An Efficient Algorithm for Finding Similar Short Substrings from Large Scale String Data

Takeaki Uno

uno@nii.jp, National Institute of Informatics
2-1-2, Hitotsubashi, Chiyoda-ku, Tokyo 101-8430, Japan

Abstract. Finding similar substrings/substructures is a central task in analyzing huge amounts of string data such as genome sequences, web documents, log data, etc. In the sense of complexity theory, the existence of polynomial time algorithms for such problems is usually trivial since the number of substrings is bounded by the square of their lengths. However, straightforward algorithms do not work for practical huge databases because of their computation time of high degree order. This paper addresses the problems of finding pairs of strings with small Hamming distances from huge databases composed of short strings. By solving the problem for all the substrings of fixed length, we can efficiently find candidates of similar non-short substrings. We focus on the practical efficiency of algorithms, and propose an algorithm running in almost linear time of the database size. We prove that the computation time of its variant is bounded by linear of the database size when the length of short strings to be found is constant. Slight modifications of the algorithm adapt to the edit distance and mismatch tolerance computation. Computational experiments for genome sequences show the efficiency of the algorithm. An implementation is available at the author's homepage[1]

1 Introduction

These days we have many huge string data such as genome sequences, web documents, log data, etc. Since the size of data is so huge that human cannot grasp them intuitively, they must be computationally analyzed. Finding similar substrings or similar substructures is an important way of analyzing the data. The similarity and distribution of substrings makes it possible to grasp the global or local structures. The number of substrings in a string is at most the square of the string length. Thus, if the distance between two substrings can be computed in polynomial time, similar substrings can be found in polynomial time by comparing all substrings one by one. However, polynomial time algorithms of high degree do not work for huge data, therefore practical fast algorithms are needed.

In the area of algorithms and computation, the problem of finding similar strings has been widely studied. The problem is usually formulated that for two given strings Q and S, find all substrings of S similar to Q. This formulation can be considered as a generalization of string matching problems. When Hamming

[1] http://research.nii.ac.jp/~uno/index.html

distance is chosen as a similarity measure, a straightforward algorithm solves the problem in $O(|S||Q|)$ time, thus a research goal is to reduce this time complexity. Here the length of S and Q is denoted by $|S|$ and $|Q|$.

For the problem of finding substrings of S with the shortest Hamming distance to Q, Abrahamson[1] proposed an algorithm running in $O(|S|(|Q|\log|Q|)^{1/2})$ time. If the maximum Hamming distance is k, the computation time can be reduced to $O(|S|(k\log k)^{1/2})$[4]. Some approximation approaches have been also developed. The Hamming distance of two strings of length l within $(1-\epsilon)$ and $(1+\epsilon)$ approximation ratio with probability δ can be computed in $O(\log l \log(1/\delta)/\epsilon)$ time [6]. For edit distance, which allows insertions and deletions, algorithms proposed by Muthukrishnan and Sahinalp[8, 9] approximate the minimum distance substring. Using these algorithms, the problem can be solved in shorter time but may fail with some solutions. These algorithms take more than $\mathrm{O}(|S|^2)$ time to find similar substrings even for fixed length strings, Thus direct application of these algorithms does not work in practice.

On the other hand, there are several studies for efficient data structures to find similar substrings. The problem is formulated such that, for a given string S, construct a data structure of not a large size such that for any query string Q, substrings of S similar to Q can be found in short time. For the problem of finding substring of S equal to Q, there are many efficient data structures such as suffix array which make it possible to find all such substrings in almost $O(|Q|)$ time. However, allowing the errors makes the problem difficult. Existing algorithms basically need $\theta(|S|)$ time in the worst case. This difficulty can be observed in many other similarity search problems, such as inner product of vectors, points in Euclidean space, texts and documents. Motivated by practical use, there have been many studies on approximation and heuristic approaches.

Yamada and Morishita [12] proposed an algorithm for computing a lower bound of the shortest Hamming distance from Q to a substring in S. The algorithm constructs a data structure in $\mathrm{O}(|S|\log|S|)$ time, then answers a lower bound in $\mathrm{O}(|Q|L)$ time for any Q, where L is a constant no greater than $|Q|$. They also proposed an efficient exact algorithm for strings with small alphabet such as genome sequences[13].

In bioinformatics area, the problem of finding substrings of two strings which are similar to each other is called homology search, and has been widely studied. Because of the huge size of genome sequences, developing exact algorithms running in short time is difficult thus many heuristic algorithms have been proposed. BLAST and FASTA[2, 3, 10] are widely used among these algorithms. The idea of BLAST is to find short substrings of S and Q that are equal and check whether there are similar substrings including them. This idea is based on the observation that two similar substrings may have common short substrings. Actually, if the Hamming distance between two strings is no more than 9% of their length, they always have common string of 10 letters. The disadvantage of this method is that when the strings are long, huge number of substrings are the same, thus a lot of comparisons must be made. Such frequently appearing strings can be considered as a kind of noise in practice, thus heuristic methods ignore

these strings in the interest of practical efficiency. Another method of solving the problem is to partition Q and S into many blocks[11]. Some statistics of the blocks are computed, for example the number of each letter in the blocks, which for pruning blocks will never be similar. Then a dynamic programming connects the blocks and produces candidates of long similar substrings. The idea is that long similar substrings are expected to be not so many.

In this paper, we focus on Hamming distance. For given a set $\mathcal{S}$ of strings of the same length l, our problem is to enumerate all pairs of similar strings in $\mathcal{S}$. We consider the case in which the length l is small, and propose a practically efficient algorithm. The idea of the algorithm is to classify the strings in several ways so that any two similar strings are in the same group for at least one classification. Only strings in the same group have to be compared, which reduces the cost of the comparison. Each string is partitioned into k blocks, then any two strings with Hamming distance at most d share at least $k-d$ blocks. Thus they are in the same group at least one classification based on combinations of $k-d$ of these blocks. By setting k to l, the Hamming distance of any two strings in the same group is at most d. Using this fact, the time complexity is bounded by $O((\sum_{i=0}^{d} {}_lC_i) \times (|\mathcal{S}| + dN)) = O(2^l(|\mathcal{S}| + dN))$, where N is the number of pairs to be output. Computational experiments show its practical efficiency.

Using the algorithm makes it possible to approach the problem of finding similar non-short substrings. We can observe that two non-short similar strings may have several short substrings with short Hamming distance. Thus, pairs of substrings including several such strings are candidates for similar substrings. This approach has a certain accuracy. For example, any two strings of 3,000 letters with Hamming distance of at most 290 includes at least three substrings of 30 letters with Hamming distance of at most two. Similar observation can be made for edit distance. We propose an algorithm for finding representative pairs of non-short substrings including certain similar short substrings. We compared the human genome and mouse genome by our algorithm. The computation is done in quite short time and we could see the homology structure figured out by the comparison.

2 Preliminary

Let Σ be an alphabet of letters, and a *string* be a sequence of letters. The *length* of a string S is the number of letters in S and is denoted by $|S|$. A sequence composed of no letter is also a string and is called an *empty string*. The length of an empty string is 0. The ith letter of a string S is written $S[i]$, and i is called the *position* of $S[i]$. The substring of S starting from the ith letter and ending at the jth letter is denoted by $S[i, j]$. For example, when string S is $ABCDEFG$, $S[3] = C$, and $S[4, 6] = DEF$. When $j < i$, we define $S[i, j]$ by the empty string. For two strings S_1 and S_2, the *concatenation* of S_2 to S_1 is a string S given by concatenating S_2 to S_1, i.e., $|S| = |S_1| + |S_2|$, $S[i] = S_1[i]$ if $i \leq |S_1|$, and $S_2[i - |S_1|]$ otherwise. The concatenation of S_2 to S_1 is denoted by $S_1 \cdot S_2$.

For two strings S_1 and S_2 of the same length, the *Hamming distance* of S_1 and S_2 is defined by the number of positions i satisfying that $S_1[i] \neq S_2[i]$. The Hamming distance is denoted by $HamDist(S_1, S_2)$. Such letters are called the *mismatch* of S_1 and S_2, and the positions of mismatches are called *mismatch positions* of S_1 and S_2. For string S and integers i and k, $i \leq k$, we denote the substring of S starting from $(\lceil |S|(i-1)/k \rceil + 1)$th letter to $(\lceil |S|i/k \rceil)$th letter, i.e., $S[\lceil |S|(i-1)/k \rceil + 1, \lceil |S|i/k \rceil]$, by $B(S, k, i)$. $B(S, k, i)$ is called the *ith block*.

For a string S, the *deletion* of the position i is a string given by $S[1, i-1] \cdot S[i+1, |S|]$. The *insertion* of letter a to S at position i is a string given by $S[1, i-1] \cdot A \cdot S[i, |S|]$ where A is the string composed of one letter a. The *change* of position i of S to a is a string given by $S[1, i-1] \cdot A \cdot S[i+1, |S|]$. For two strings S_1 and S_2, the *edit distance* of S_1 and S_2 is the smallest number of combinations of insertion, deletion and change needed to transform S_1 to S_2.

The problem we address in this paper is formulated as follows. Let $\mathcal{S}$ be a multi set of strings of the same length. $\mathcal{S}$ is allowed to include more than one same string, and every string has an ID to be distinguished from the others. The problem is formulated as follows.

Short Hamming Distance String Pair Enumeration Problem
Input: A multi set $\mathcal{S}$ of strings of fixed length l, threshold value d
Output: All pairs of strings S_1 and S_2 such that $HamDist(S_1, S_2) \leq d$.

Hereafter we fix the input set $\mathcal{S}$ of strings of length l and a threshold value d.

3 Multi-Classification Algorithm

The basic idea of the algorithm is to classify the strings in several ways so that any two similar strings are in the same group at least once. Let $C(k, j)$ be the set of j distinct integers taken from $1, \ldots, k$. For example, $C(4, 2) = \{\{1, 2\}, \{1, 3\}, \{1, 4\}, \{2, 3\}, \{2, 4\}, \{3, 4\}\}$. For a string S and a set $C = \{i_1, ..., i_{k-d}\}$, $i_j < i_{j+1}$ taken from $C(k, k-d)$, we define $Sig(S, C) = B(S, k, i_1) \cdot B(S, k, i_2) \cdot \ldots \cdot B(S, k, i_{k-d})$. We suppose that an integer $k, d < k \leq l$ is chosen, and have a look at the following property.

Lemma 1. *If $HamDist(S_1, S_2) \leq d$, at least one $C \in C(k, k-d)$ satisfies $Sig(S_1, C) = Sig(S_2, C)$.*

Proof. The statement is obvious from the pigeonhole principle. Suppose that $HamDist(S_1, S_2) \leq d$. Observe that if $B(S_1, k, j) \neq B(S_2, k, j)$ holds, it includes at least one mismatch, i.e., $S_1[i] \neq S_2[i]$ holds for some i, $\lceil |S|(i-1)/k \rceil + 1 \leq i \leq \lceil |S|i/k \rceil$. Since S_1 and S_2 have at most d mismatches, at most d integers j satisfy $B(S_1, k, j) \neq B(S_2, k, j)$, thereby at least $k-d$ integers h satisfy $B(S_1, k, h) = B(S_2, k, h)$. Setting C to the set of those integers h satisfying $B(S_1, k, h) = B(S_2, k, h)$ shows that $Sig(S_1, C) = Sig(S_2, C)$. □

.

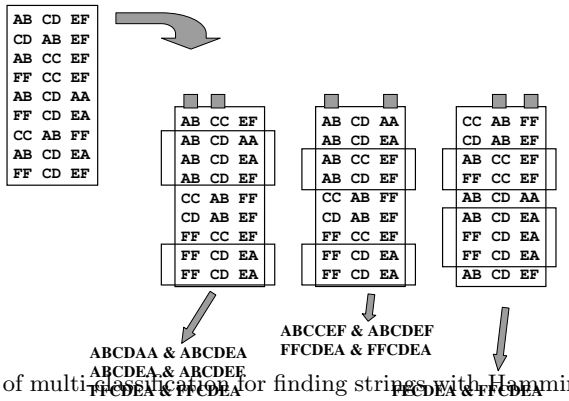

Fig. 1. Example of multi-classification for finding strings with Hamming distance of at most one, by dividing strings in three blocks and classifying them by two blocks.

This lemma motivates us to restrict the comparison to those pairs of strings satisfying the condition of the lemma. To efficiently find these pairs, we focus on the combinations of integers. For each $C \in C(k, k-d)$, we classify the strings S in $\mathcal{S}$ according to $Sig(S, C)$ so that two strings S_1 and S_2 satisfy $Sig(S_1, C) = Sig(S_2, C)$ if and only if they are in the same group. In Fig. 1, we show an example of this method, which we call the *multi-classification method*. In the example, there are nine strings and set $d = 1$ and $k = 3$. Each block is composed of two letters, and classifications by two blocks are done three times. For each classification there are several groups represented by rectangles with more than one strings, and some of them contain strings with Hamming distance of at most one, written at the head of the arrows.

ALGORITHM MultiClassification_Basic ($\mathcal{S}$:set of strings of length l, d)
1. choose k from $d+1, \ldots, l$
2. **for each** $C \in C(k, k-d)$ **do**
3. classify all strings $S \in \mathcal{S}$ by $Sig(S, C)$
4. **for each** group K of the classification
output all pairs S_1 and S_2 in K satisfying $HamDist(S_1, S_2) \leq d$
6. **end for**

The classification for C is done by sorting $Sig(S, C)$ in $O(l(k-d)/k \times |\mathcal{S}|)$ time by a radix sort. We compute the probability that two randomly chosen letters from strings of $\mathcal{S}$ are the same, and choose k such that the expected size of each group in a classification is less than 1. Then the comparisons for a group is not so many, and the bulk of the computation time is for radix sort. Since $l(k-d)/k$ is expected to be relatively small when l is small, it can be expected that the practical performance of the algorithm will be high.

3.1 Reducing the Cost for Radix Sort

Here we present a way to reduce the total computation time for radix sort by unifying the sort of the prefix of Sig. Suppose that we repeatedly and recursively add integers one by one to construct $C \in C(k, k-d)$ like a backtrack algorithm.

Then, after choosing i in some iteration of the backtracking, $B(S, k, i)$ is common to all C generated in the recursive call, i.e., until i is removed. Thus, the radix sort for $B(S, k, i)$ can be done at the iteration and the result can be used in the recursive calls. As a result, the computation time for each radix sort is reduced to $O(l/k \times |\mathcal{S}|)$. We describe the algorithm in the next subsection.

3.2 Avoiding Duplication without Memory

The multi-classification described above may output duplicates, i.e., output one pair of strings many times. For example, in Fig. 1, the pair FFCDEA and FFCDEA is output three times. A way to avoid such duplication is to store all the pairs found in memory and check the duplication when a new pair is found. Although this is simple, it requires a lot of memory. Here, we present a method that does not store found pairs and thus requires no extra memory.

A pair of strings S_1 and S_2 is output more than once if $B(S_1, k, i) = B(S_2, k, i)$ holds more than $k - d$ integers i. Then, $Sig(S_1, C) = Sig(S_2, C)$ holds for many C's. For given S_1 and S_2, let $C^*(S_1, S_2)$ be the lexicographically minimum one among $\{C' | C' \in C(k, k-d), Sig(S_1, C') = Sig(S_2, C')\}$. Our idea is to output an S_1 and S_2 pair only when the current operating C is equal to $C^*(S_1, S_2)$. Since, $C^*(S_1, S_2)$ is the collection of the $k - d$ smallest i's satisfying $B(S_1, k, i) = Sig(S_2, k, i)$, the computation is not a heavy task. The algorithm is the following which requires an initial call with $\mathcal{S}$, d and k, and set $C = \emptyset$.

ALGORITHM MultiClassification ($\mathcal{S}$:set of strings of length l, d, k, C)
1. **if** $|C| = k - d$ **then** output all pairs S_1 and S_2 in K
   satisfying $HamDist(S_1, S_2) \leq d$ and $C = C^*(S_1, S_2)$; **return**
2. **for each** i in $c+1, \ldots, (k-d)+(c-|C|)$ **do** (c = the maximum integer in C)
3. do a radix sort to classify all strings $S \in \mathcal{S}$ according to $B(S, k, i)$
4. **for each** group K of the classification with $|K| > 1$
   call MultiClassification (K, d, k, $C \cup \{i\}$)
5. **end for**

Theorem 1. *The computation time of algorithm MultiClassification except for step 1 is bounded by* $O(l/k \times |\mathcal{S}| \times_l C_d)$.

3.3 A Fixed Parameter Tractable Algorithm

The time complexity of the algorithm presented in the previous subsection is still $O(|\mathcal{S}|^2)$ since the bottle neck of the computation is actually step 1. For example, if all strings in $\mathcal{S}$ are the same, $HamDist(S_1, S_2)$ must be computed ${}_lC_d$ times for every S_1 and S_2 pair in $\mathcal{S}$, thereby the total computation time is $O(l|\mathcal{S}|(|\mathcal{S}| +_l C_d))$. Here we will save the computation time in step 1.

Let $k = l$. Then, for each i, $B(S, k, i)$ is composed of one letter, thus $Sig(S_1, C) = Sig(S_2, C)$ immediately means $HamDist(S_1, S_2) \leq d$. This implies that the Hamming distance does not have to be computed for any pair in each group. Another task in step 1 is avoiding duplications. We do this in another way.

Duplicate outputs occur when $HamDist(S_1, S_2)$ is strictly smaller than d. If $HamDist(S_1, S_2) = d$, exactly one $C \in C(k, k-d)$ satisfies $Sig(S_1, C) = Sig(S_2, C)$. This implies that without any check, we can output pairs with Hamming distance equal to d without duplications. Thus, we change d' from 0 to d and output only pairs with Hamming distance equal to d', we need no check for duplications. We call this algorithm the *complete version*. For the complete version of our algorithm, we obtain the following theorem. Note that the computation of $HamDist(S_1, S_2)$ is done in $O(d)$ time if $Sig(S_1, C) = Sig(S_2, C)$.

Theorem 2. *The short Hamming distance string pair enumeration problem for set $\mathcal{S}$ of strings of length l and distance threshold d can be solved in $O((\sum_{i=0}^{d} {}_lC_i) \times (|\mathcal{S}| + dN)) = O(2^l(|\mathcal{S}| + dN))$ time where N is the number of output string pairs.*

4 Approach to Long Substrings

In this section, we consider the problem of finding all pairs of substrings of a given string S that are similar to each other in some sense. In the sense of time complexity, the existence of polynomial time algorithms for this kind of problem is trivial since we have to compare only a polynomial number of pairs. However, in a practical sense, this problem is difficult since even if if we restrict the pars to be strings of the same length, $O(|S|^3)$ pairs of substrings must be compared. For huge strings the computation time must be quasi linear time, thus $O(|S|^3)$ time is far from practical efficiency.

Here we approach this problem with our algorithm. For a string S, distance threshold value d and length l, a pair of positions $(p, q), p \neq q$ is an *l-d seed* if $HamDist(S[p, p+l-1], S[q, q+l-1]) \leq d$. We can find all l-d seeds by giving all the substrings of S of length l to our multi-classification algorithm. One typical approach to capturing the similarity structures by using such seeds is as follows. We partition S into non-short blocks, for example, partition a string of 1,000,000 letters into 1,000 strings of 1,000 letters. We define the similarity measure of blocks $S[k_1, h_1]$ and $S[k_2, h_2]$ by the number of l-d seeds (p, q) satisfying $k_1 \leq p \leq h_1$ and $k_2 \leq q \leq h_2$. We can visualize the similarity structure of this measure by a figure such that the intensity of the color of the pixel (x, y) is given by the number of l-d seeds in xth block and yth block. The left of Figure 2 shows an example of pictures obtained by this method. If the blocks are large, any two blocks have a sufficiently large number of seeds, thus all pixels will be the same color. For large scale data, we need more precise method of deleting such noise.

A pair of positions $(p, q), p \neq q$ is a *normal l-d seed* if (p, q) is an l-d seed and p is a multiple of l. The normal l-d seeds can also be enumerated by our algorithm with shorter time than the usual l-d seeds. For a width threshold w and count threshold c, we say the pair of substrings $S[k_1, h_1]$ and $S[k_2, h_2]$ is a normal (w, c, l, d) candidate if there are distinct c normal l-d seeds (p, q) satisfying $k_1 \leq p \leq h_1$, $k_2 \leq q \leq h_2$ and $|p - q| \leq w$. A pair of similar substrings can be considered to be a normal (w, c, l, d) candidate for non-trivial w, c, l, and d. Especially if the Hamming distance of two substrings is short, they must

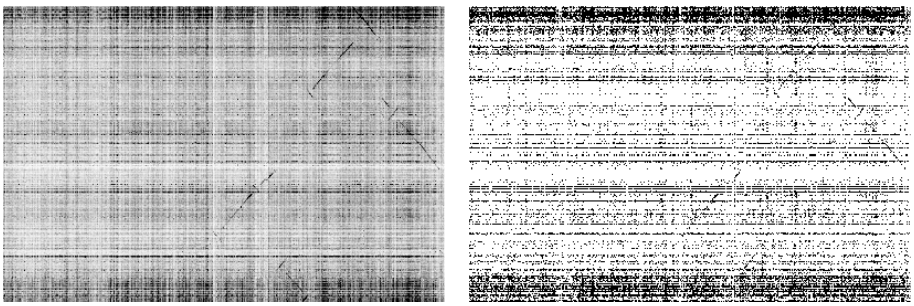

Fig. 2. Matrix showing similarity of mouse 11 chromosomes (X-axis) and Human 17 chromosome (Y-axis), with black cells on similar parts; we can see similar substructures as diagonal lines, but the figure is noisy because of the low resolution.

be a normal (w, c, l, d) candidate for a certain (w, c, l, d). For example, if the Hamming distance of two substrings of length 3000 is at most 290, they have to be a normal $(0, 3, 30, 2)$ candidate. If the edit distance of two substrings of length 3000 is at most 190 and has at most 50 of insertions and deletions, then they have to be a normal $(50, 3, 30, 2)$ candidate. Thus, we are motivated to enumerate all normal (w, c, l, d) candidates. However, for a set of c normal l-d seeds, there would be many normal (w, c, l, d) candidates including these seeds. Thus, the number of enumerated candidates can be large. Recall that the aim here is to find candidates of similar substrings, or to capture the similarity structures. Not many similar candidates are needed to represent one similar structure. Thus, here we propose a simple algorithm to output a set of pairs of substrings such that any normal (w, c, l, d) candidate is obtained by a slight modification of one of the pair.

For an integer z, we consider a slit of width $2w$. An l-d seed (p, q) is included in the slit of z if $z \leq p - q \leq z + 2w$. For each multiple z of w, we find all integers i such that there are at least c l-d seeds (p, q) included in the slit of z such that $i \leq p + q < i + a$ where a is a given length, and one of them satisfies $i = p + q$. For such integers i, the pair of substrings $S[(i + z + w)/2, (i + z + w)/2 + a]$ and $S[(i - z - w)/2, (i - z - w)/2 + a]$ is a desired pair. We output all such pairs. This requires sorting of all l-d seeds, but remaining process is very light and simple. We display a figure made by this approach in the right of Figure 2.

5 Applications and Extensions

In the practical applications there are many variants of similar string finding problems. In the following subsections we present several problems to which we can apply our multi classification algorithm.

5.1 Computing Mismatch Tolerance

In real world applications, we often need to find several unique short strings which are similar to no other strings. Such unique strings can be used as characterizations, invariants of string databases, or markers of substructures. A typical

application is in microarray. A microarray is a tool for biological experiments that can detect the existence of short strings, say 25 letters, in the genome sequence of a species or organizations. If a unique short substring in a gene sequence is known, the existence of the substring indicates the existence of the gene. To allow for experimental error, the substring has to have no similar substring.

When the Hamming distance is used, one of the uniqueness measure is called *mismatch tolerance*. The mismatch tolerance is the shortest Hamming distance to the other string. More precisely, for a set $\mathcal{S}$ of strings of the same length l, the mismatch tolerance of string S, denoted by $mis(S, \mathcal{S})$ is defined by $\min\{HamDist(S, S') \mid S' \in \mathcal{S} \setminus \{S\}\}$. If $mis(S, \mathcal{S})$ is large, S has no similar string in $\mathcal{S}$ in the sense of Hamming distance, thus our aim is to find the strings having not so small mismatch tolerance. Here we define our problem.

All Mismatch Tolerance Computing Problem
Input: for a set $\mathcal{S}$ of strings of the same length l, distance threshold d
Output: all $S \in \mathcal{S}$ such that $mis(S, \mathcal{S}) \leq d$

This problem can be solved by solving the short Hamming distance string pair enumeration problem. Actually, we do not have to output pairs, thus we do not check the duplications. Moreover, in the complete version of our algorithm, we have to execute the algorithm only for $d' = d$, and omit the computation of Hamming distance. Thus we obtain the following theorem.

Theorem 3. *The all mismatch tolerance computing problem for set $\mathcal{S}$ of strings of length l and distance threshold d can be solved in $O({}_lC_d|\mathcal{S}|) = O(2^l|\mathcal{S}|)$ time.*

5.2 General Edit Distance

In many studies and real world applications, the distance between two strings, genomes, and documents is evaluated by edit distance. The multi classification algorithm proposed above fails for edit distance since the position of the block shifts by the preceding insertions and deletions. For example, the edit distance between S_1 =ABCDEFGH and S_2 =ACDEFGHI is 2, obtained by deleting the second letter of S_1 and the eighth letter of S_2. By setting $k = 4$, the strings are partitioned into substrings of two letters. Although there are only two positions edited, no substrings in the partitions of S_1 and S_2 are the same, since the substrings in the middle are shifted by the deletion of the second letter.

For adapting to edit distance, we consider $\hat{C}(k, d)$ instead of $C(k, k-d)$ where $\hat{C}(k, d)$ is the set of $k - d$ signed or unsigned integers taken from 1 to d, i.e., $\hat{C}(k, d) = \{C \mid |C| = d, C \subseteq \{1, 1^+, 1^-, 2, 2^+, 2^-, \ldots, k, k^+, k^-\}\}$. i^+, i^- and i means an insertion, a deletion and a change at the ith block. For $C \in \hat{C}(k, d)$, let $\mathrm{sft}(C, i) = |\{j^+ \mid j < i, j^+ \in C\}| - |\{j^- \mid j < i, j^- \in C\}|$, and $Eq(C) = \{i \mid i, i^+, i^- \notin C\}$. We denote $S[\lceil |S|(i-1)/k \rceil + 1 + j, \lceil |S|i/k \rceil + j]$ by $\hat{B}(S, i, j)$. Then, for string S and $C \in \hat{C}(k, d)$, we define $\hat{Sig}(S, C)$ by $\hat{B}(S, i_1, \mathrm{sft}(C, i_1)) \cdot \hat{B}(S, i_2, \mathrm{sft}(C, i_2)) \cdot \ldots \cdot \hat{B}(S, i_{k-d}, \mathrm{sft}(C, i_{k-d}))$ where $Eq(C) = \{i_1, \ldots, i_{k-d}\}, i_j < i_{j+1}$. By using the terminology, we obtain the following lemma.

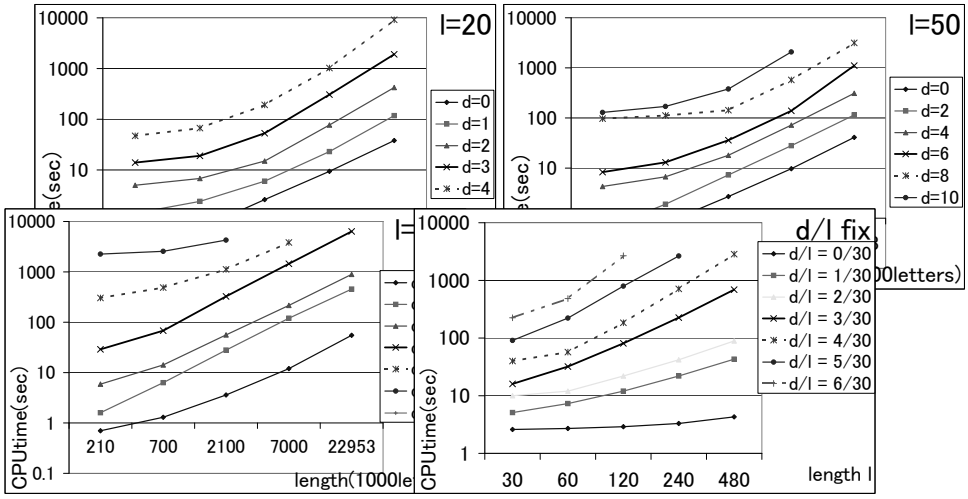

Fig. 3. Increase in computation time against the increase in database size with fixed l and d: the right-lower figure is for fixed d/l inputting a string of 2.1 million letters

Lemma 2. *If the edit distance between strings S_1 and S_2 is no more than d, at least one $C \in \hat{C}(k,d)$ satisfies $Sig(S_1, Eq(C)) = \hat{Sig}(S_2, C)$.*

The proof is omited by the page limit. Based on the lemma, we are motivated to classify all strings by $Sig(S, Eq(C))$ and $\hat{Sig}(S, C)$ for all $C \in \hat{C}(k,d)$ to obtain all the pairs of strings satisfying the condition of the lemma. By checking the edit distance for all pairs in each group classified, we can find all pairs of strings with edit distance at most d.

Theorem 4. *The computation time of algorithm MultiClassification modified to edit distance is bounded by $O(3^d l/k \times |\mathcal{S}| \times_l C_d)$, except for that for step 1.*

Theorem 5. *For set $\mathcal{S}$ of strings of length l and distance threshold d, we can find all pairs of strings with edit distance at most d in $O(3^d(\sum_{i=0}^{d} {}_lC_i)\times(|\mathcal{S}|+l^2N)) = O(2^l 3^d(|\mathcal{S}| + l^2N))$ time where N is the number of string pairs to be output.*

6 Computational Experiments

This section shows the results of computational experiments of the basic version of our algorithm. The code was written in C, and compiled with gcc. We used a note PC with a Pentium M 1.2GHz processor with 768 MB of memory, with cygwin which is a Linux emulator on Windows. The implementation is available at the author's homepage; http://research.nii.ac.jp/˜uno/index.html.

The instance is the set of substrings of fixed length taken from the Y chromosome of Homo sapiens. The length is set to $20, 50$ and 300. Figure 3 shows the results. Each line corresponds to one threshold value d. The X-axis is the number of input substrings, and Y-axis is the computation time. Both axes use

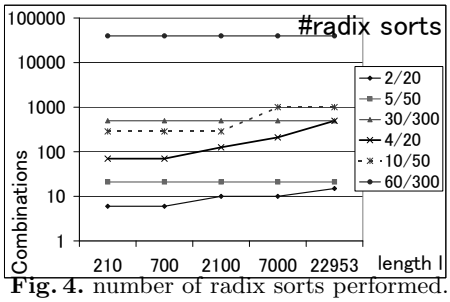

Fig. 4. number of radix sorts performed.

log scales. We can see that the computation time increases slightly higher than linear, but smaller than the square. Figure 4 shows the number of executed radix sorts. The number increases drastically if the number of mismatches increases, but does not increase much as the increase of input size.

We also show the increase in computation time against the increase of l with fixed d/l. The instance is fixed to that with 2.1 million strings, and the result is shown in the right-lower figure of Figure 3. From these results, at least for genome sequences our algorithm is quite scalable for the increase of input string.

7 Conclusion

We proposed an efficient algorithm for enumerating all pairs of strings with Hamming distance at most given d from string set $\mathcal{S}$. We focused on the practical efficiency of algorithms, and proposed an algorithm based on multiple classifications according to combinations of blocks of each string. We proved that the computation time of its variant is bounded by linear of the string length when the length of strings in the string set is constant. A simple modification of the algorithm adapts the edit distance, and computation of mismatch tolerance.

We also proposed a method of finding similar non-short substrings from huge strings. We modeled similar non-short strings by two non-short strings including several short similar substrings. We presented an efficient algorithm for finding those strings from huge strings. By the computational experiments for genome sequences, we demonstrated the practical efficiency of the algorithm. On the comparison of genome sequences, we could find similar long substrings from human and mouse genomes in a practically short time.

Acknowledgments

We gratefully thank to Professor Asao Fujiyama of National Institute of Informatics of Japan, Professor Shinichi Morishita of Tokyo University Doctor Takehiko Itoh of Mitsubishi Research Institute, and Professor Hidemi Watanabe of Hokkaido University, for their valuable comments. We would also like to thank to Professor Tsuyoshi Koide and Doctor Juzo Umemori of National Institute of Genetics for their contribution to the evaluation of the algorithm on practical genome problems.

References

1. K. Abrahamson, Generalized String Matching, *SIAM J. on Comp.*, **16(6)**, pp. 1039–1051, 1987.
2. S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman, Basic local alignment search tool, *J. Mol. Biol.* **215**, pp. 403–10, 1990.
3. S. F. Altschul, T. L. Madden, A. A. Schaffer, J. Zhang, Z. Zhang Z, W. Miller, D. J. Lipman, Gapped BLAST and PSI-BLAST: a new generation of protein database search programs, *Nucleic Acids Res.*, **25**, pp. 3389–3402, 1997.
4. A. Amir, M. Lewenstein, and E. Porat, Faster Algorithms for String Matching with k Mismatches, *In Symposium on Disc. Alg.*, pp. 794–803, 2000.
5. P. Brown and D. Botstein, Exploring the New World of the Genome with DNA Microarrays, *Nature Genetics*, **21**, pp. 33–37, 2000.
6. J. Feigenbaum, S. Kannan, M. Strauss, and M. Viswanathan, An Approximate l1-difference Algorithm for Massive Data Streams, *In Proc. FOCS99*, 1999.
7. U. Manber and G. Myers, Suffix Arrays: A New Method for On-line String Searches, *SIAM J. on Comp.*, **22**, pp. 935–948, 1993.
8. S. Muthukrishnan and S. C. Sahinalp, Approximate Nearest Neighbors and Sequence Comparison with Block Operations, *In Proc. 32nd annual ACM symposium on Theory of Computing*, pp. 416–424, 2000.
9. S. Muthukrishnan and S. C. Sahinalp, Simple and Practical Sequence Nearest Neighbors under Block Edit Operations, *In Proc. CPM2002*, 2002.
10. W. R. Pearson, Flexible sequence similarity searching with the FASTA3 program package, *Methods in Molecular Biology* **132**, pp. 185–219, 2000.
11. S. Yamada, O. Gotoh, H. Yamana, Improvement in Accuracy of Multiple Sequence Alignment Using Novel Group-to-group Sequence Alignment Algorithm with Piecewise Linear Gap Cost, *BMC Bioinformatics* **7**, pp. 524, 2006.
12. T. Yamada and S. Morishita, Computing Highly Specific and Mismatch Tolerant Oligomers Efficiently, *Bioinformatics Conference 2003*, 2003.
13. T. Yamada and S. Morishita, Accelerated Off-target Search Algorithm for siRNA, *Bioinformatics* **21**, pp. 1316–1324, 2005.