A Fast Algorithm for Approximate String Matching on Gene Sequences

Zheng Liu[1], Xin Chen[1], James Borneman[2], and Tao Jiang[1]

[1] Department of Computer Science, University of California, Riverside
[2] Department of Plant Pathology, University of California, Riverside

Abstract. Approximate string matching is a fundamental and challenging problem in computer science, for which a fast algorithm is highly demanded in many applications including text processing and DNA sequence analysis. In this paper, we present a fast algorithm for approximate string matching, called FAAST. It aims at solving a popular variant of the approximate string matching problem, the *k-mismatch problem*, whose objective is to find all occurrences of a short pattern in a long text string with at most k mismatches. FAAST generalizes the well-known Tarhio-Ukkonen algorithm by requiring two or more matches when calculating shift distances, which makes the approximate string matching process significantly faster than the Tarhio-Ukkonen algorithm. Theoretically, we prove that FAAST on average skips more characters than the Tarhio-Ukkonen algorithm in a single shift, and makes fewer character comparisons in an entire matching process. Experiments on both simulated data sets and real gene sequences also demonstrate that FAAST runs several times faster than the Tarhio-Ukkonen algorithm in all the cases that we tested.

1 Introduction

Approximate string matching is a fundamental and challenging problem in computer science. It is an operation that usually costs a large amount of computational resources. Therefore, a fast algorithm for approximate string matching is highly demanded in many applications including text processing and gene sequence analysis. There are two important variants of the approximate string matching problem: the k-mismatch problem and the k-difference problem. In both, we are given a short *pattern* string $P = p_1 p_2 \cdots p_m$ and a long *text* string $T = t_1 t_2 \cdots t_n$ over an alphabet Σ, and an integer k. The k-mismatch problem is to find all occurrences of the pattern P in the text T with at most k mismatches (*i.e.*, substitutions) allowed, whereas the k-difference problem finds all substrings of T with *edit distance* at most k to P. We are interested in the former problem in this paper.

There are many various algorithms dealing with the k-mismatch problem in the string matching literature. In 1992, Baeza-Yates and Gonnet [1] first proposed the Shift-Add algorithm for exact string matching, and then naturally generalized it to handle mismatches. El-Mabrouk and Crochemore [5] tackled the k-mismatch problem by incorporating the Boyer-Moore technique [3] into the Shift-Add algorithm. Two algorithms that are the most relevant to ours are those proposed by Tarhio and Ukkonen [9] and by Baeza-Yates and Gonnet [2]. Both can be considered as generalizations of the Boyer-Moore algorithm [3] for exact string matching, but they employ different methods to calculate shift distances. A basic principle of these algorithms is to skip as many characters as possible while not missing any pattern occurrence. The Baeza-Yates-Gonnet algorithm takes advantage of *the good suffix rule*. If the shifted pattern matches the pattern of the previous alignment with at most $2k$ mismatches, further comparisons are needed to check a possible match at the shifted position. Therefore, the shift distance is the minimum distance (> 0) so that the shifted pattern matches this pattern at the previous alignment with at most $2k$ mismatches. The Tarhio-Ukkonen algorithm instead takes advantage of *the bad character rule*. When more than k mismatches occur, the last $k + 1$ characters of the text in the current alignment need have at least one match after the pattern is shifted to the right. The shift distance is thus calculated as the minimum distance (> 0) so that the shifted pattern has at least one match to the last $k + 1$ characters of the text in the previous alignment.

In this paper, we present a fast approximate string matching algorithm, called FAAST, which further generalizes the Tarhio-Ukkonen algorithm. Instead of requiring at least one match in the last $k + 1$ characters of the text in the previous alignment, the new algorithm requires at least x matches in the last $k + x$ characters when calculating shift distances, where x is a small integer value (typically 2 or 3 in our experiments). Apparently, the new algorithm will be the Tarhio-Ukkonen algorithm if we define $x = 1$. Although it seems a trivial modification, FAAST could run significantly faster than the Tarhio-Ukkonen algorithm, as demonstrated in our experiments.

In the ongoing *Oligonucleotide Fingerprinting Ribosomal Genes* (OFRG) project [10], we have applied the FAAST algorithm to the *gene sequence acquisition problem*. The problem of gene sequence acquisition is, given a collection of gene DNA sequences and a primer, how to extract all the gene sequences that contain the primer sequence (allowing a few mismatches). From the computational point of view, it is equivalent to the k-mismatch problem. In some cases, however, there are degenerate characters (*i.e.*, representing more than one character in an alphabet) in the primer sequence. This problem has traditionally been tackled by constructing a *nondeterministic finite automata* (NFA) [7, 8], which unfortunately requires a long preprocessing time and a large amount of memory space. In this paper, we will propose a simple approach to deal with degenerate characters based on the FAAST algorithm.

The rest of the paper is organized as follows. The next section reviews the Tarhio-Ukkonen algorithm. In Section 3, we focus on discussing our new algorithm FAAST. Experiments on simulated data and real gene sequences are presented in Section 4, and some concluding remarks are given in Section 5.

2 The Tarhio-Ukkonen Algorithm

Based on the Boyer-Moore-Horspool (BMH) algorithm [6], the Tarhio-Ukkonen algorithm [9] generalizes both the right-to-left scanning of the pattern and the computation of shift distances to allow string matching with k-mismatches. The BMH algorithm always tries to match the text character above the rightmost character of the pattern no matter where a mismatch occurs during an alignment. Similarly, when there are more than k mismatches occur, the Tarhio-Ukkonen algorithm shifts the pattern to a position such that the rightmost $k + 1$ text characters in the previous alignment have at least one match. The shift distance is defined as the minimum one that satisfies the above condition.

Assume a substring $t_{j-k}...t_j$ of the text is aligned with the rightmost $k + 1$ characters $p_{m-k}...p_m$ of P, and a shift is needed. For each $i \in [m - k, m]$ and each $a \in \Sigma$, we denote by $d_k[i, a]$ the minimum distance between p_l and p_i such that $p_l = a$ and $l < i$. Precisely, for a given i, $d_k[i, a]$ is initially set as $m - k$ and then updated if a smaller distance value is found, that is

$$d_k[i, a] = min\{\{m - k\} \cup \{s|p_{i-s} = a, s \in [1, i - 1], a \in \Sigma\}\} \tag{1}$$

Similarly, denote by $d[t_{j-k}...t_j]$ the minimum distance to shift the pattern to a position so that there is at least one match in the text substring above $p_{m-k}...p_m$. Then, we have

$$d_k[t_{j-k}, t_{j-k+1}, ..., t_j] = min\{d_k[m - i, t_{j-i}], i \in [0, k]\} \tag{2}$$

The Tarhio-Ukkonen algorithm can solve the k-mismatch problem in expected time $O(kn(1/m-k+k/c))$, where c is the alphabet size [9]. In the following, we give a simple example to illustrate, how the Tarhio-Ukkonen works in an approximate string matching process. The example uses pattern $P =$ AAGTCGTAAC and text $T =$ AACTGTTAACTTGCGACTAG, with $k = 2$. The Tarhio-Ukkonen algorithm constructs a shift table of $d_k[i, a]$ for P by (1), as shown in Table 1. The first two shifts in the approximate matching process is detailed in Table 2.

3 Our Algorithm

FAAST – a fast algorithm for approximate string match – can find all occurrences of a pattern $P = p_1...p_m$ in a text string $T = t_1...t_n$ with up to k mismatches. In this section, we first describe the idea and implementation of the FAAST

Table 1. A shift table of $d_k[i, a]$ $(k = 2, m = 10, n = 20)$

position	A	C	G	T
8	6	3	2	1
9	1	4	3	2
10	1	5	4	3

Table 2. An example of running Tarhio-Ukkonen algorithm($k = 2, m = 10, n = 20$)

Text:	AACTGTTAACTTGCGACTAG	
Pattern:	AAGTCGTAAC	(Shift 1)
	AAGTCGTAAC	(Shift 2)
	AAGTCGTAAC	
Shift 1:	The 3rd mismatch occurs at the 3rd position of the pattern string. The shift distance is calculated based on the last 3 characters of the aligned text, *i.e.*, AAC. $d_k[AAC] = min\{d_k[8, A], d_k[9, A], d_k[10, C]\} = 1$	
Shift 2:	The 3rd mismatch occurs at the 7th position of the pattern string. The shift distance is calculated based on the last 3 characters of the aligned text, *i.e.*, ACT. $d_k[ACT] = min\{d_k[8, A], d_k[9, C], d_k[10, T]\} = 3$	

algorithm. Then, its correctness and efficiency are proved and analyzed. Finally, a special consideration is taken in FAAST to enable it to work for patterns with degenerate characters.

3.1 Algorithm Description

Note that, in the Tarhio-Ukkonen algorithm, the shift distance is calculated as the minimum one such that there exists at least one match when aligning the rightmost $k + 1$ text characters in the current alignment with the pattern after a shift. In order to achieve faster matching process, FAAST instead calculates the shift distance as the minimum one such that the rightmost $k + x$ characters of the current aligned text will have at least x matches after the shift. Here, x generally takes a small integer value, *e.g.*, two or three. An example will be given at the end of this subsection to demonstrate that FAAST generally skips more characters than the Tarhio-Ukkonen algorithm in a shift.

FAAST consists of a preprocessing step and a matching step, as the Tarhio-Ukkonen does. In the preprocessing step, FAAST will calculate the shift distances of all possible strings of length $k + x$ in the alphabet Σ and tabulate them in a table d_{kx}, as follows. First, given a pattern string $P = p_1p_2 \cdots p_m$ and a position i ($i \in [m - k - x + 1, m]$), we denote by $\mathcal{U}_{kx}[i, a]$ a set of distances between p_i and all occurrences of the character a to the left of p_i in P. That is,

$$\mathcal{U}_{kx}[i, a] = \{s | p_{i-s} = a, s \in [1, i - 1], a \in \Sigma\} \tag{3}$$

Then, given a string $t_{j-k-x+1} \cdots t_j$ of length $k + x$ and a shift distance l ($l \in [1, m - k]$), we define a set as

$$\mathcal{V}_{kx}[t_{j-k-x+1} \cdots t_j, l] = \{i | l \in \mathcal{U}_{kx}[m - i, t_{j-i}], i \in [0, k + x - 1]\} \tag{4}$$

In $\mathcal{V}_{kx}[t_{j-k-x+1} \cdots t_j, l]$, we store the position offsets (relative to t_j) of all characters in $t_{j-k-x+1} \cdots t_j$ that will be matched after shifting the pattern to the right by l characters. For example, if only t_j and t_{j-2} are matched after shifting

Algorithm 1: Computation of table d_{kx}

```
1.  for $a$ in $\Sigma$
2.    for $i := m$ downto $m-k-x+1$
3.      $\mathcal{U}_{kx}[i,a] := \{m-k\}$;                {Set initialization}
4.  for $i := m$ downto $m-k-x+1$
5.    for $s := i-1$ downto 1
6.      if $i-s < m-k$ then
7.        $\mathcal{U}_{kx}[i,p_s] := \mathcal{U}_{kx}[i,p_s] \cup \{i-s\}$;      {Set union}
8.  for each string $t_{j-k-x+1}\cdots t_j$ in $\Sigma$
9.    for $l := 1$ to $m-k$
10.     $|\mathcal{V}_{kx}[t_{j-k-x+1}\cdots t_j, l]| := 0$;     {Set size initialization}
11.   for $l := 1$ to $m-k$
12.     for $i := m$ downto $m-k-x+1$
13.       if $l \in \mathcal{U}_{kx}[i, t_{j-m+i}]$ then
14.         $|\mathcal{V}_{kx}[t_{j-k-x+1}\cdots t_j, l]| := |\mathcal{V}_{kx}[t_{j-k-x+1}\cdots t_j, l]| + 1$;
15.     if $|\mathcal{V}_{kx}[t_{j-k-x+1}\cdots t_j, l]| \geq min[x, m-k-l]$
16.       $d_{kx}[t_{j-k-x+1}\cdots t_j] := l$;
17.       break;                                  {Go to step 8}
```

the pattern to the right by l characters, the value of $\mathcal{V}_{kx}[t_{j-k-x+1}\cdots t_j, l]$ would be $\{0, 2\}$. Finally, the shift distance $d_{kx}[t_{j-k-x+1}\cdots t_j]$ can be calculated using the following formula, where $|\cdot|$ is the size of a set:

$$d_{kx}[t_{j-k-x+1}\cdots t_j]$$

$$= min\{l \,|\, |\mathcal{V}_{kx}[t_{j-k-x+1}\cdots t_j, l]| \geq min\{x, m-k-l\}, l \in [1, m-k]\} \qquad (5)$$

This formula guarantees that the minimum l distance is made such that either the current aligned rightmost $k+x$ text characters have at least x matches in the next alignment when all these text characters are aligned with the pattern after the shift, or the current aligned rightmost $k+x$ text characters have at least $m-l-k$ $(< x)$ matches in the next alignment when only $m-l$ $(< k+x)$ text characters are aligned with the pattern after the shift. Both cases requires no more than k mismatches in the new alignment between the $k+x$ text characters and the shifted pattern.

The details of the preprocessing algorithm are provided in pseudocode as Algorithm 1: Step 1-7 describes the construction of set $\mathcal{U}_{kx}[i,a]$. The details of the size calculation for set $\mathcal{V}_{kx}[t_{j-k-x+1}\cdots t_j, l]$ is covered in step 9-14, and the rest part of Algorithm 1 fills out the table $d_{kx}[t_{j-k-x+1}\cdots t_j]$. Note that the calculation of d_{kx} uses a pattern as input, but does not depend on any text.

In the matching step, we compare a pattern P with a text string T. We denote by h the index of a character that is currently scanned in T and by i the index of the corresponding character in P. $t_{j-k-x+1}...t_j$ refers to the text string that is aligned above $p_{m-k-x+1}...p_m$. The matching process remains similar to that in the Tarhio-Ukkonen algorithm except that, when there are more than k

mismatches occur, we look up a different table for shift distances. The details of this step are provided in pseudocode as Algorithm 2:

Algorithm 2: Approximate string matching

```
1.   $j := m$;
2.   while $j \leq n$ do begin
3.     $h := j$; $i := m$; $e := 0$;        {$e$: the number of mismatches}
4.     while $i > 0$ and $e \leq k$ do begin
8.       if $t_h \neq p_i$ then
9.         $e := e + 1$;
9.       $i := i - 1$; $h := h - 1$;
10.    end of while
11.    if $e \leq k$ then
12.      record the occurrence position $j$;
13.    $j := j + d_{kx}[t_{j-k-x+1} \cdots t_j]$;
14.  end of while.
```

An example is given to illustrate how a shift table d_{kx} is calculated and how an approximate string matching proceeds. We use the same pattern and text as those used in the previous section for the Tarhio-Ukkonen algorithm. The set of shift distances, *i.e.*, $\mathcal{U}_{kx}$, is listed in Table 3 and a part of the table $\mathcal{V}_{kx}[t_{j-k-x+1} \cdots t_j, l]$ is listed in Table 4. The first two shifts in the approximate matching process is detailed in Table 5. As we have seen earlier, distances of the first two shifts made by the Tarhio-Ukkonen algorithm are 1 and 3, whereas they are 7 and 7 by FAAST, respectively. Therefore, our algorithm FAAST can generally skip many more characters than the Tarhio-Ukkonen algorithm in a single shift. As a result, FAAST could significantly speed up the approximate string matching process, as proved theoretically in the next subsection.

Table 3. A set of shift distances of $\mathcal{U}_{kx}$ $(k = 2, x = 3, m = 10, n = 20)$

position	A	C	G	T
6	4,5	1	3	2
7	5,6	2	1,4	3
8	6,7	3	2,5	1,4
9	1,7,8	4	3,6	2,5
10	1,2,8,9	5	4,7	3,6

3.2 Algorithm Analysis

In this section, we discuss in detail the correctness of FAAST, its time and space complexity, the average shift distance, and the total character comparisons.

Correctness. We establish the correctness of FAAST by the following theorem.

Table 4. An example of $\mathcal{V}_{kx}[t_{j-k-x+1}\cdots t_j, l]$ $(k=2, x=3, m=10, n=20, l=[1..8])$

l	1	2	3	4	5	6	7	8
AAAAA	0,1	0		4	3,4	2,3	1,2	0,1
...								
GCGAC	1	2,3	4		0,2		1	1
...								
GTCGT			0,1,2,3,4			0,1		
...								
TTAAC	0	4	3		0	2	1,2	1
...								
TTTTT	2	1,4	0,3	2	1	0		

Table 5. An example of running FAAST$(k=2, x=3, m=10, n=20)$

```
   Text:   AACTGTTAACTTGCGACTAG
Pattern:   AAGTCGTAAC                (Shift 1)
                   AAGTCGTAAC        (Shift 2)
                           AAGTCGTAAC
```

Shift 1:	The 3rd mismatch occurs at the 3rd position of the pattern string. The shift distance is calculated based on the last 5 characters of the aligned text, *i.e.*, TTAAC. $d_{kx}[TTAAC]=7$
Shift 2:	The 3rd mismatch occurs at the 5th position of the pattern string. The shift distance is calculated based on the last 5 characters of the aligned text, *i.e.*, GCGAC. $d_{kx}[GCGAC]=7$

Theorem 1. *Given any alignment between P and $t_{j-m+1}...t_j$ in T, P can be shifted by $d_{kx}[t_{j-k-x+1}\cdots t_j]$ characters to the right without passing by any approximate occurrences of P in T.*

Proof. Denote by $p_{i-k-x+1}...p_i$ the substring of P that is aligned below $t_{j-k-x+1}$ $\ldots t_j$ after shifting $d_{kx}[t_{j-k-x+1}\cdots t_j]$ characters to the right. Note that P may be aligned with only a part of $t_{j-k-x+1}...t_j$, and we omit such cases in our proof just for simplicity. Assume that an occurrence of P is passed by during the shift. When aligning this occurrence with P, we have a substring of P, denoted by $p_{i'-k-x+1}...p_{i'}$, that is aligned below $t_{j-k-x+1}...t_j$, such that there are at most k mismatches in the alignment of $p_{i'-k-x+1}...p_{i'}$ with $t_{j-k-x+1}...t_j$, and $i < i'$. These lead to a contradiction to the definition of $d_{kx}[t_{j-k-x+1}\cdots t_j]$, and the theorem thus follows. □

Time and space complexity. In the preprocessing part, calculation of $\mathcal{U}_{kx}$ takes time $O(m(k+x))$ and space $O((m-k)(k+x)c)$, where c is the alphabet size of Σ, which is 4 for DNA sequences. Meanwhile, it takes time $O(c^{k+x}(m-k)(k+x))$ and space $O(c^{k+x})$ to tabulate d_{kx}. Therefore, the total time spent on preprocessing is $O((k+x)((m-k)c^{k+x}+m))$ and space is $O(c^{k+x}+c(m-k)$

$(k+x))$). In the matching part, it needs $O(mn)$ time in the worst case. Instead, we are more interested in its performance in the average case.

Average shift distance. The average shift distance refers to the number of characters in the text that the pattern is expected to skip in one shift. Intuitively, the larger the average shift distance is, the faster the approximate matching, and thus the better the algorithm performs.

We use the *random string assumption* in our analysis. It assumes that each character in P and T is independently chosen at random from the alphabet set Σ. Also, we assume that the probability that two characters give rise to a match is p. Under this assumption, we have

Lemma 1. *The probability P_{kx} for the last $k+x$ characters of P to have at least x matches in an alignment with T is $P_{kx} = 1 - \sum_{i=0}^{x-1} C_{k+x}^{i}(1-p)^{k+x-i}p^{i}$.*

Proof. Note that the probability, denoted as $P_{kx,i}$, of $k+x$ characters having exactly i matches in an alignment forms a binomial distribution, *i.e.*,

$$P_{kx,i} = C_{k+x}^{i}(1-p)^{k+x-i}p^{i} \tag{6}$$

By summing up $P_{k+x,i}$ with i from 0 to $x-1$, we obtain

$$P_{kx} = 1 - \sum_{i=0}^{x-1} C_{k+x}^{i}(1-p)^{k+x-i}p^{i} \tag{7}$$

□

We simplify the calculation of the average shift distance, without taking into account the effect of the limit length of a pattern. Therefore, the shift distance can take a value up to the infinity in the calculation, which provides an approximation to the real average shift distance.

Theorem 2. *The average shift distance E_{kx}^{d} of the algorithm is $E_{kx}^{d} \approx 1/P_{kx}$.*

Proof. We denote by $P_{s,kx}$ the probability that the shift distance s is taken. Then,

$$P_{s,kx} = (1-P_{kx})^{s-1}P_{kx}, \quad s > 0 \tag{8}$$

Therefore, we have

$$E_{kx}^{d} \approx \sum_{s=1}^{\infty} sP_{s,kx} = \sum_{s=1}^{\infty} s(1-P_{kx})^{s-1}P_{kx} \quad = 1/P_{kx} \tag{9}$$

□

In the following, k is set to be 3, as used in our gene sequence analysis application [10]. We plot the curves of the average shift distances against the character matching probability p, in Fig. 1(a). Different values of x are employed, including one, which is used in the Tarhio-Ukkonen algorithm. As shown in the figure, the average shift distances become much larger as we increase x from 1 to 3, for small values of p. Therefore, FAAST can provide a very fast approximate matching process, in particular for gene DNA sequences where p is about 0.25 (the alphabet size is 4).

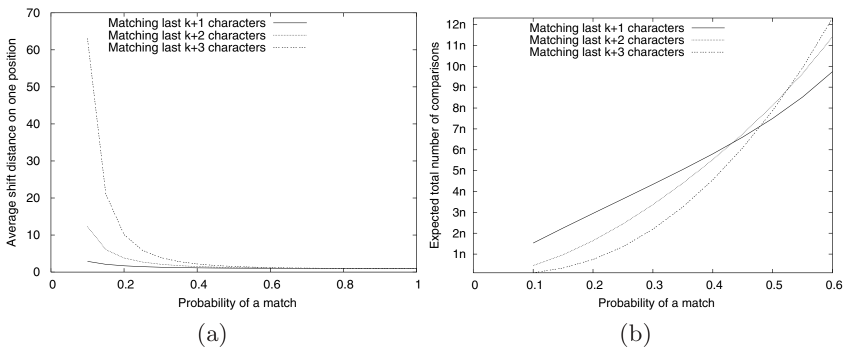

Fig. 1. (a) Comparison of average shift distances under different numbers of matches in the last $k + x$ characters, where $k = 3$. (b) The expected total number of character comparisons under different numbers of matches in the last $k + x$ characters, where $k = 3$

Total character comparisons. The total number of character comparisons made in an entire matching process is proportional to the running time of the program. Therefore, it is a very useful criterion to measure the performance of a string matching algorithm. Here, we again use the random string assumption and ignore the effect of the limit length of a pattern, as discussed above, to simplify our calculations.

Lemma 2. *The expected number E^c_{kx} of comparisons made between two successive shifts, is $E^c_{kx} \approx (k + x)/(1 - p)$.*

Proof. Note that the matching process between two successive shifts does not terminate until we find the $(k+1)^{th}$ mismatch in an alignment. As discussed in [9], the distribution of $E^c_{kx} - (k + x)$ converges to a negative binomial distribution when the pattern size increases to infinity. The expected value of $E^c_{kx} - (k + x)$ under this distribution is $(k+x)p/(1-p)$. That is, $E^c_{kx} - (k+x) \approx (k+x)p/(1-p)$, and thus the lemma follows. $\square$

By the above lemma and Theorem 3, we can easily obtain

Theorem 3. *The expected total number TE^c_{kx} of character comparisons made for a text of length n is $TE^c_{kx} \approx nP_{kx}(k + x)/(1 - p)$.*

Fig. 1(b) shows the expected total number of comparisons TE^c_{kx} made under different values of x and p. We can see that, if more matches are required in the calculation of shift distances and if the character matching probability is $p < 0.5$, a large amount of comparisons could be saved in a string matching process. For example, given a gene DNA sequence of length n and $p = 0.25$, a total of $2.29n$ character comparisons will be saved as x increases from 1 to 3.

3.3 Degenerate Characters

In many applications, we need to find in a text all occurrences of a pattern string that contains degenerate characters. For example, in the gene sequence acquisition problem, degeneracy in a DNA sequence refers to the phenomenon that one character may represent several nucleotide bases. In the official IUPAC-IUB single-letter base codes [4], R stands for G/A, Y for T/C, and H for A/C/T, *etc.* A naive method treats the pattern string with degeneracy as a set of multiple patterns, which makes string matching several times slower.

Degeneracy brings up two new issues for the FAAST algorithm. One is the new definition of match, and the other occurs in the calculation of shift distances. We consider two degenerate characters as a match if they share a common non-degenerate character, *e.g.*, R and H is a match when aligning them. When calculating shift distances, FAAST treats a degenerate character as any of its corresponding non-degenerate characters, and takes the minimum shift distance given by these non-degenerate characters as the shift distance of the degenerate character. The procedure in the matching step of FAAST remains unchanged. We notice that, in this way, FAAST will not miss any occurrence of pattern with at most k mismatches. Experiments on strings with degenerate characters are presented in the next section.

4 Experimental Results

We have tested FAAST on both simulated data sets and real DNA gene sequence data on a PC with Intel Pentium CPU (2.8GHz and 1G memory), and compared its performance with that of the Tarhio-Ukkonen algorithm [9].

To produce simulated data sets, we used a random generator to select four DNA bases {A, C, G, T} randomly with equal probabilities. Text sequences we tested are $2M$ (*i.e.*, two millions) bases long, and a pattern with 39 bases. We listed in Table 6 the average shift distances, total numbers of character comparisons, preprocessing time, and the total running time. The results show a clear tendency that, as x increases from one to seven, FAAST can shift by larger distances on average and make fewer comparisons. Though the preprocessing time is increasing, the total running time is consistently decreasing. For example, FAAST needs only 11.2 seconds with $x = 5$, whereas the Tarhio-Ukkonen algorithm (*i.e.*, $x = 1$) takes 210.2 seconds, which is about 18 times slower.

Table 6. The average shift distances, total character comparisons, preprocessing time, and total running time of FAAST on simulated DNA sequences of $2M$ bases. The pattern size is 39, and $k = 3$

x	1	2	3	4	5	6	7
Average shift distance	1.41	2.76	5.59	16.38	31.31	37.77	38.87
Total Comparisons($\times 2M$)	6.70	3.68	1.86	0.65	0.34	0.28	0.27
Total running time(sec)	210.2	114.4	58.1	20.6	11.2	10.8	16.7
Preprocessing time(sec)	0.01	0.01	0.03	0.08	0.36	1.58	6.90

To test the performance of our algorithm on real gene DNA sequence data, we downloaded 18,491 18S ribosomal fungal DNA sequences and 81,343 16S ribosomal bacterial DNA sequences from the NCBI DNA sequence database. We randomly picked 150 sequences to form a test set and compared the preprocessing time, string matching time and total running time of the Tarhio-Ukkonen algorithm with our generalized algorithm FAAST. This was repeated 5 times and the average result was reported. The bacterial sequence set includes totally 170K bases and the fungal sequence set includes 179K bases. The detailed results are shown in Table 7 and Table 8 for bacterial sequences and fungal sequences, respectively. For 150 random bacterial sequences, if we choose x as 5, our algorithm needs a total running time of 2.63 seconds whereas the Tarhio-Ukkonen algorithm (*i.e.*, $x = 1$) needs 18.78 seconds. Similarly, for 150 random fungal sequences, our algorithm needs only 5.62 seconds but the Tarhio-Ukkonen algorithm needs 16.45 seconds. Therefore, our algorithm with $x = 5$ runs about 7 times and 3 times faster than the Tarhio-Ukkonen algorithm on the bacterial and fungal sequences that we tested, respectively.

Table 7. The total running time, preprocessing time, and string matching time of FAAST on 150 bacterial DNA sequences with different x values and $k = 3$. The pattern used is AGRRTTTGATYHTGGYTCAG

x	1	2	3	4	5	6	7
Total running time(sec)	18.78	13.05	7.74	3.84	2.63	3.21	8.55
Preprocessing time(sec)	0.01	0.01	0.02	0.09	0.35	1.57	6.96
String matching time(sec)	18.77	13.04	7.72	3.75	2.28	1.64	1.59

Table 8. The total running time, preprocessing time, and string matching time of FAAST on 150 fungal DNA sequences with different x values and $k = 3$. The pattern used is TTAGCATGGAATAATRRAATAGGA

x	1	2	3	4	5	6	7
Running time(sec)	16.45	11.43	9.24	6.78	5.62	8.24	26.48
Preprocessing time(sec)	0.02	0.03	0.08	0.32	1.34	5.77	23.86
String matching time(sec)	16.43	11.40	9.16	6.46	4.28	2.47	2.62

5 Conclusion

FAAST has been embedded in a web-based system to enable biologists to build their own gene sequence databases. The whole system has been successfully used in the OFRG project [10]. The algorithm is designed especially for gene DNA sequences with an alphabet of size 4, to solve the gene sequence acquisition problem. As the alphabet size and the x value get large, we notice that the time and memory required for the shift distance calculation increase quickly, which in turn deteriorates the performance of FAAST. We plan to look into this problem in the future.

Acknowledgements

The research was supported in part by NSF grants DBI-0133265 and CCR-0309902, and National Key Project for Basic Research (973) grant 2002CB512801.

References

1. Baeza-Yates, R., Gonnet, G.H. (1992) A New Approach to Text Searching. *Communication of the ACM*, Vol. 35, No.10.
2. Baeza-Yates, R.A., Gonnet, G.H. (1994) Fast String Matching with Mismatches. *Information and Computation* 108, pp. 187-199.
3. Boyer, R.S., Moore, J.S. (1977) A fast string searching algorithm. *Communications of the ACM*, 10(20), pp. 762-772.
4. Cornish-Bowden, A. (1985) Nomenclature for incompletely specified bases in nucleic acid sequences: recommendations 1984, *Nucl. Acids Res*, 13, pp. 3021-3030.
5. El-Mabrouk, N., Crochemore, M. (1996) Boyer-Moore strategy to efficient approximate string matching. *LNCS*, Vol. 1075, Combinatorial Pattern Matching, pp. 24-38.
6. Horspool, R.N. (1980) Practical fast searching in strings. *Software - Practice and experience*, 10, 501-506.
7. Navarro, G. and Raffinot, M. (1999) Fast Regular Expression Search. *LNCS*, Vol. 1668, Proceedings of the 3rd International Workshop on Algorithm Engineering, pp. 198-212.
8. Navarro, G. (2003) Approximate Regular Expression Searching with Arbitrary Integer Weights. *LNCS*, Vol. 2906, Proceedings of ISAAC'03, pp. 230-239.
9. Tarhio, J., Ukkonen, E. (1993) Approximate Boyer-Moore String Matching. *SIAM J. Comput.*, 22, pp. 243-260.
10. Valinsky L., Scupham A., Vedova G.D., Liu Z., Figueroa A., Jampachaisri K., Yin B., Bent E., Mancini-Jones, R., Press J., Jiang T., and Borneman J. (2004) Oligonucleotide Fingerprinting of Ribosomal RNA Genes (OFRG), pp. 569-585. In G. A. Kowalchuk, F. J. de Bruijn, I. M. Head, A. D. L. Akkermans, J. D.van Elsas (ed.) Molecular Microbial Ecology Manual (2nd ed). Kluwer Academic Publishers, Dordrecht, The Netherlands.