate with s (= 2) matching subpatterns

3.2.4 Step 4: Finding the Ordered Subpatterns

For a substring of text X, when there are at least s subpatterns P_i of query pattern P matching within this substring, it will be a possible candidate. The reason is that k errors will affect at most k subpatterns in the worst case. Figure 17 shows the illustration. For example, the total number of subpatterns ($NumSP$) of query pattern P is 5, and the number of allowed errors (k) is 3. Since 3 errors affect at most 3 subpatterns, there exist at least $s = (NumSP - k) = (5 - 3) = 2$ matching subpatterns.

To efficiently find these ordered s subpatterns, we apply the following two steps: (1) mapping the reasonable end positions to a 2-dimensional binary array, RM; (2) using a sliding window of size $(k + 1)$ over array RM to find the ordered subpatterns.

First, we utilize procedure $MapToRM($
$PrunedEPT$, $RM)$ to initialize a 2-dimensional binary array, RM. Let $maxEnd$ and $minEnd$ denote the maximal and the minimal values of positions in table
$PrunedEPT$. Array RM is a two-dimensional array with size $NumSP * (MaxEnd - minEnd + 1)$. Initially, each cell value of array RM is set to 0. After initializing array RM, we change the cell values of array RM to 1, according to the value of $PL_i[j]$, $\forall i \in 1 \ldots NumSP$ and $\forall j \in 1 \ldots |PL_i|$, where PL_i represents the position list of $PrunedEPT[P_i]$.

In our previous example shown in Figure 16-(b), $minEnd$ and $maxEnd$ are those positions denoted with "*" and "**", *i.e.*, 17 and 22, respectively. Therefore, we have array RM with size $NumSP * (MaxEnd - minEnd + 1) = 5 * 6$. Figure 18 shows the resulting array mapped from table $PrunedETP$ shown in Figure 16-(b). In this array, for example, since subpattern P_1 has one end position, 20, the value of $RM[P_1, 20]$ is changed to 1.

Second, we want to find whether there are at least s subpatterns occurring in a substring of text X in order. After mapping all end positions of subpatterns in the previous step, we

RM

P_i	17	18	19	20	21	22
P_1	0	0	0	1	0	0
P_2	0	0	0	1	0	0
P_3	0	0	0	1	0	0
P_4	0	0	0	0	0	1
P_5	1	0	0	0	0	1

Figure 18: Array RM for reasonable end positions of matching subpatterns in X

apply procedure $Find\_s\_WithSW($ RM, $ListOfDPRange)$ to find the candidates which contain s subpatterns occurring in order. We utilize a sliding window, SW, with size $(k+1)$, and a counting array, CT, with size $(maxEnd - minEnd - 1 - k) * NumSP$. Basically, we use the "-/+" operations on the columns of array RM and array CT to find whether there are s subpatterns occurring in order. Note that our sliding window is applied on array RM, while the sliding windows of other previous filter methods are applied on text X.

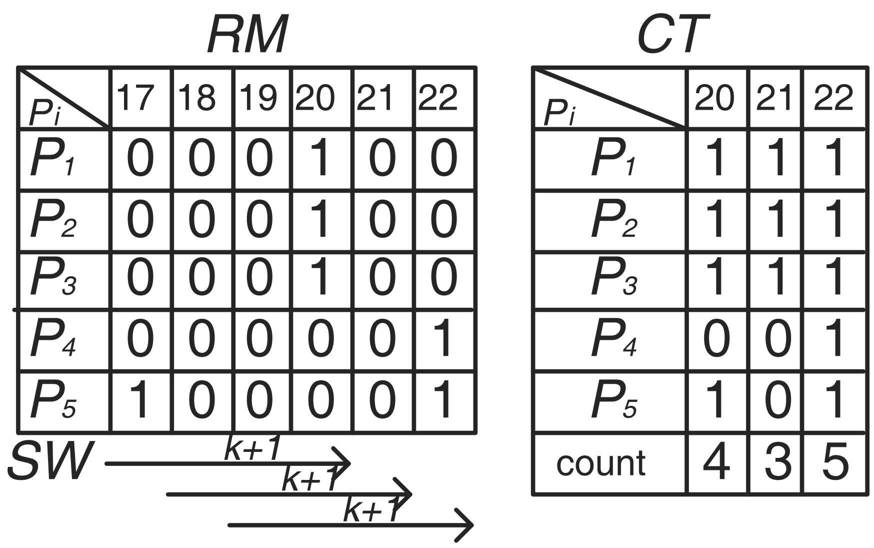
$CT[P_i, j]$ is used to count the number of occurrence of subpatterns P_i between $RM[P_i, (j-k)]$ and $RM[P_i, j]$. For the first sliding window, we use k times of "+" operations on the columns between $RM[P_i,$
$minEnd]$ and $RM[P_i, (k+minEnd)]$, and store the result in $CT[P_i, (k+minEnd)]$. Therefore,

$$\begin{aligned} & CT[P_i, (k + minEnd)] \\ =& RM[P_i, minEnd] + \ldots + RM[P_i, (k + minEnd)]. \end{aligned} \tag{3}$$

Then, to efficiently count the number of occurrence of subpatterns P_i, instead of using k times of "+" operations, the rest of values of array CT, except values of the first column, are calculated as

$$\begin{aligned} & CT[P_i, j] \\ =& CT[P_i, (j - 1)] + RM[P_i, j] - RM[P_i, (j - k - 1)], \end{aligned} \tag{4}$$

where $RM[*, j]$ is the new column that enters the sliding window, and $RM[*, (j-k-1)]$ is


RM

P_i	17	18	19	20	21	22
P_1	0	0	0	1	0	0
P_2	0	0	0	1	0	0
P_3	0	0	0	1	0	0
P_4	0	0	0	0	0	1
P_5	1	0	0	0	0	1

CT

P_i	20	21	22
P_1	1	1	1
P_2	1	1	1
P_3	1	1	1
P_4	0	0	1
P_5	1	0	1
count	4	3	5

Figure 19: The result of finding s ordered subpatterns of query pattern P in text X

the oldest column that leaves the sliding window (*i.e.*, shifting the sliding window to right by one column in array RM).

After counting the values of columns within one sliding window of array RM and storing the result in one column of array CT, we count the number of occurrences of subpatterns P_i in this column of array CT, where $i = 1 \ldots NumSP$. If $CT[P_i, j] \neq 0$, $CT[count, j]$ is increased by 1, where $i = 1 \ldots NumSP$. That is, $CT[count, j]$ records the number of matching subpatterns in the current sliding window.

Take Figure 19 as an example. We utilize a sliding window, SW, with size $(k + 1) = (3 + 1) = 4$ to find $s = (NumSP - k) = (5 - 3) = 2$ ordered subpatterns. For the first sliding window shown in Figure 19, we apply "+" operations on values of the first $(k+1) = (3+1) = 4$ columns (*i.e.*, the first sliding window). Figure 20 shows the process for the first sliding window. The result is {1, 1, 1, 0, 1} for subpatterns $P_1 \ldots P_5$, respectively. This result is stored in the first column of array CT, *i.e.*, $CT[*, 20]$ in Figure 19. With this result, we obtain $CT[count, 20] = 4$, which means that there are 4 subpatterns (*i.e.*, P_1, P_2, P_3, and P_5 in this example) occurring within this sliding window in order. Since this number of matching subpatterns is larger than the value of $s(= 2)$, there may be an ASM happened. Therefore, we mark one range of text X according to the last position, 20, of the current sliding window as a candidate, *i.e.*, $X[((20 + lastLen) - (m + 2k)) \ldots (20 + lastLen)] =$

		P_1	P_2	P_3	P_4	P_5
	RM[17]	0	0	0	0	1
	RM[18]	0	0	0	0	0
	RM[19]	0	0	0	0	0
+	RM[20]	1	1	1	0	0
=	CT[20]	1	1	1	0	1

Figure 20: The processing result of $CT[*, 20]$ in our example

		P_1	P_2	P_3	P_4	P_5
	CT[20]	1	1	1	0	1
-	RM[17]	0	0	0	0	1
		1	1	1	0	0
+	RM[21]	0	0	0	0	0
=	CT[21]	1	1	1	0	0

Figure 21: The processing result of $CT[*, 21]$ in our example

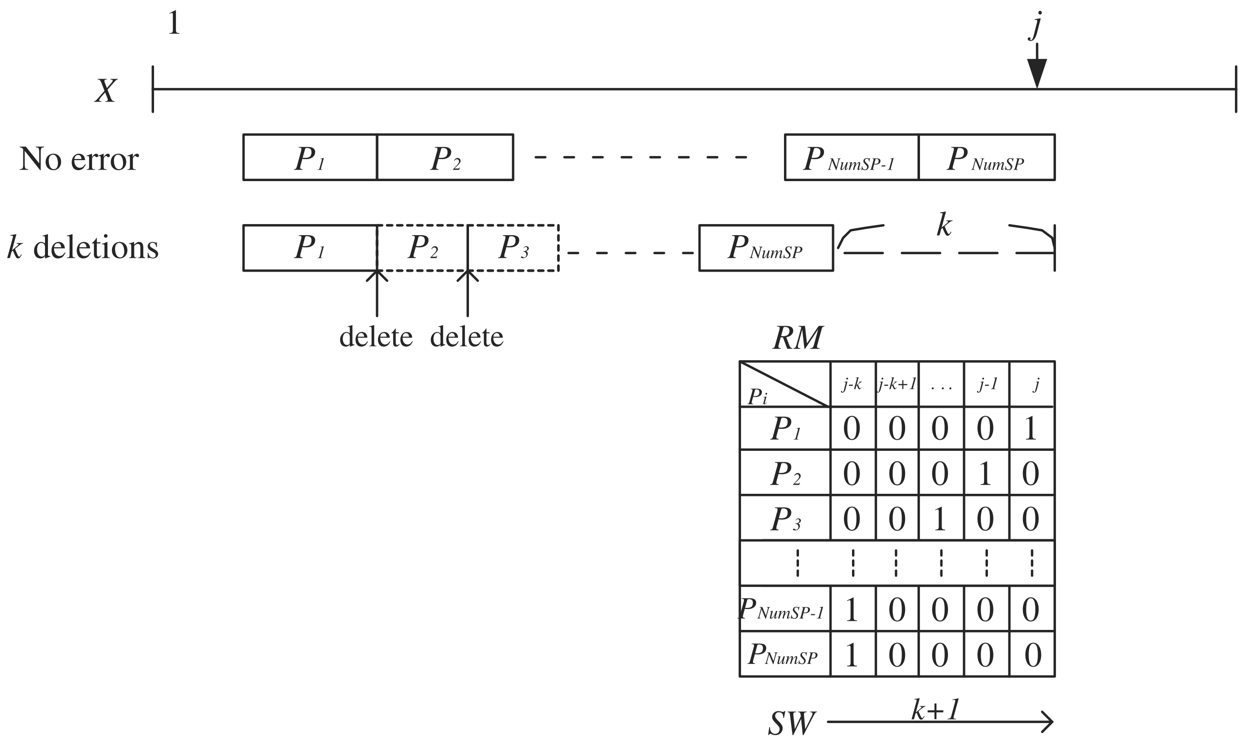
$X[((20 + 2) - (17 + 2 * 3)) \dots (20 + 2)] = X[-1 \dots 22]$, where $lastLen$ is generated in Step 2. (We will explain how this range is derived later.) Because the minimal start index of X is 1, $X[-1 \dots 22]$ is modified to $X[1 \dots 22]$. We record this range in $ListOfDPRanges$.

Up to this point, the counting step of this sliding window is done. The sliding window leaves column $RM[*, 17]$ and includes the next column, $RM[*, 21]$. Then, $CT[*, 21]$ is calculated as $CT[*, 21] = CT[*, 20] - RM[*, 17] + RM[*, 21]$. Figure 21 shows the process of the result of $CT[*, 21]$. Therefore, we get $CT[*, 21] = \{1, 1, 1, 0, 0\}$ for subpatterns $P_1 \dots P_5$, respectively. With this result, we know that there are 3 subpatterns occurring in order, and there may be an ASM happened. Therefore, we mark $X[((21 + lastLen) - (m + 2k)) \dots (21 + lastLen)] = X[((21 + 2) - (17 + 2 * 3)) \dots (21 + 2)] = X[0 \dots 23]$ as a candidate. Similarly, $X[0 \dots 23]$ is modified to $X[1 \dots 23]$ and is stored in $ListOfDPRanges$.

In Figure 19, there are three times of window movements, and we store the counting results in array CT to determine whether there exist more than s matching subpatterns. In our example, we will store $X[1\ldots 22]$, $X[1\ldots 23]$, and $X[1\ldots\ 24]$ in list $ListOfDPRange$. We can merge those candidates whose ranges have overlaps with others in $ListOfDPRange$. In this example, candidates $X[1\ldots\ 22]$, $X[1\ldots 23]$, and $X[1\ldots 24]$ can be merged into one candidate, $X[1\ldots 24]$. It will be verified in Step 5.

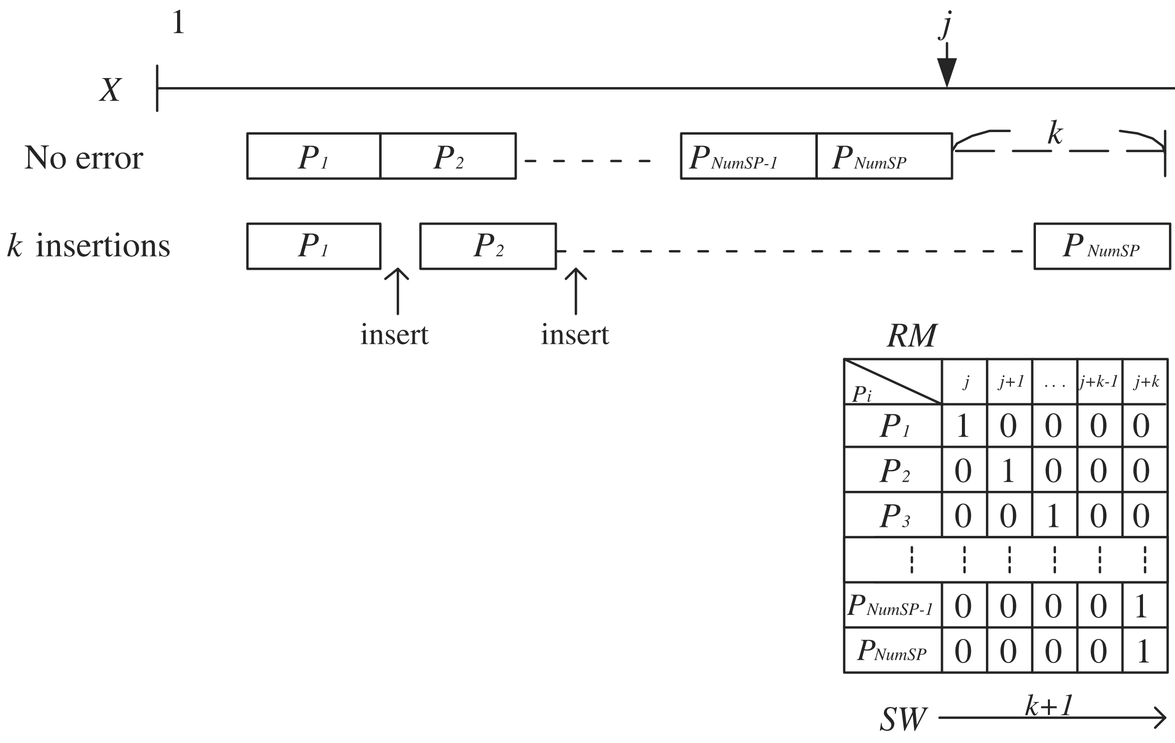
Now, we explain why the size of a window is $(k+1)$. Figures 22 and 23 show the illustration. Assume that all subpatterns can be matched in text X one by one without any error, and the end position of subpattern P_{NumSP} is j. When all errors are deletions, the related end positions of the subpatterns, which are behind the k deletions, should move forward by k characters (*i.e.*, at position $j-k$), as shown in Figure 22. A sliding window with size $j-(j-k)+1=(k+1)$ will include this case. On the other hand, when all errors are insertions, the related end positions of the subpatterns, which are behind the k insertions, should move backward by k characters (*i.e.*, at position $j+k$), as shown in Figure 23. A sliding window with size $(j+k)-j+1=(k+1)$ will include this case. (Note that k deletions and k insertions can not occur at the same time.) Therefore, the size of a sliding window can be $(k+1)$.

Assume that in array CT (as shown in Figure 19), the value of $CT[count, j]$ is not smaller than s, which means that there exist at least s subpatterns occurring in the sliding window. In this case, we pass $X[(j+lastLen)-(m+2k)\ldots(j+lastLen)]$ as a candidate, where $lastLen$ is the length of the rest of the query pattern that are not split into subpatterns, as mentioned in Subsection 3.2.2. The length of a candidate in our hash trie filter is $(m+2k)$, while in the $(k+s)$ q-samples method [22], this length is about $(m+3k)$. The length of a candidate will affect the time for verifying. Therefore, our hash trie filter is more precise and more efficient than the $(k+s)$ q-samples filter. We use Figure 24 to illustrate why this length can be $(m+2k)$. The maximal length of a candidate is determined when all errors are insertions, since the length of a candidate with no error or k deletions must be shorter than the length of a candidate with k insertions. Assume that all k insertions occur after subpattern P_1. The difference of the end positions between subpatterns P_1 and P_x $(2 \leq x \leq NumSP)$ is k. Therefore, if the end position of P_x is j, the range of


1
j
X
No error
P_1 P_2 $P_{NumSP-1}$ P_{NumSP}
k deletions
P_1 P_2 P_3 P_{NumSP}
k
delete delete
RM
SW
k+1

P_i	j-k	j-k+1	. . .	j-1	j
P_1	0	0	0	0	1
P_2	0	0	0	1	0
P_3	0	0	1	0	0
⋮	⋮	⋮	⋮	⋮	⋮
$P_{NumSP-1}$	1	0	0	0	0
P_{NumSP}	1	0	0	0	0

Figure 22: The situation when query pattern P occurs in text X with k deletions



P_i	j	j+1	. . .	j+k-1	j+k
P_1	1	0	0	0	0
P_2	0	1	0	0	0
P_3	0	0	1	0	0
⋮	⋮	⋮	⋮	⋮	⋮
$P_{NumSP-1}$	0	0	0	0	1
P_{NumSP}	0	0	0	0	1

Figure 23: The situation when query pattern P occurs in text X with k insertions

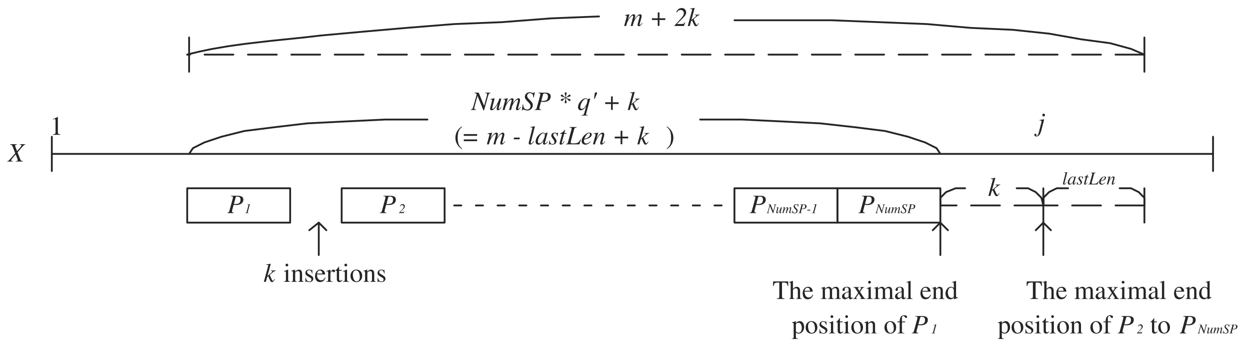


Figure 24: Illustration of the length of a candidate with k insertions

$X[(j+lastLen)-(m+2k)\dots(j+lastLen)]$ will contain such a candidate with k insertions. (Note that m is equal to $NumSP * q' + lastLen$.)

3.2.5 Step 5: Verification with Dynamic Programming

The last step of procedure $Search(P, HT)$ shown in Figure 8 is to verify candidates with dynamic programming. We apply procedure $VerifyWithDP$ to find approximate matches for query pattern P from the candidates. The dynamic programming [21] evaluates edit distance matrix $C[0\dots|P|,\ 0\dots|X|]$, where $C[i, j]$ represents the minimum number of operations needed to match $X[1\dots i]$ with $P[1\dots j]$. The idea is that any text position is the potential start of an ASM. This is achieved by setting $C[0, j] = 0$ for all $0 \le j \le n$. The formulas are shown as follows:

$$\begin{cases} C[i,0] = i \\ C[0,j] = 0 \\ C[i,j] = \begin{cases} C[i-1,j-1], & \text{if } T[i] = P[i] \\ 1 + min(C[i-1,j], C[i,j-1], C[i-1, \\ j-1]), & \text{otherwise.} \end{cases} \end{cases}$$

If the value of $C[|P|, i]$ is not larger than k, there is an ASM which ends at position i. In this case, a traceback step is performed to find the start position of this ASM.

4 Performance

In this section, we study the performance of the proposed hash trie filter, and make a comparison with the $(k + s)$ q-samples filter by simulation. The same query patterns are used for the hash trie filter (HTF) and the $(k + s)$ q-samples filter ($ksqF$). We apply the same dynamic programming algorithm to verify the candidates for both the hash trie filter and the $(k + s)$ q-samples filter. Our experiments were 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.

4.1 The Real DNA Sequences

In order to experiment in the real situation, the data used in our performance study were chosen from the real DNA sequences in GenBank (http://www.ncbi.nlm.
nih.gov). We chose gene "Z73521" contained in human chromosome 16, whose length is 7881 base pairs, and contig "NT_167199.1" contained in human chromosome Y, whose length is 34821 base pairs, to be the database sequences. Moreover, we generate query patterns from them. The query patterns were selected randomly from these DNA sequences.

The filter methods will lose their filtration power at error levels ($= k/m$) over 30% [19], where k is the number of allowed errors and m is the length of the query pattern. Furthermore, the optimal error level for filter methods seems to be at about $0 - 20\%$ [22]. Therefore, our experiments were based on error levels about $0 - 20\%$. Besides, the value of w is set to 2 in this simulation, where w is the threshold used to prevent the value of s from being too small, as described in Subsection 3.2.1.

4.2 Time for Constructing a Hash Trie

In this subsection, we show the simulation result of constructing the hash trie. The database system will create a hash trie at first, which stores the positions of grams split from the DNA sequence. In this simulation, we would like to know how much time the system spends to construct a hash trie for different lengths of DNA sequences. We create several long DNA sequences by concatenating gene "Z73521" with itself until the total length is 5000, 15000, 20000, 25000, and 30000 base pairs, respectively. Figure 25 shows the construction time for DNA sequences with different lengths. From this figure, we show that our method requires only a little time to construct the hash trie in this simulation.

Next, we compare the total response time of our method with/without including time for constructing the hash trie, by using the real DNA sequence "Z73521" and a query

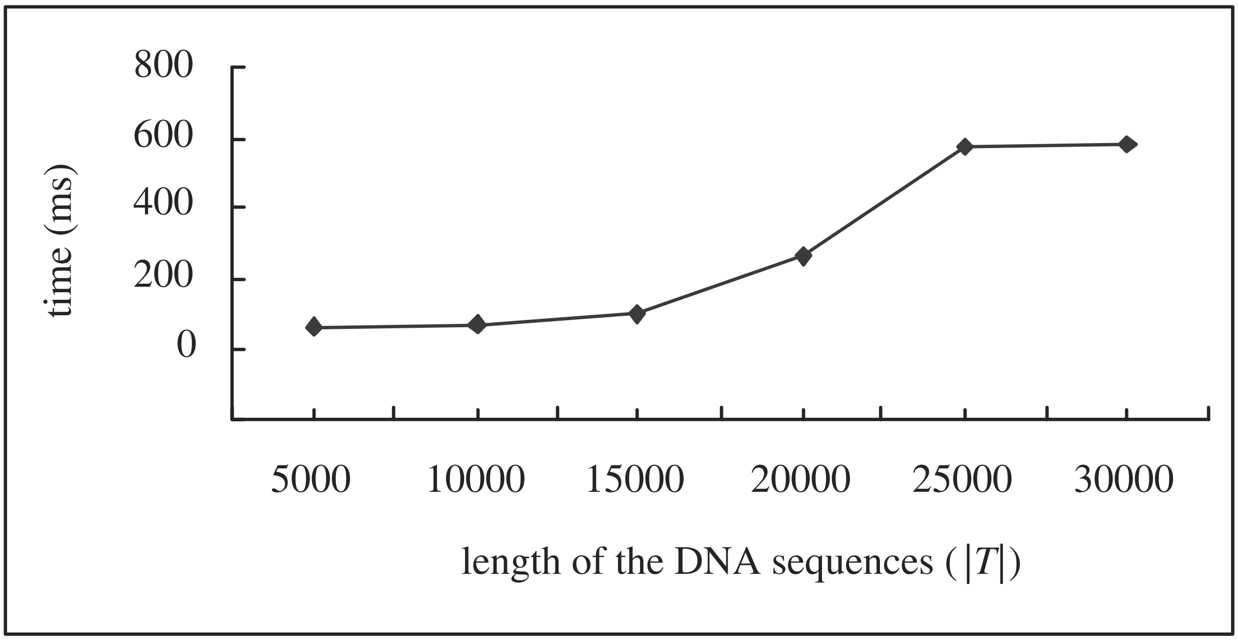


Figure 25: Time for constructing the hash trie for different lengths of DNA sequences

pattern with length 100 bps, under different number of errors, 0, 4, 8, 10, 12, 15, and 18, respectively. Figure 26 shows the result. From this figure, we show that the construction time does not have much effect on the experimental result.

4.3 Performance Under Short Query Patterns

In this subsection, we use gene "$Z73521$" as the database sequence and study the performance of the proposed hash trie filter under short query patterns. Figures 27, 28, and 29 show comparisons of the response time, the number of processing candidates, and the precision between the $(k + s)$ q-samples filter and the hash trie filter, respectively, under short query patterns with length 30 and varying the number of errors. From Figure 27, we observe that the hash trie filter is more efficient than the $(k + s)$ q-samples filter in terms of the response time no matter what value the number of errors is, under short query patterns. Obviously, as the number of errors increases, the response time for searching in both filter methods increases. However, the response time of the $(k + s)$ q-samples filter increases more quickly than that of the hash trie filter.

Figure 28 shows the reason why the hash trie filter is more efficient than the $(k + s)$ q-samples filter. From this figure, we observe that the number of candidates of the hash

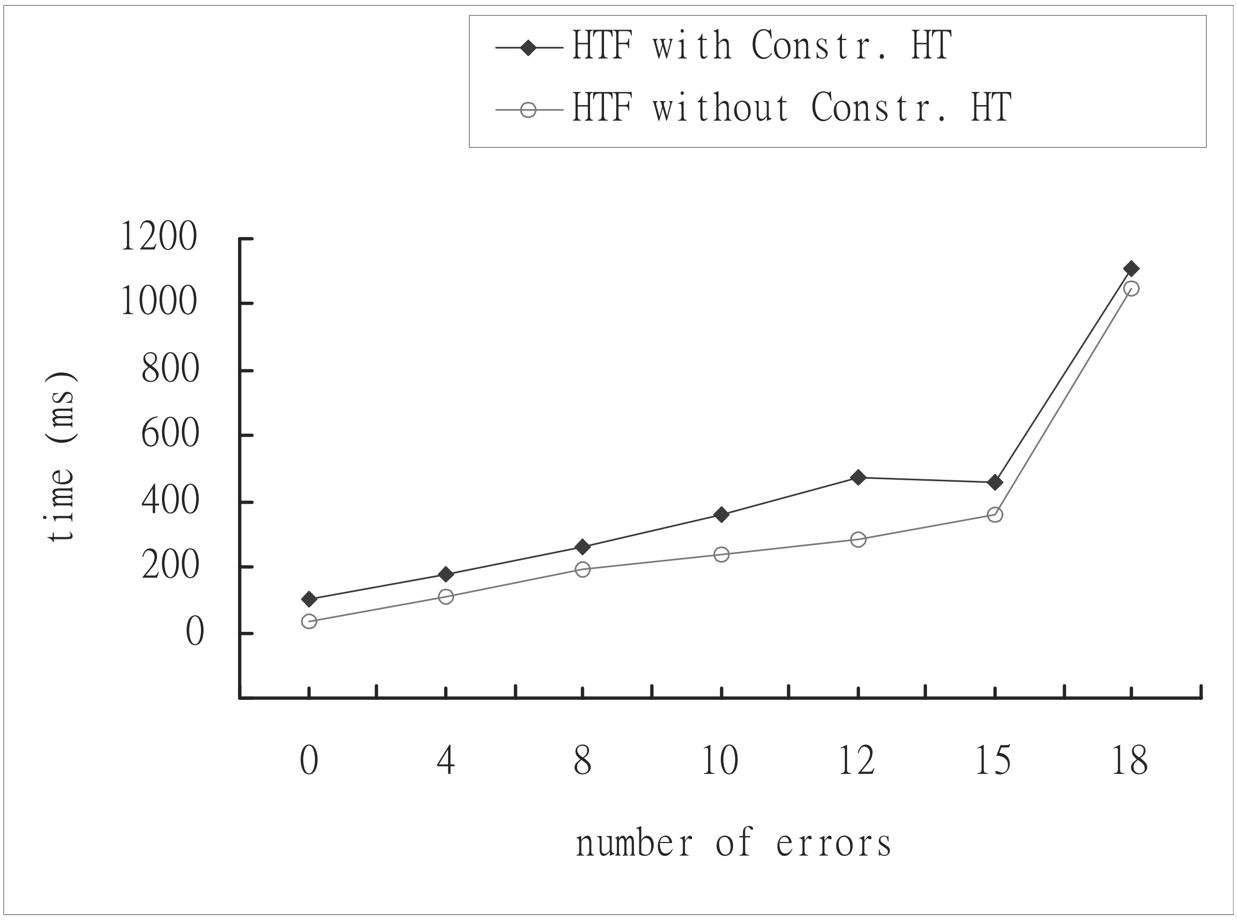


Figure 26: A comparison of the response time for the hash trie filter (HTF) with/without including hash trie construction

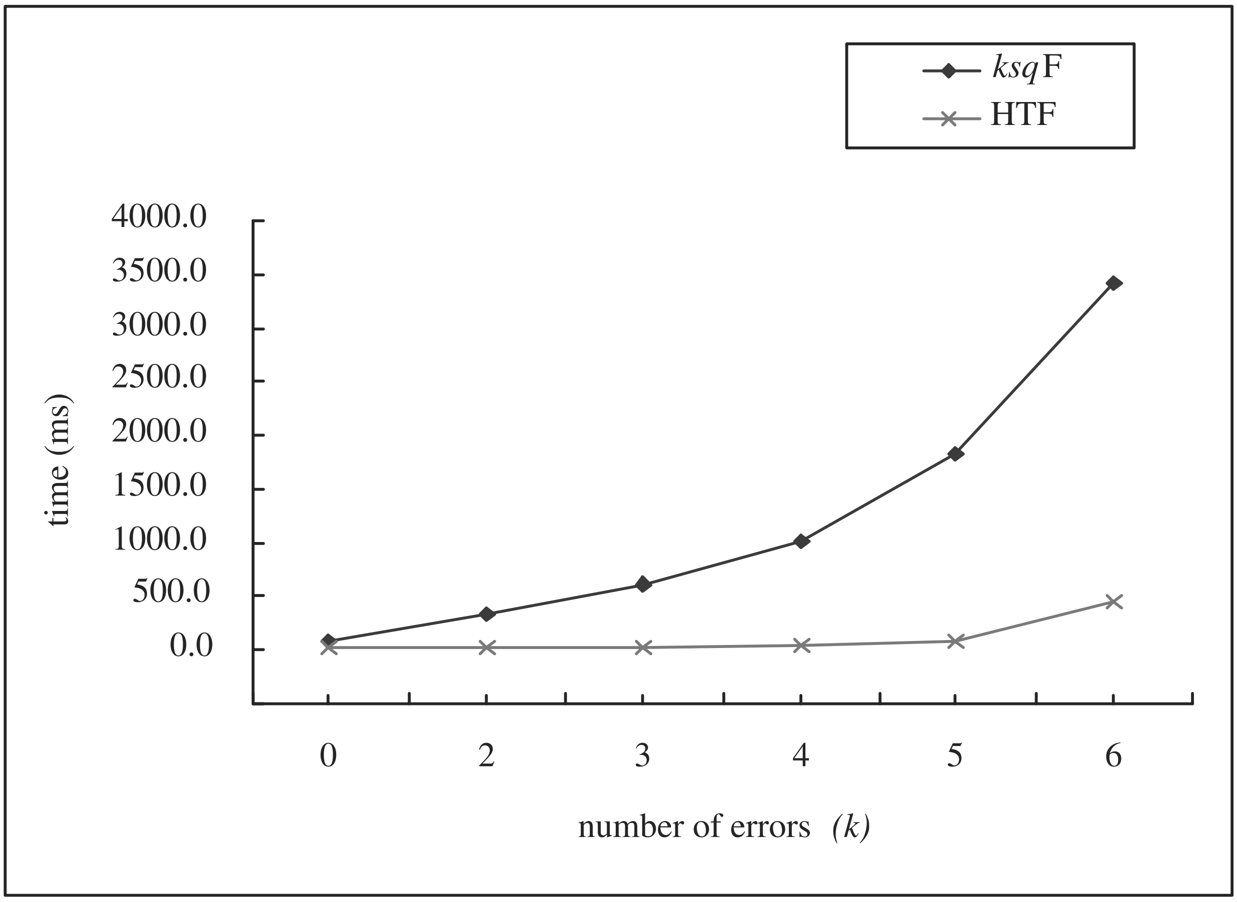


Figure 27: A comparison of the response time under short query patterns with length 30

Table 3: The values of s in the hash trie filter under short query patterns with length 30

k	0	2	3	4	5	6
s	10	8	7	6	5	4

trie filter is fewer than that of the $(k+s)$ q-samples filter, no matter what value the number of errors is, under short query patterns. Therefore, the time for verifying these candidates in the hash trie filter is also shorter than that of the $(k+s)$ q-samples filter. Obviously, as the number of errors increases, the number of candidates in both filter methods increases. However, the number of candidates in the $(k+s)$ q-samples filter increases more quickly than that of the hash trie filter.

Figure 29 shows a comparison of the precision between the $(k+s)$ q-samples filter and the hash trie filter. The precision is measured by the following formula: $(N_{hit}/N_{candidates})*100\%$, where N_{hit} is the number of hitting answers within the candidates, and $N_{candidate}$ is the number of candidates. Since the number of candidates of the hash trie filter is fewer than that of the $(k+s)$ q-samples filter, the precision of the former filter is also higher than that of the later filter. Obviously, as the number of errors increases, the precision in both filter methods decreases. When the number of errors is small, the precision of our hash trie filter could be up to 100% due to the large value of s. As shown in Table 3, when $m = 30$ and $k = 2$, we have $s = 8$. Although our strategy makes use of the large value of s to decide the condition of a candidate, it will not cause any missing case. The reason is that in the worst case, k errors will affect at most k subpatterns.

4.4 Performance Under Long Query Patterns / Low Error Levels / High Error Levels

In this subsection, we use gene "$Z73521$" as the database sequence and study the performance of the proposed hash trie filter under long query patterns/low error levels/high error levels. Figure 30 shows the comparisons of the response time, the number of processed candidates, and the precision, between the $(k+s)$ q-samples filter and the hash trie filter. Figure 30-(a) shows the comparisons under long query patterns with length 100 and varying the numbers of errors. Figure 30-(b) shows the comparisons under a low error level,

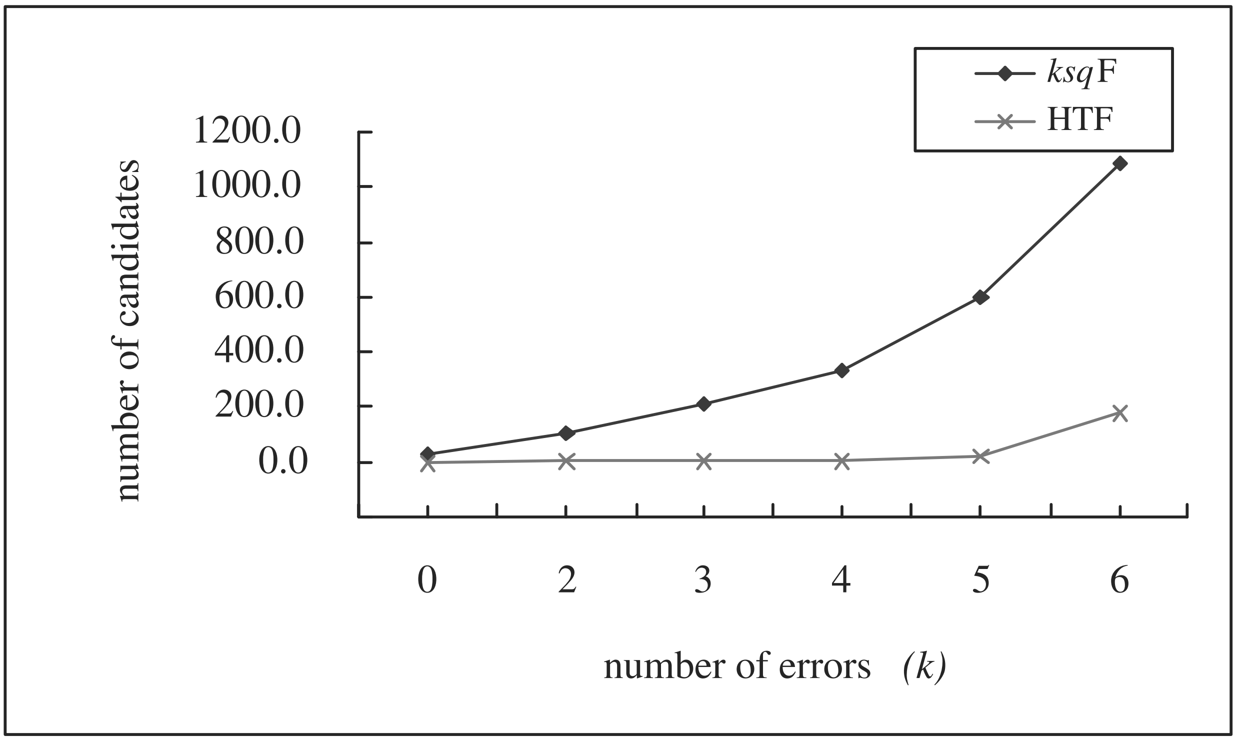


Figure 28: A comparison of the number of processing candidates under short query patterns with length 30

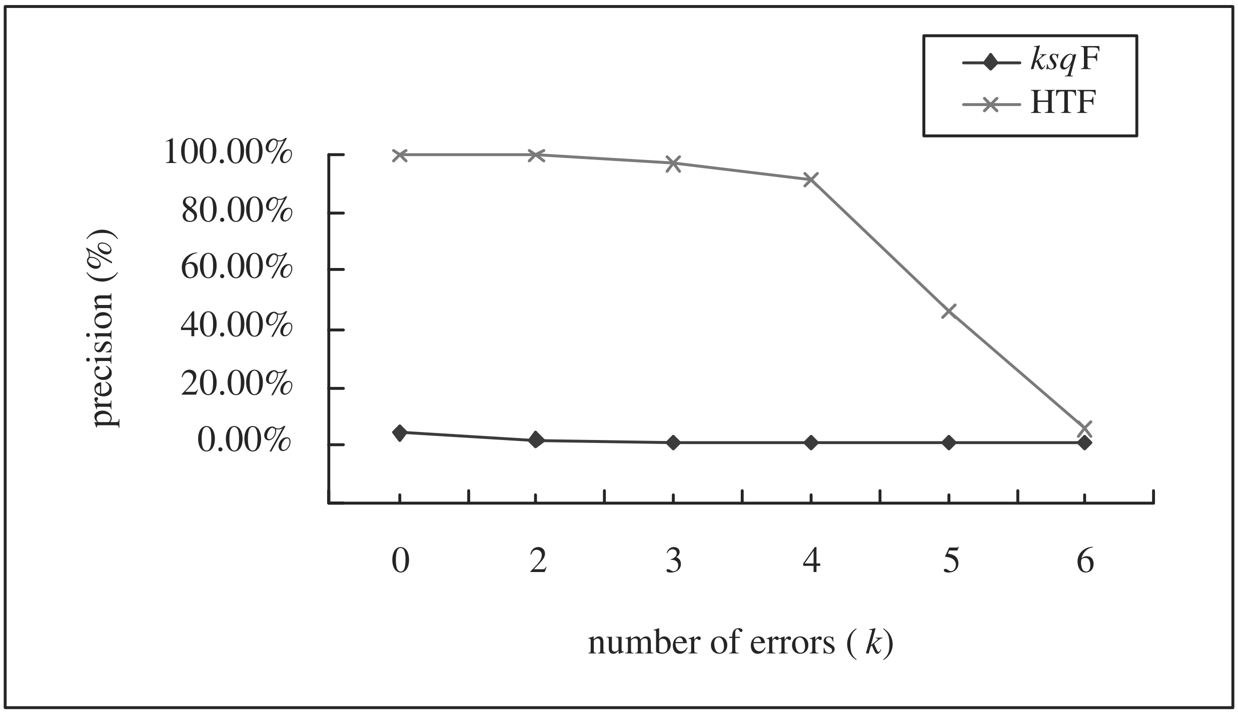
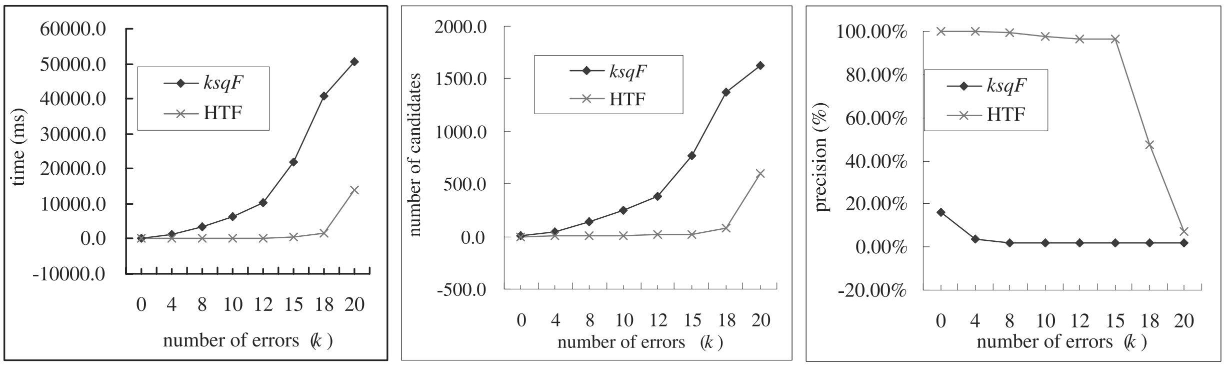


Figure 29: A comparison of the precision under short query patterns with length 30

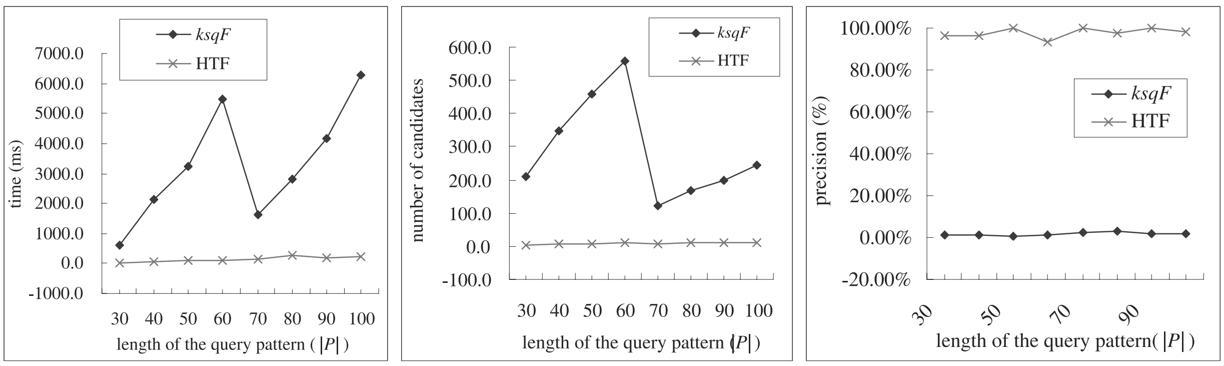
$(m/k) = 10\%$, where the lengths of the query sequences are from 30 to 100. Figure 30-(c) shows the comparisons under a high error level, $(m/k) = 20\%$, where the lengths of the query sequences are from 30 to 100. From these results, we show that the hash trie filter is more efficient than the $(k + s)$ q-samples filter in terms of the response time under long query patterns, low error levels, and high error levels for these experiments. Moreover, the precision of the hash trie filter is higher than that of the $(k+s)$ q-samples filter. The reason is that the number of candidates of the hash trie filter is fewer than that of the $(k + s)$ h-samples filter. Therefore, the verification time of the hash trie filter is also shorter than that of the $(k + s)$ q-samples filter.

In Figures 30-(a) and 30-(b), the numbers of candidates generated by our hash trie filter (HTF) are very small, and the precision of the hash trie filter is near to 100%. The reason is that the hash trie filter dynamically adjusts the value of s (*i.e.*, the least number of matching subpatterns). In these two experiments, the values of s are large enough to make almost only the truly-matching subsequences pass our checking conditions for candidates. Therefore, the hash trie filter could generate candidates more precisely than the $(k + s)$ q-samples filter.

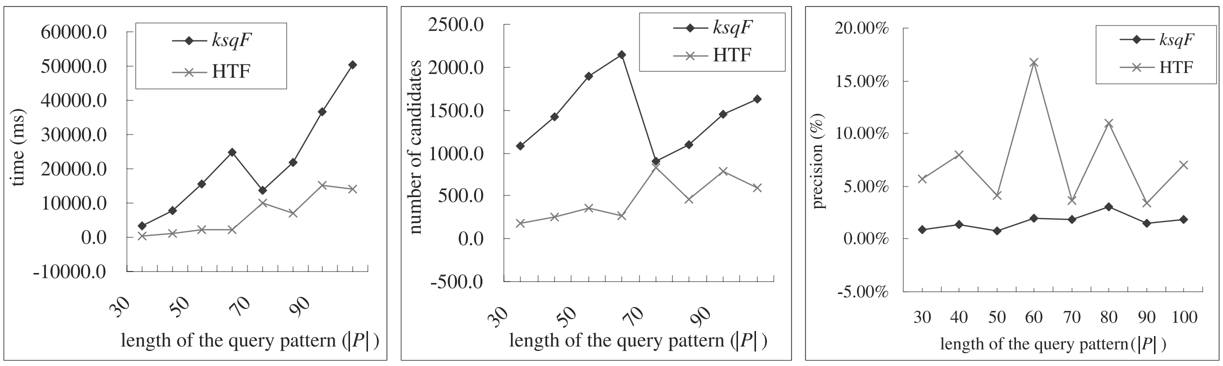
On the other hand, in Figure 30-(a), the precision of the $(k+s)$ q-samples filter decreases quickly when the number of errors increases from 0 to 4, while in Figures 30-(b) and 30-(c), the precision seems to be more stable. The reason is that the precision is mainly affected by the error level, *i.e.*, $e = k/m$. In Figure 30-(a), as the error level increases, the precision decreases. In Figures 30-(b) and 30-(c), the error level is set to a fixed value. Therefore, the precision of the $(k + s)$ q-sample filter seems to be stable in Figures 30-(b) and 30-(c). Note that in these experiments shown in Figures 30, the precision of the $(k + s)$ q-sample filter is near to 0%. The reason is that in these experiments, the number of truly-matching subsequences in the DNA sequence is very small. (We could see that as shown in Figures 30-(a) and 30-(b), the numbers of candidates generated by the proposed hash trie filter are near to 0.) Since the $(k + s)$ q-sample filter generates too many candidates, it precision is near to 0%.



(a)



(b)



(c)

Figure 30: Comparisons of performance between the $(k+s)$ q-samples filter and the hash trie filter: (a) under long query patterns with length 100; (b) under a low error level, $(m/k) = 10\%$; (c) under a high error level, $(m/k) = 20\%$

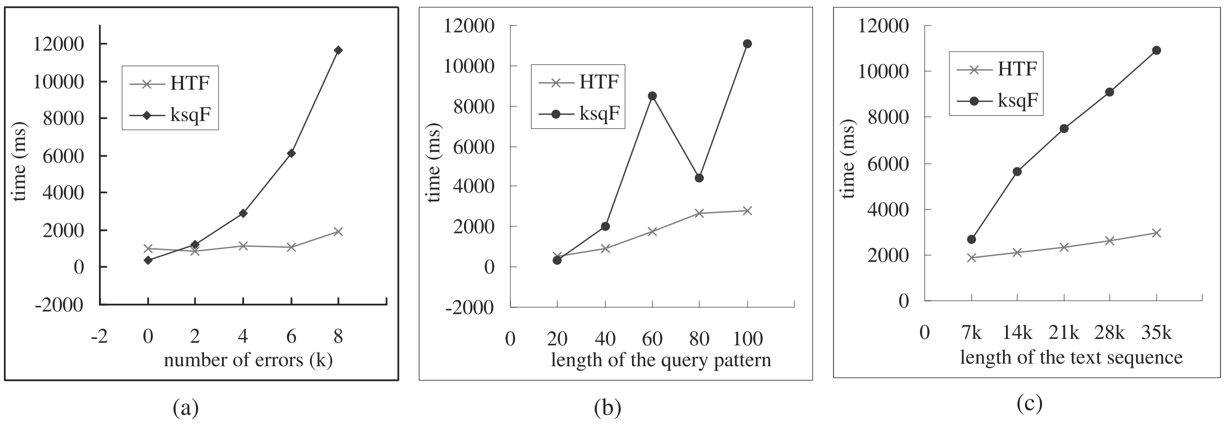


Figure 31: Comparisons of performance between the $(k + s)$ q-samples filter and the hash trie filter: (a) varying the value of k; (b) varying the length of the query pattern; (c) varying the length of the database sequence

4.5 Performance Under Long Database Sequences

In this subsection, we use contig "NT_167199.1" as the database sequence. Contig "NT_167199.1" is a DNA sequence of length 34821 base pairs, which is about 4.5 times the length of gene "Z73521" used in previous experiments. Figure 31 shows the experimental results. Figure 31-(a) shows the experimental result under query patterns with length 50 and varying the value of k. Figure 31-(b) shows the experimental result under an error level of 10% and varying the length of the query pattern. Figure 31-(c) shows the result under query patterns with length 100 and an error level of 10%, where we vary the length of the database sequence by cutting this DNA sequence into 1/5, 2/5, 3/5, and 4/5 of the original sequence.

From Figure 31, we also show that the hash trie filter is more efficient than the $(k+s)$ q-samples filter in terms of the response time. In Figures 31-(a) and 31-(b), the experimental results are similar to those results based on gene "Z73521" in previous experiments. In Figure 31-(c), we could observe that as the length of the database sequence increases, the difference of the response time between the $(k + s)$ q-samples filter and the hash trie filter also increases. The reason is that as the length of the database sequence increases, the number of possible candidates contained in this sequence also increases. In this case, both filters generate more candidates. However, the hash trie filter generates candidates more strictly than the $(k+s)$ q-samples filter. Therefore, the response time of the hash trie filter is also shorter than that of the $(k + s)$ q-samples filter.

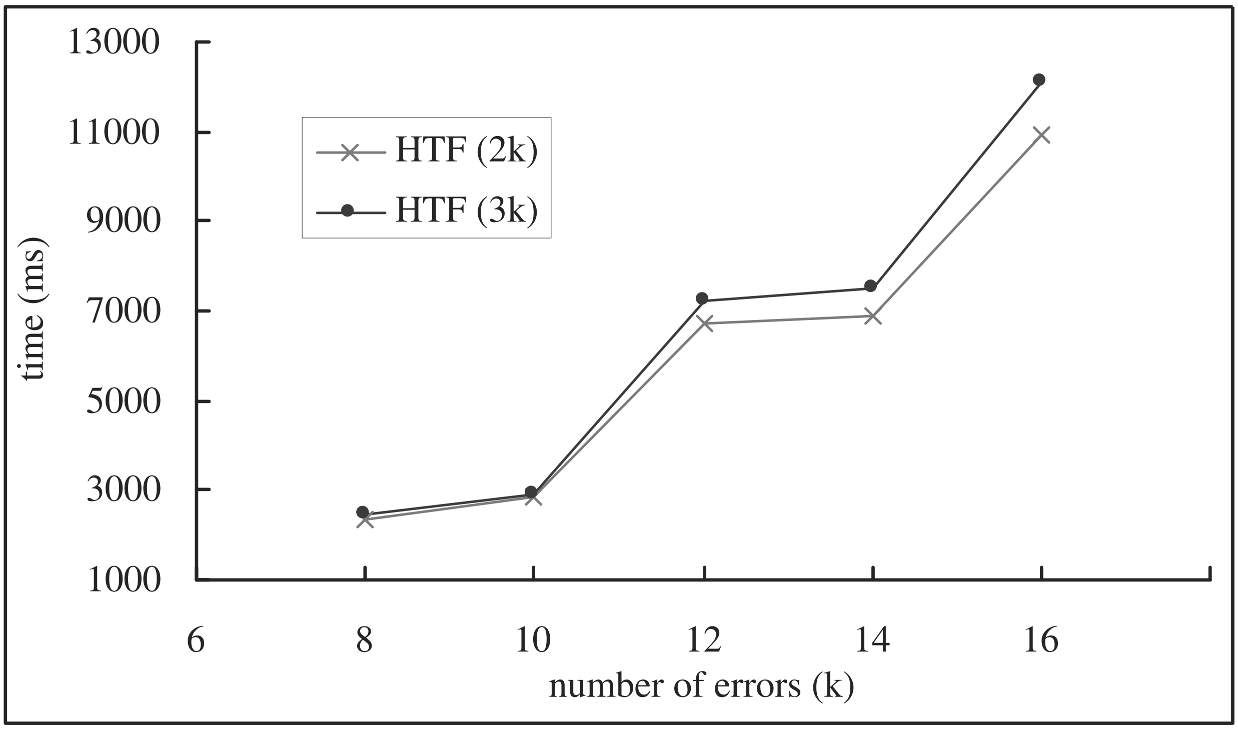


Figure 32: A comparison of the response time between the original hash trie filter and the modified hash trie filter

4.6 Improvement on the Length of a Candidate

As we mentioned in Subsection 3.2.4, the length of a candidate in the hash trie filter is $(m + 2k)$, while it is about $(m + 3k)$ in the $(k + s)$ q-samples filter. In this subsection, we study the improvement on the length of a candidate. We modify the length of a candidate in our hash trie filter to $(m+3k)$, and observe the difference of the response time between the original hash trie filter and the modified hash trie filter. Figure 32 shows the experimental result, where “HTF ($2k$)” is the original filter and “HTF ($3k$)” is the modified filter. From this figure, we observe that as the value of k increases, the difference of the response time between “HTF ($2k$)” and “HTF ($3k$)” also increases. The reason is that as we mentioned in Subsection 3.2.4, the length of a candidate will affect the time for verifying this candidate by dynamic programming. Even if the numbers of candidates generated by “HTF ($2k$)” and “HTF ($3k$)” are the same, “HTF ($2k$)” needs shorter response time than “HTF ($3k$)”.

4.7 Analysis

In this subsection, for the proposed hash trie filter, we analyze the maximal number of allowed errors, k, under the combinations of various parameters, including: (1) threshold w of the least number of matching subpatterns, (2) the length of query patterns, m, and (3) the related length of a subpattern, q'.

Basically, when a query is inquired, first, we will compute the length q' of a subpattern by using the formula of the $(k+s)$ q-samples filter, *i.e.*, $q' = \lceil \log_\sigma m \rceil$. Then, our method will dynamically decide the needed number of matching subpatterns (*i.e.*, s) which also satisfy the global order, according to the number of allowed errors (*i.e.*, k). We let $NumSP = \lfloor m/q' \rfloor$ and $s = NumSP - k$. As the value of k decreases, the value of s and the precision will also increase. On the other hand, as the value of k increases, the value of s will decrease. When s is smaller than threshold w, for example, $w = 2$ in our simulation, we will reduce the length of each gram, *i.e.*, q', by 1, to increase the number of subpatterns, s. Therefore, the value of s will also increase. In other words, one of the differences between our method and the $(k+s)$ q-samples filter is that we will take the number of allowed errors, k, into consideration to dynamically decide the value of s, so that the usability and the precision of the hash trie filter will increase.

Figure 33 shows the analysis of the maximal allowed values of k, denoted as Max. k, under various parameters w and m, and the original values of q'. Note that the empty cells in Figure 33 mean that the hash trie filter can not work under those circumstances. From this figure, we observe that under the same value of m, when the value of w increases, Max. k will decrease. This means that the usability of the hash trie filter will be restricted if we do not dynamically adjust the value of q'. Figure 34 shows the analysis of the maximal allowed values of k under various parameters w and m, and adjusted values of q', where the value of each q' is shrunken by 1. From this figure, we show that under the same value of m, Max. k becomes larger than that in Figure 33. This is because as the value of q' decreases, the number of subpatterns, $NumSP$, will increase. Therefore, the maximal allowed value of k (*i.e.*, Max. k) will also increase.

Max. *k*		threshold *w*																							
m	*q'*	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
10	2	4	3	2	1																				
20	3	5	4	3	2	1																			
30	3	9	8	7	6	5	4	3	2	1															
40	3	12	11	10	9	8	7	6	5	4	3	2	1												
50	3	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1									
60	3	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1					
70	4	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1								
80	4	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1					
90	4	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1			
100	4	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1

Figure 33: The maximal allowed values of k under various parameters w and m, and the original values of q'

Max. *k*		threshold *w*																															
m	*Min q'*	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
10	1	9	8	7	6	5	4	3	2	1																							
20	2	9	8	7	6	5	4	3	2	1																							
30	2	14	13	12	11	10	9	8	7	6	5	4	3	2	1																		
40	2	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1													
50	2	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1								
60	2	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1			
70	3	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1										
80	3	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1							
90	3	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1			
100	3	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1

Figure 34: The maximal allowed values of k under various parameters w and m, and adjusted values of q' (denoted as Min. $q'(= q' - 1)$)

5 Conclusion

For DNA sequence databases, approximate string matching is widely utilized by molecular biologists. In this paper, we have given the definition of a hash trie and proposed a new filter method, the hash trie filter, for approximate string matching in DNA sequence databases. A hash trie is a tree-like data structure with leaf nodes storing the positions of q-grams split from text X. The occurring positions in text X of every subpattern split from query pattern P can be found by traversing the hash trie. Then, we have proposed a new way to efficiently find the ordered subpatterns. Our method utilizes several techniques to reduce the number of candidates, and could provide the same results as other approximate string matching methods. From the experimental results, we have shown that the response time of the hash trie filter is shorter than that of the $(k + s)$ q-samples filter. The number of processing candidates of the hash trie filter is also fewer than that of the $(k + s)$ q-samples filter. Moreover, the precision of the hash trie filter is higher than that of the $(k + s)$ q-samples. Furthermore, we have shown that our method could provide better performance than the $(k+s)$ q-sample filter under high error levels, long query patterns, or long database sequences.

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.

References

[1] Altschul SF, Gish W, Miller W, Myers EW, Lipman DJ (1990) Basic local alignment search tool. J Mol Biol 215(3):403–410

[2] Baeza-Yates R, Gonnet G (1992) A new approach to text searching. Commun ACM 35(10):74–82

[3] Chang W, Lawler E (1994) Sublinear approximate string matching and biological applications. Algorithmica 12(4):327–344

[4] Chang W, Marr T (1994) Approximate string matching and local similarity. In: 5th annual symposium on combinatorial pattern matching, pp 259–273

[5] Dobrišek S, Žibert J, Pavešić N, Mihelič F (2009) An edit-distance model for the approximate matching of timed strings. IEEE Trans Pattern Anal Mach Intell 31(4):736–741

[6] Farach-Colton M, Landau GM, Sahinalp SC, Tsur D (2007) Identification of common molecular subsequences. J Comput Syst Sci 73(7):1035–1044

[7] Friedberg EC, Walker GC, Siede W (1995) DNA repair and mutagenesis. American Society Microbiology, America

[8] Houle JL, Cadigan W, Henry S, Pinnamaneni A, Lundahl S (2000) Database Mining in the Human Genome Initiative. Available at: http://www.biodatabases.com/whitepaper01.html. Accessed 2 Sept. 2009

[9] Hunt E, Atkinson MP, Irving RW (2001) A database index to large biological sequences. In: 27th conference on very large databases, pp 139–148

[10] Hunt E, Atkinson MP, Irving RW (2002) Database indexing for large DNA and protein sequence collections. VLDB J 10(1):256–271

[11] Hyyro H, Pinzon Y, Shinohara A (2005) Fast bit-vector algorithms for approximate string matching under indel distance. In: 31st annual conference on current trends in theory and practice of informatics, pp 380–384

[12] Karkkainen J, Na JC (2007) Faster filters for approximate string matching. In: Workshop on algorithm engineering and experiments, pp 1–7

[13] Lee HP, Tsai YT, Tang CY (2004) A seriate coverage filtration approach for homology search. In: ACM symposium on applied computing, pp 180–184

[14] Lipman DJ, Pearson WR (1985) Rapid and sensitive protein similarity searches. Science 227(4693):1435–1441

[15] Ma B, Tromp J, Li M (2002) PatternHunter: faster and more sensitive homology search. Bioinformatics 18(3):440–445

[16] Mazeika A, Böhlen MH, Koudas N, Srivastava D (2007) Estimating the selectivity of approximate string queries. ACM Trans Database Syst 32(2):1–40

[17] Myers G (1999) A fast bit-vector algorithm for approximate string matching based on dynamic programming. J ACM 46(3):395–415

[18] Navarro G (1997) Multiple approximate string matching by counting. In: 4th south American workshop on string processing, pp 95–111

[19] Navarro G (2001) A guided tour to approximate string matching. ACM Comput Surv 33(1):31–88

[20] Navarro G, Sutinen E, Tanninen J, Tarhio J (2000) Indexing text with approximate q-grams. In: 11th annual symposium on combinatorial pattern matching, pp 350–363

[21] Smith TF, Waterman MS (1995) Identification of common molecular subsequences. J Mol Biol 147(1):195–197

[22] Sutinen E, Tarhio J (1995) On using q-gram locations in approximate string matching. In: 3th annual european symposium on algorithms, pp 327–340

[23] Sutinen E, Tarhio J (1996) Filtration with q-samples in approximate string matching. In: 7th annual symposium on combinatorial pattern matching, pp 50–63

[24] Sutinen E, Tarhio J (2004) Approximate string matching with ordered q-grams. Nord J Comput 11(4):321–343

[25] Takaoka T (1994) Approximate pattern matching with samples. In: 5th international symposium on algorithms and computation, pp 234–242

[26] Ukkonen E (1985) Finding approximate patterns in strings. J Algorithms 6(1):132–137

[27] Ukkonen E (1992) Approximate string matching with q-grams and maximal matches. Theor Comput Sci 92(1):191–211

[28] Williams HE, Zobel J (2002) Indexing and retrieval for genomic databases. IEEE Trans Knowl Data Eng 14(1):63–78