Knowl Inf Syst (2010) 25:229–251
DOI 10.1007/s10115-009-0271-6

REGULAR PAPER

Multi-sorting algorithm for finding pairs of similar short substrings from large-scale string data

Takeaki Uno

Received: 30 September 2008 / Revised: 20 October 2009 / Accepted: 23 October 2009 /
Published online: 25 November 2009

Abstract Finding similar substrings/substructures is a central task in analyzing huge string data such as genome sequences, Web documents, log data, feature vectors of pictures, photos, videos, etc. Although the existence of polynomial time algorithms for such problems is trivial since the number of substrings is bounded by the square of their lengths, straightforward algorithms do not work for huge databases because of their high degree order of the computation time. This paper addresses the problem of finding pairs of strings with small Hamming distances from huge databases composed of short strings of a fixed length. Comparison of long strings can be solved by inputting all their substrings of fixed length so that we can find candidates of similar non-short substrings. We focus on the practical efficiency of algorithms, and propose an algorithm that runs in time almost linear in the input/output size. We prove that the computation time of its variant is linear in the database size when the length of the short strings is constant, and computational experiments for genome sequences and Web texts show its practical efficiency. Slight modifications adapt to the edit distance and mismatch tolerance computation. An implementation is available at the author's homepage.

Keywords Neighbor search · Neighbor graph construction · Similarity analysis · Data analysis · Large scale data · Homology search

1 Introduction

These days, we have many huge string data such as genome sequences, Web documents, log data, feature vectors of pictures, photos, videos, etc. Since the size of these data is so huge that humans 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 make 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

T. Uno (✉)
National Institute of Informatics, 2-1-2, Hitotsubashi, Chiyoda-ku, Tokyo 101-8430, Japan
e-mail: uno@nii.jp; uno@nii.ac.jp
URL: http://research.nii.ac.jp/~uno/index.html

substrings can be found in polynomial time by comparing all substrings one by one. However, algorithms that spend no less than square time do not work for huge data; therefore, practical fast algorithms, say quasi linear time algorithms, are needed.

In this paper, we consider the problem of enumerating all pairs of similar strings in a set $\mathcal{S}$ of strings of the same length l. As a similarity measure, we use Hamming distance. Thus the definition of the problem is as follows.

Short Hamming Distance String Pair Enumeration Problem:
Input: set $\mathcal{S}$ of strings of the same length l, distance threshold d
Output: all pairs of strings S_1 and S_2 such that the Hamming distance between S_1 and S_2 is at most d.

We call a pair of strings with Hamming distance of at most d a *similar string pair*. We consider the case in which the length l is small, and propose a practical and efficient algorithm. The idea of the algorithm is to classify the strings in several ways so that two strings of a similar string pair 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. Suppose that we partition each string into k blocks. We choose $k-d$ indices of blocks and classify the strings so that two members of a group have the same letters on the specified blocks. Two strings in a similar string pair must share at least the same $k-d$ blocks; thus by examining all the combinations of $k-d$ indices, the two strings must be in the same group for at least one combination. Thus, performing pairwise comparison in each group for each combination, we can find all the similar string pairs. We call this "multi-sorting algorithm", since we use radix sort for the task of classification. If the $k-d$ blocks have sufficiently many letters, the members of each group is expected to be few; thus all pair comparisons takes quite a short time, and practical computation time will be close to linear time. As we show later, the results of the computational experiments for genome strings show that the algorithm is practically efficient. The algorithm can be parallelized well. Computational experiments show that the algorithm is scalable for the number of processors.

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 of the algorithm can be bounded by $O({}_lC_d \times (|\mathcal{S}| + dN)) = O(2^l(|\mathcal{S}| + dN))$, where N is the number of pairs to be output. This is linear in the input and output size, but exponential in d; thus for fixed l, we can say that the algorithm is linear time in the input and output size. This is the theoretical aspect of this algorithm. This algorithm can be applied to the case of edit distance, with a slight modification. However, the computation time will be much longer compared to the Hamming distance, thus we thought that the Hamming distance is useful practically.

One of the disadvantages of our problem model is that we may have many output pairs, and each pair does not give the shape of local similarity structures. We introduce a new similarity measure called *continuous interval Hamming distance* of two strings defined by the maximum Hamming distance of all their substrings of given length l. Using this measure we can naturally introduce a maximality to the similar strings. By outputting maximal similar string pairs, which can be obtained from similar string pairs, we can capture the local similarity and reduce the output at the same time. The computation time for checking the maximality is bounded by the number of similar string pairs and thus we lose no efficiency in terms of time complexity. The experimental results show that the number of output solutions is drastically small compared to similar string pairs.

Using the algorithm makes it possible to approach the problem of finding similar non-short strings, and similar non-short substrings of long strings. We can observe that two non-short similar strings may have several similar string pairs as their substrings. Thus,

pairs of non-short strings including several similar string pairs are candidates for non-short similar substrings. This approach has a certain certification of accuracy. For example, any two strings L_1 and L_2 of 3,000 letters with the Hamming distance of at most 293 have at least three pairs of substrings of 30 letters starting from the same position of L_1 and L_2 such that the Hamming distance between them is two at most.

This fact motivates us to find mid-length strings pairs L_1 and L_2 such that L_1 and L_2 have at least a certain number of similar substrings such that the difference of their starting positions is bounded by some constant. From these observations, we propose a way to find "candidates" of similar substrings, by finding a set of a certain number of similar string pairs composed of substrings whose positions are close to each other. In this way, we compared the human genome and mouse genome by our algorithm. The computation is done in quite a short time, and even though we found the candidates of similar structures, we could see homology structures figured out by the comparison.

The organization of this paper is as follows. The rest of this section shows related works and applications to the problem. Section 2 is for preliminaries, and Sect. 3 shows our algorithm. Section 4 describes the details of the application to long similar string detection, and Sect. 5 presents some extensions and generalizations of our algorithm and problem. We show the computational experiments in Sect. 6.

1.1 Related works

The short Hamming distance string pair enumeration problem can be considered as a kind of neighbor graph (or, ϵ-neighbor graph) construction problem. For given data, the neighbor graph construction problem is to construct the graph such that each vertex corresponds to an object in the data, and an edge connects two vertices when corresponding two objects are similar to each other, i.e., the distance between them are no more than the given threshold. Neighbor graph is often used for clustering, for example [5,12]. Constructing a neighbor graph is to find all the pairs of objects such that the distance is no more than the threshold. Hence, if the data are a collection of short strings of the same length, it is equivalent to the short Hamming distance string pair enumeration problem. Even when the objects in the data are not short string, nor the distance is not given by Hamming distance, we can use Local Sensitive Hashing. Local Sensitive Hashing is a way to maps objects to short strings such that similar objects are mapped to strings of short Hamming distance with high probability. In [12], they also use Local Sensitive hashing for the clustering, but they find only the pairs of the same strings. By finding string pairs of short Hamming distance, we may be able to more efficiently find the clusters.

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 the Hamming 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 also been 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 [8]. For edit distance, which allows insertions and deletions, algorithms proposed by Muthukrishnan and Sahinalp [15,16] approximate the

minimum distance substring. Using these algorithms, the problem can be solved in a shorter time but may fail with some solutions. These algorithms take more than $O(|S|^2)$ time to find similar substrings even for fixed length strings, Thus, direct applications of these algorithms does not work in practice.

On the other hand, there have been 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 a 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 [21] 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 $O(|S| \log |S|)$ time, then answers a lower bound in $O(|Q|L)$ time for any Q, where L is a constant no greater than $|Q|$. They also proposed an efficient exact algorithm for strings with a small alphabet such as genome sequences [22].

In bioinformatics, the problem of finding substrings of two strings that 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 a short time is difficult thus many heuristic algorithms have been proposed. BLAST and FASTA [2,3,17] 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 of length more than 10 is no more than 9% of their length, they always have a common substring of 10 letters. The disadvantage of this method is that when the strings are long, a large number of substrings are the same and thus a lot of comparisons must be made. Such frequently appearing strings can be considered to be not important 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 [20]. Some statistics of the blocks are computed, for example, the number of each letter in the blocks, which is for detecting the pairs of blocks which 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 few.

1.2 Applications

There are many kinds of applications of string similarity detections in both industrial and academic areas. We look at some of them.

(1) Genome sequence homology detection
A genome sequence is a string composed of letters ATGC witch represents the sequence of amino acid in the DNA of a species, or an individual animal/plant. To capture the evolutionary changes of organisms, their (common) ancestors, or the genomic difference of species, finding similar substrings of genome sequences is an important task. The length of a genome sequence is usually quite large. For mammalians including homo sapiens, the total length of genome sequences is up to 4 billion. Therefore, the speed of comparison algorithms is quite important. A popular tool for this task BLATZ

is based on the idea of finding the pairs of similar substrings and visualizing those pairs by a dot plot. However, since it finds exactly the same quite short strings as the seeds of similar substrings, quite many string pairs will be found; thus BLATZ may take more than 1 week to compare two chromosomes of length 100 million [18]. Murasaki [18] terminates in a short time, say 2 or 3 h, by decreasing the error ratio, i.e., it finds only pairs of long and quite similar strings. By using our approach, we can find pairs of similar not-so-short substrings in a short time, say 15 min, without decreasing the error ratio. Using the pairs found as the seeds, we can speed up without losing much accuracy.

(2) Assembling fragments to genome sequences [9]
It is not easy to read the genome sequence from DNA at all once. What we can do now is to read short sequences with at most thousand letters. To construct the whole genome sequence completely, we usually randomly cut many DNA fragments taken from an organism, read such fragments of thousand letters, and connect the fragments by using the overlapping fragments. Since the genome-sequencing machine generates errors, and the genome itself has errors, overlaps have to be detected with errors. Thus, similarity detection among many fragments is important for this task. Generally, the number of fragments needed to construct a genome sequence of length 1 million is at least 6 million; thus construction of the human genome needs to solve the problem of 6 billion. The similar short substrings of fragments can be good candidates (seeds) for finding the overlaps.

(3) Mapping fragments to reference strings (genome sequences) [10]
Mapping a fragment means finding the positions of a reference string to which the fragment and the substrings starting from the positions have a small distance. This mapping is used in several application areas. In genomic science, genome sequence fragments of a species or an individual organism are mapped to the genome sequence of a similar species, or the same species, to detect small differences (changes) of the genome sequences between species or individuals. Constructing the entire genome sequence by assembling the fragments is a difficult task, and requires a large number of fragments, but mapping is lighter task and performs well when we want to capture the small differences. The similar substrings of fragments and reference strings gives us the positions on which the fragments are mapped; thus we can efficiently solve the problem by using our algorithm. If the fragments are long, short substrings will give candidates of the position to be mapped.
This detection can be applied to OCR (optical character recognition). When OCR device reads a document, it makes many errors thus many letters will be replaced with other letters. By mapping each sentence of the document to a text database such as Web documents, we may find the same substring of the sentence but including no error. Such detection enables automatic error correction of OCR documents.

(4) Detecting reference/copy relation of documents [19]
By finding similar sentences/phrases from document databases, we can detect the reference/copy relation among many documents. The relation is between parts of documents; thus the entire documents do not have to be similar. This will be a base of evaluation of value/originality of documents, detecting violation of copyrights, dependency, etc. Such reference/copy relations can be found by detecting similar substrings, which can be found by finding similar short substrings as the seeds. Note that it is not easy to find "sentences having the same meaning but written in different ways".

(5) Finding pairs of similar images, and similar scenes of videos [7]
Similarity of two images can be evaluated by extracting features of the images. The features can be represented by a string obtained by discretizing the features; thus we can

find pairs of similar images among many images in a short time by using our algorithm. Since movies are sequences of images, by finding similar images among those images, we can find similar scenes among many movies.

(6) Finding similar structures from sequential data [13]
Sequential data can be transformed to a string by discretizing the time and the values. Thus, partially similar subsequences can be detected by using our algorithm. In particular, similar parts/phrases of musics/songs can be detected. Since they often have the same phrase with different speed, we have to consider expansion/contraction of music data. This can be done by duplicating sequential data at different speeds. In this way, phrases are expanded/contracted uniformly. Thus our algorithm is expected to be efficient for music data.

(7) Finding partial similarity among matrix (geometical) data
Finding similar submatrices (with small Hamming distance) from large matrices can be solved in the same way to our algorithm, by dealing submatrices as substrings. Using this, we can find partially similar areas of images from many images. In this way, even when an image is cut, has a shade in front of the background image, and/or translation, we can detect similar areas. To deal with the rotation/expansion/contraction, we also use the same technique to (6); we prepare duplicated images obtained by rotating and/or expanding the original image. The partial similarity among them gives us the similarity with expansion, rotation, and translation.

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 letters is also a string and is called an *empty string*. The length of an empty string is 0. The ith letter of 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 a given threshold value d, we say two strings S_1 and S_2 of the same length are *similar* if their Hamming distance is no greater than d, and call them *similar string pair*. For string S and integers b and k, $b \leq k$, we denote the substring of S starting from $(\lceil |S|(b-1)/k \rceil + 1)$th letter to $(\lceil |S|b/k \rceil)$th letter, i.e., $S[\lceil |S|(b-1)/k \rceil + 1, \lceil |S|b/k \rceil]$, by $B(S, k, b)$. $B(S, k, b)$ is called the *bth block*.

For 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 insertions, deletions, and changes 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 string that is the same, 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, and threshold value d
Output: All pairs of strings S_1 and S_2 in $\mathcal{S}$ 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, and assume that the size of Σ is smaller than the total size of $\mathcal{S}$, i.e., $l \times |\mathcal{S}|$.

3 Multi-sorting 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 string S and set $C = \{b_1, ..., b_{k-d}\}$, $b_j < b_{j+1}$ taken from $C(k, k-d)$, we define $Sig(S, C) = B(S, k, b_1) \cdot B(S, k, b_2) \cdot \ldots \cdot B(S, k, b_{k-d})$. We suppose that an integer k, $d < k \leq l$ is chosen, and have a look at the following property:

Lemma 3.1 *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 $B(S_1, k, b) \neq B(S_2, k, b)$ holds, it includes at least one mismatch, i.e., $S_1[i] \neq S_2[i]$ holds for some b, $\lceil |S|(b-1)/k \rceil + 1 \leq i \leq \lceil |S|b/k \rceil$. Since S_1 and S_2 have at most d mismatches, at most d integers b satisfy $B(S_1, k, b) \neq B(S_2, k, b)$, thereby at least $k - d$ integers b satisfy $B(S_1, k, b) = B(S_2, k, b)$. Setting C to the set of those integers h satisfying $B(S_1, k, b) = B(S_2, k, b)$ shows that $Sig(S_1, C) = Sig(S_2, C)$. □

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 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-sorting method*. In this example, there are nine strings and set $d = 1$ and $k = 3$. Each block is composed of two letters, and classified by two blocks are done three times. For each classification there are several groups represented by rectangles with more than one string, and some of them contain strings with a Hamming distance of at most one, written at the head of the arrows.

ALGORITHM MultiSorting_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 $\mathcal{S}$

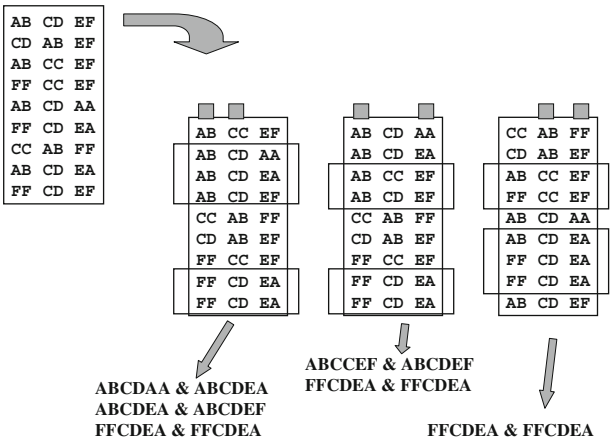

Fig. 1 Example of multi-sorting for finding strings with Hamming distance of at most one, by dividing strings in *three blocks* and classifying them by *two blocks*

are the same, and choose k such that the expected size of each group in a classification is less than 1. Then the comparisons in a group are 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

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 an integer one by one to C, which represents the combination of the blocks, to construct $C \in C(k, k-d)$ like a backtrack algorithm. Then, after choosing an integer b in some iteration of the backtracking, $B(S, k, b)$ is common to all C generated in the recursive call, i.e., until b is removed. Thus, the radix sort for $B(S, k, b)$ 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 $O(l/k \times |\mathcal{S}|)$. A radix sort usually requires $O(l/k \times (|\mathcal{S}| + |\Sigma|))$ time, since it puts each string into one of the $|\Sigma|$ buckets, and scans the buckets in increasing order of letters. However, we do not want to sort them, but just classify the strings; thus we need only non-empty buckets. Moreover, the buckets do not have to be sorted in alphabetical order. Thus, instead of scanning all the buckets, we make a list (or stack) of the ID's of non-empty buckets. Therefore, when we are given initialized $O(|\Sigma|)$ memory, the classification can be done in $O(l/k \times |\mathcal{S}|)$ time. Note that after the classification the memory re-initialized in $O(l/k \times |\mathcal{S}|)$ time, since at most $O(l/k \times |\mathcal{S}|)$ memory has been accessed. We describe the algorithm in the next subsection. Additionally, when the members of a bucket is not so many, say less than 10, we can perform the pairwise comparison directly, instead of recursively executing the radix sort. This can reduce the computation time when there are many buckets with small sizes.

3.2 Avoiding duplication without memory

The multi-sorting 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 for duplication when a new pair is found. Although this is simple, it requires much memory. We present a method that does not store the pairs found in memory, and thus requires no extra memory.

A pair of strings S_1 and S_2 is output more than once if $B(S_1, k, b) = B(S_2, k, b)$ holds for more than $k - d$ integers i, since $Sig(S_1, C) = Sig(S_2, C)$ 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 $B(S_1, k, b) = B(S_2, k, b)$, the computation is not a heavy task. The algorithm is described as follows, that requires an initial call with $\mathcal{S}$, d and k, and set $C = \emptyset$:

ALGORITHM MultiSorting ($\mathcal{S}$: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** b larger than the maximum integer in C **do**
3. radix sort to classify all strings $S \in \mathcal{S}$ according to $B(S, k, b)$
4. **for each** group K of the classification with $|K| > 1$
   call MultiSorting (K, d, k, $C \cup \{b\}$)
5. **end for**

Theorem 3.1 *The memory usage of algorithm MultiSorting is $O(|\mathcal{S}| + |\Sigma|)$, which is almost two integers for each letter in precise. The computation time of the algorithm except for step* 1 *is bounded by $O(|\Sigma| + l/k \times (|\mathcal{S}|) \times {}_lC_d)$.*

3.3 Choosing good k

The MultiSorting algorithm needs a parameter k, and it affects the computation time; if k is too small, the cost for pairwise comparison will be large, and if k is too large, the cost for radix sort will be large. In our computational experiments, the computation time was sufficiently short for many ks. Thus, we do not have to be serious about choosing k. We propose a simple way to choose a good k, and the computational experiments shows it is efficient enough in the practice.

For each letter $a \in \Sigma$, let $p(a)$ be the number of appearances of a in $\mathcal{S}$ divided by $l \times |\mathcal{S}|$. $p(a)$ is the probability that a letter randomly chosen from a string of $\mathcal{S}$ is a. The probability that two randomly chosen letters are the same is $p = \sum_{a \in \Sigma} p(a)$. Under the assumption that the letters of strings in S are independently chosen at random, two substrings of length t of some strings in $\mathcal{S}$ will be the same with probability p^t. If we divide each letter into k blocks, MultiSorting algorithm executes a radix sort with approximately ld/k letters; hence two strings will be in the same bucket with probability approximately $p^{ld/k}$. This means that the expected number of the members in each bucket is $|\mathcal{S}| \times p^{ld/k}$.

Let k^* be the minimum k such that $|\mathcal{S}| \times p^{ld/k} \leq 1$, i.e., the expected number of the members is no more than one. We choose k^* for the parameter k. The computation time for pairwise comparison would be $O(|\mathcal{S}|)$ under the above assumption, thus shorter than the computation time for radix sorts. When the assumption does not hold, the cost for pairwise comparison will be large and will be equal to or longer than that for radix sorts. However, in our computational experiments, this choice usually attains almost minimum computation time among all possible ks.

3.4 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 $O(l|\mathcal{S}|(|\mathcal{S}| + {}_lC_d))$. We will save the computation time in step 1.

Let $k = l$. Then, for each i, $B(S, k, b)$ 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 any pair of strings in the same group will be output. Thus, the computation time in step 1 is bounded by product of the size of output, and the maximum number of duplications for one pair, which is $|C(l, d)| = {}_lC_d$. 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 $Sig(S_1, C) = Sig(S_2, C)$, and each radix sort is performed in $O(|\mathcal{S}|)$ time.

Theorem 3.2 *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 $O(|\Sigma| + {}_lC_d \times (|\mathcal{S}| + dN)) = O(|\Sigma| + 2^l(|\mathcal{S}| + dN))$ time with $O(|\mathcal{S}| + |\Sigma|)$ memory where N is the number of output string pairs.*

4 Approach to long substrings

The algorithms proposed here are to detect similarity in short strings. In this section, we show an approach to detect non-short similar substrings based on short similar substring enumeration. A straightforward approach may take more than square time, since the length of strings to be compared cannot be bounded by a constant.

One typical approach to capturing similarity structures by using similar string pairs 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 pairs of similar substrings taken from one block and a substring taken from the other block. We can visualize the similarity structure in this measure by a figure such that the intensity of the color of the pixel (x, y) is given by the similarity. This method is called "dot plot" in bioinformatics. The left of Fig. 2 shows an example of images obtained by this method. The images are drawn by solving the problem with parameters $l = 30$ and $d = 2$. The computation is done in few minutes.

If the blocks are large, any two blocks contain a sufficiently large number of similar string pairs; thus all pixels will be of the same color. Moreover, we need much time for computation. In such cases, we have to reduce the number of outputs, without losing important information. One simple way to reduce the output is to output the pairs included in longer similar substrings. For example, we choose a constant k, and output a similar substring pair S_1 and S_2 only if S_1 and S_2 are substrings of L_1 and L_2 of length kl such that $S_1 = L_1[i, i + l - 1]$ and $S_2 = L_2[i, i + l - 1]$ hold for some i, and $HamDist(L_1, L_2) \leq kd$. We can also use the edit distance to be sensitive for insertion/deletion error.

Another way for reducing the output is sampling the substrings. For example, if we choose 1 of 10 substrings as substrings to be compared, we can reduce the number of output pairs by possibly 1/100. However, this approach may miss some middle-length similar substrings.

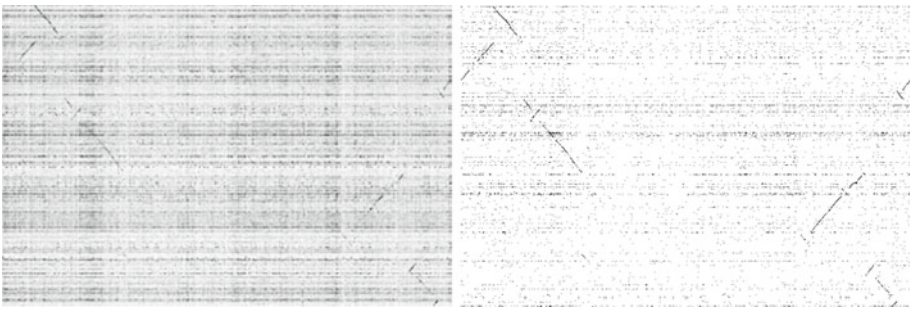

Fig. 2 Matrix showing similarity of mouse 11 chromosomes (X-axis) and Human 17 chromosome (Y-axis), with *black cells* on similar parts; we can see similar substructures as *diagonal lines*; the *left figure* represents the density of similar string pairs, and the *right figure* is the result of our filtering

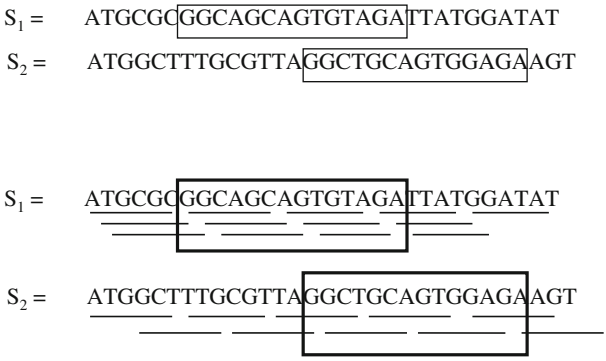

Fig. 3 (*Up*) Maximal (6,1)-CIH substrings of S_1 and S_2; (*down*) positions (substrings) taken by the interleaving approach with $p = 2$. the maximal substrings include two similar string pairs, and the *left one*, GCAGCA and GCTGCA, are the canonical pair

We propose a way to sampling the substrings that never miss similar substrings with a certain length.

Suppose that we are going to compare long strings T_1 and T_2, by finding similar substrings for length l and Hamming distance at most d. We first choose a divisor p of l. Then, we take substrings from T_1 such that their starting positions are $1, p+1, 2p+1, \ldots$, and take substrings from T_2 such that their starting positions are $1, 2, \ldots, p,\ l+1, l+2, \ldots, l+p,\ 2l+1, 2l+2, \ldots, 2l+p, \ldots$. An example of such a method of taking substrings is shown at the bottom of Fig. 3. We call this method the *interleave method*. Suppose that L_1 and L_2 are substrings of T_1 and T_2 of length m such that $L_1 = T_1[i, i+m-1]$, $L_2 = T_2[j, j+m-1]$. Let $k = (i-j) \bmod l$, $x = p\lceil k/p \rceil$, and $y = k - (k \bmod p)$ Note that when $i - j < 0$, $k = l - ((j-i) \bmod l)$. Then, we can see that two substrings S_1 of L_1 and S_2 of L_2 of length l starting from $b+1$th letter, i.e., $S_1 = T_1[i, i+b+l-1]$ and $S_2 = T_1[j+b, j+b+l-1]$, are both taken to be compared if and only if $(i+b) \bmod l = x$, since $i + b \bmod l = x$ means that $j + b \bmod l = y$. Thus, for every l consecutive substrings pairs of length l, at least one pair satisfies that both substrings are taken for the comparison. Thus, intuitively, we never miss L_1 and L_2, if they are not sufficiently short and their Hamming distance is not large. More precisely, we never miss L_1 and L_2 if their length is no less than $2l$, and the Hamming distance is less than $(d+1) \times \lfloor |L_1|/l \rfloor$. In the case that l has no divisor, or few divisors, we can choose a number $l' < l$ and its divisor p, and take strings from positions $1, 2, \ldots, p,\ l'+1, l'+2, \ldots, l'+p,\ 2l'+1, 2l'+2, \ldots, 2l'+p, \ldots$. In this way, we lose the above certification, but expect that the practical efficiency does not change much.

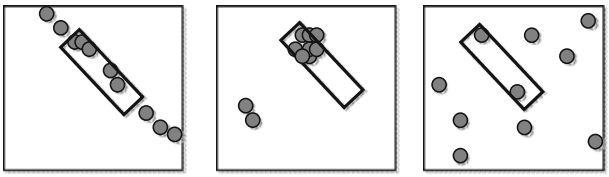

Fig. 4 Three examples of cells with the same number of similar string pairs. *Each dot* represents the position of a string pair. The *diagonal rectangle* is the bounding condition to be a seed. When the threshold number is three, the *left and center cells* have seeds. However, the seeds in the *central cell* are too concentrated in a small area and thus we have to remove the seeds

We briefly explained the second approach in the introduction. Observe that any two strings L_1 and L_2 sufficiently longer than l with a Hamming distance less than $\lfloor |L_1|/l - h \rfloor \times d + (d-1)$ must include at least h similar string pairs. For example, when $l = 30$, $d = 2$, and L_1 and L_2 are of 3,000 letters with a Hamming distance of at most 293, they include at least three similar string pairs. Generally speaking, if the Hamming distance of two strings of length Kl is at most $(K-h)d$, then the string pair has at least h pairs of substrings starting from the same positions and having a Hamming distance at most d. This is because the string has K disjoint substring pairs, and at most $(K-h)$ of them can have no less than d distinct letters. We have similar observations for the edit distance. This comes from the same reason as Lemma 3.1. We call such pairs a *seed*. It implies that there are long similar strings with over 3,000 letters only if there is a seed. This motivates us to find seeds to capture the long string similarity; draw an image by putting a dot if there are such three pairs. Furthermore, if the member of a seed lies in a short interval, say 300, and there exits no other similar string pair near by them, then the seed indicates short similar strings; thus we also delete such isolated seeds within a small area. To find such pairs, we classify all similar string pairs according to the difference in the starting positions of two strings in the pair. We then sort the similar pairs with the same difference in the starting positions according to the starting position of the first string. Then, by scanning the obtained sorted list of pairs, we can easily find seeds. The classification and sorting of each group can be done using a radix sort in linear time. This task can be done in $O(N)$ time, where N is the number of similar pairs.

This approach can be applied even when we consider the edit distance. An insertion/deletion can make the Hamming distance of two strings quite large. Thus, if the number of insertions/deletions between two strings is relatively small, we can state a certain certification of accuracy. Consider an example of two strings of 3,000 letters with an edit distance of at most 198 with insertions/deletions of at most 55. Then, they have at least three substrings of 30 letters with a Hamming distance of at most two. For the edit distance, the differences in the start positions of the pairs in a seed do not have to be the same, but the differences are bounded, at most 55. In Fig. 4, we present some examples of similar string pairs in cells. Some bounding conditions of seeds are inside the diagonal boxes.

To find all seeds which are composed of three pairs in the above case, we modify the above method. The starting positions of pairs in a seed can differ and thus we have to merge several groups with similar starting position differences, then sort the pairs and scan them. We call the merged groups a *belt*. An intuitive image of a belt is shown in Fig. 5. Thus, the computation time is multiplied by the width of the belt, but we can reduce the computation time by using a binary tree. We construct a binary tree representing the sorted order of the pairs in a belt, then we shift the belt by removing one group and adding group to the belt, and by using binary tree, we update the sorted list of the pairs in the belt. In this way, the computation time to find all pairs not included in any seed is $O(N \log N)$.

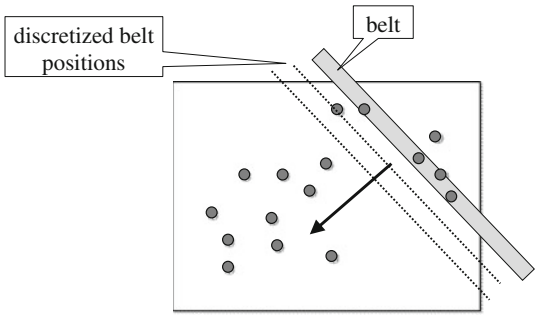

Fig. 5 Example of belt; it sweeps in *left-down direction*. It updates the sorted order of string pairs it contains, and finds the pairs satisfying the condition to be a seed. Discretized belt that has the doubled width, is placed only at the *dotted lines*

In practice, we can use a simpler method by discretizing the belts. We double the width w of the belt, and scan only one belt among w belts. An example of the positions of belt is shown in Fig. 5. The computation time of this method is linear, but it never misses any seeds. However, some similar string pairs not included in any seed may be judged to be in a seed. However, we believe such an error is not critical, since we can consider that random noise produces few errors. If there are many such errors, we should consider them as a kind of similarity. The right image of Fig. 2 was made in this approach. The image was drawn by first finding the problem with parameters $l = 30$ and $d = 3$, and finding seeds composed of three pairs with length 3,000 and width 300, by the discrete belt approach. We discard the isolated seeds, if a seed is in an area of length 300, and has no similar string pair with a distance shorter than 2,700, in the belt. The resolution of the image is 2,000 by 2,000, and each dot is written when it has at least one seed. Each dot is enlarged for emphasizing. We successfully removed the noise patterns from the image and emphasized the similar structures for better understanding. The computation was done in few minutes.

5 Extensions

In 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-sorting 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 string. Such unique strings can be used as characterizations, invariants of string databases, or markers of substructures. A typical application is in microarrays. A microarray is a tool for biological experiments that can detect the existence of short strings, say 25 letters, in the genome sequence of species or an organism. 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 similarity measure is the Hamming distance, *mismatch tolerance* [21] is such a uniqueness measure. 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 $\min\{HamDist(S, S') \mid S' \in \mathcal{S}\backslash\{S\}\}$. If $mis(S, \mathcal{S})$ is large, S has no similar string in $\mathcal{S}$ in the sense of Hamming distance. Thus, our

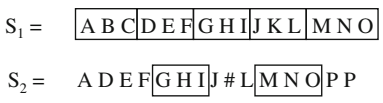

Fig. 6 Two strings S_1 and S_2 with edit distance of three. Among five blocks, two blocks of S_1 have same corresponding blocks. The second block of S_1 has actually the same block in S_2, but it is outside, and thus does not satisfy the condition of Lemma 5.1

aim is to find the strings with no large and not too small mismatch tolerance. We define the problem as follows:

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

The output strings are similar to at least one other string in $\mathcal{S}$; thus the remaining strings are unique, i.e., similar to no other string in $\mathcal{S}$. This problem can be solved by solving the short Hamming distance string pair enumeration problem. We do not have to output pairs and therefore we do not check duplications. Moreover, in the complete version of our algorithm, we have to execute the algorithm only for $d' = d$, and omit the computation of the Hamming distance. Thus we obtain the following theorem.

Theorem 5.1 *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 Generalize to edit distance

In many studies and real-world applications, the distance between two strings, genomes, and documents is evaluated by edit distance. The multi-sorting 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 the 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., $\hat{C}(k, d) = \{C \mid |C| = d, C \subseteq \{1, 1^+, 1^-, 2, 2^+, 2^-, \ldots, k, k^+, k^-\}\}$. b^+, b^- and b means an insertion, a deletion and a change at the bth block, respectively. For $C \in \hat{C}(k, d)$, let $\mathrm{sft}(C, b) = |\{j^+ \mid j < b, j^+ \in C\}| - |\{j^- \mid j < b, j^- \in C\}|$, and $Eq(C) = \{b \mid b, b^+, b^- \notin C\}$. We denote $S[\lceil |S|(b-1)/k \rceil + 1 + j, \lceil |S|b/k \rceil + j]$ by $\hat{B}(S, b, j)$. We display two of these blocks of string S_2 in Fig. 6. Then, for string S and $C \in \hat{C}(k, d)$, we define $\overline{Sig}(S, C)$ by $\hat{B}(S, b_1, \mathrm{sft}(C, b_1)) \cdot \hat{B}(S, b_2, \mathrm{sft}(C, b_2)) \cdot \ldots \cdot \hat{B}(S, b_{k-d}, \mathrm{sft}(C, b_{k-d}))$ where $Eq(C) = \{b_1, \ldots, b_{k-d}\}, b_h < b_{h+1}$. By using this terminology, we obtain the following corollary obtained from Lemma 3.1. An example is shown in Fig. 6.

Lemma 5.1 *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 $Sig(S_1, Eq(C)) = \overline{Sig}(S_2, C)$.*

Proof Suppose that the edit distance e between strings S_1 and S_2 of length l is no more than d, and A is an alignment which yields the edit distance. Note that an alignment is a matching

of letters of S_1 and S_2, allowing matching a letter and a gap, and here A has exactly e matches between a letter and a gap, or two distinct letters. Let $X = \{b_1, \ldots, b_e\}$ be the set of the positions of the letters of S_1 in these e matches. When a letter of S_2 matches with a gap between ith letter and $(i+1)$th letter of S_1, we define the number for this match by $i + 0.5$. We say that bth block of S_1 is *outside* if the number of positions included in the blocks from the first $(b-1)$th is strictly larger than $b-1$. Since $|\mathrm{sft}(C, b)| < b$, bth block of S_1 can satisfy $\hat{B}(S_2, b, \mathrm{sft}(C, b)) = B(S_1, k, b)$ holds for some $C \in \hat{C}(k, d)$, which means that the bth block of S_1 and the shifted bth block of S_2 according to C are the same, if the bth block includes no position in X and is not outside. Thus, the statement of this lemma holds if at least $k - d$ blocks include no position of X and are not outside.

If there is no outside block, then in the similar way to the proof of Lemma 3.1, we can see that the statement of the lemma holds. If there are outside blocks, we suppose that the bth block has the largest index among all outside blocks. Then, among $k-b$ blocks from $b+1$ to k, at most $d-b$ blocks have positions in X. This implies that at least $(k-b)-(d-b) = k-d$ blocks are not outside and include no position of X. This concludes the proof. □

From this lemma, we are motivated to classify all strings by $Sig(S, Eq(C))$ and $\overline{Sig}(S, C)$ 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 classified group, we can find all pairs of strings with an edit distance of at most d.

Theorem 5.2 *The computation time of Multi-sorting algorithm modified to edit distance is bounded by $O(|\Sigma| + 3^d l/k \times |\mathcal{S}| \times_l C_d)$, except for that for step 1.*

The duplication can also be checked by introducing a representative among all pairs of $C \in \hat{C}(k, d)$ and $Eq(C)$ satisfying $Sig(S_1, Eq(C)) = \overline{Sig}(S_2, C)$. Such a representative can be computed in $O(l^2)$ time.

First, by using a usual shortest-path algorithm, find an alignment of S_1 and S_2 whose cost is equal to the edit distance. From Lemma 5.1, we can see that there are at least $k - d$ non-overlapping same blocks in the alignment. Note that two same blocks are included in an alignment if they match ith letters of both blocks for any i. We iteratively choose such same blocks with the smallest start position so that they are not overlapping, $k - d$ times. We define the representative blocks of S_1 and S_2 by the blocked obtained in this way. Note that since edit distance computation algorithms are deterministic, they always compute a uniquely determined solution for the same input. In this way, we can compute the representative in $O(l^2)$ time.

Theorem 5.3 *For set $\mathcal{S}$ of strings of length l and distance threshold d, we can find all pairs of strings with an edit distance of at most d in $O(|\Sigma| + 3^d \times_l C_d \times (|\mathcal{S}| + l^2 N)) = O(|\Sigma| + 2^l 3^d (|\mathcal{S}| + l^2 N))$ time, where N is the number of string pairs to be output.*

5.3 Dealing with large l

In some application areas, the length l of the string will be large. For example, let us consider a database of Euclidean vectors composed of discrete values, which can be considered as strings, by regarding each value as a letter. Let us consider the case that the dimension of the vectors is large, and threshold d is also large. For example, if the dimension is 1,000 and $d = 100$, and the number of blocks is 103, the multi-sorting algorithm performs radix sorts ${}_{103}C_3 \geq 150{,}000$ times.

In such cases, we partition the dimension into *chunks*, and find pairs of vectors such that for some b, the Hamming distance of their bth chunks is small, more precisely, the same

error ratio. Here the error ratio of two strings is their Hamming distance over their length. For the above example, the error ratio of the Hamming distance is $100/1{,}000 = 0.1$. When we partition the dimension into 10 chunks of equal size, we find pairs of vectors such that there holds for some b, the Hamming distance of their bth chunks is at most 10. For each b, we execute the multi-sorting algorithm to the bth chunks of all vectors. If the hamming distance of two vectors is no greater than 100, the Hamming distance of their bth chunks is at most 10 for at least one b. Thus, any such pair of vectors is found at least once in the executions of the multi-sorting algorithm. By choosing good-sized chunks, we can perform the comparison quickly, even when the dimension is large. We set the number of chunks so that the expected computation time will be the minimum, by estimating the computation time with the method in Sect. 3.3 for all possible choices. In out computational experiments, this choice usually attains the computation time closed to the minimum.

5.4 Maximal similar substrings

When we want to capture similar substrings of two long strings T_1 and T_2, we may face some difficulties from the model of similarity. One of the difficulties is the large amount of output. When we find all similar short substrings, we may obtain many pairs, and a large number of string pairs makes it difficult to look at all the similar pairs in details. The second difficulty is the definition of the representative similar substrings, or in other words, the definition of the boundary of similar substrings.

Generally, maximality is a useful way to introduce a representative. For example, suppose that we give a threshold θ for the error ratio, and want to find a pair of substrings with an error ratio no greater than θ. Then, we have to define the maximal similar substring pair by the substrings that are included in no other such pair. However, since the inclusion relation does not satisfy the monotone property, this definition of maximality is not useful in practice; when the error ratio of T_1 and T_2 is no greater than θ, we may miss many internal similar substrings.

Even if we give a length m for similar strings to be found, an ambiguity occurs at the definition of the end of the similar strings. For strings AAABBBBAAA and CCCBBBBCCC, $d = 2$, and length $m = 6$, we have three pairs of similar strings (AABBBB,CCBBBB), (ABBBBA,CBBBBC), and (BBBBAA,BBBBCC), which should be considered as one similar structure. Moreover, for strings ACCADDAEEA and BCCBDDBEEB, we have five pairs of similar strings (ACCADD,BCCBDD), (CCADDA,CCBDDB), (CADDAE,CBDDBE), (ADDAEE,BDDBEE), and (DDAEEA,DDBEEB). In such a case, we would say that the strings are uniformly similar.

To avoid these difficulties, we introduce a new similarity measure called *continuous interval Hamming distance* (CIH). The CIH of strings L_1 and L_2 for length parameter l, written as $CIH(L_1, L_2, l)$, is defined by the maximum Hamming distance between $L_1[i, i+l-1]$ and $L_2[i, i+l-1]$ among all i, $1 \leq i \leq |L_1| - l + 1$. Intuitively, it is the maximum Hamming distance of substrings in L_1 and L_2 with the same length l and starting at the same position.

In CIH, we can clearly define the maximal similar substrings, and the end of similar substrings, since a monotone property holds. If $CIH(L_1, L_2, l) = d, CIH(L_1[i, i+k], L_2[i, i+k], l)$ is always no greater than d for any $k \geq l$ and $1 \leq i \leq |L_1| - k + 1$. Thus, we can naturally define a maximal pair of substrings with CIH, l, and d, by a pair of substrings L_1 of T_1 and L_2 of T_2 whose CIH is no greater than d but any extension of them has a CIH of more than d. More precisely, a pair of substrings $L_1 = T_1[i, i+k]$ and $L_2 = T_2[j, j+k]$ is *maximal* (l, d)*-CIH* if (a) $CIH(L_1, L_2, l) \leq d$, (b) $CIH(T_1[i-1, i+k], T_2[j-1, j+k], l) > d$

if $i, j > 1$, and (c) $CIH(T_1[i, i+k+1], T_2[j, j+k+1], l) > d$ if $i+k \leq |T_1|$ and $j+k \leq |T_2|$. Therefore, the end of the similar substrings is clearly defined.

Finding all maximal (l, d)-CIH substrings is not difficult by using our multi-sorting algorithm. We find all pairs of substrings of length l and a Hamming distance of at most d, and find the maximal (l, d)-CIH substring pair by expanding the pair. By doing this, we can find all of them, but some duplications occur. To avoid duplications, we introduce the *canonical similar substring pair* for maximal (l, d)-CIH substrings, by the leftmost similar substrings. More precisely, for maximal (l, d)-CIH substrings L_1 and L_2, the canonical similar substring pair S_1 and S_2 is the pair $S_1 = L_1[1, l]$ and $S_2 = L_2[1, l]$. For substrings $S_1 = T_1[i, i+l-1]$ and $S_2 = T_2[j, j+l-1]$, the pair S_1 and S_2 is the canonical similar substring pair of a maximal (l, d)-CIH substring pair if and only if (a) $i = 1$, (b) $j = 1$, or (c) $HamDist(T_1[i-1, i+l-2], T_2[j-1, j+l-2]) > d$. Thus, it can be checked in $O(1)$ time if we know the Hamming distance of S_1 and S_2. By outputting the maximal (l, d)-CIH substring pair only when its canonical similar substring pair is found, we can avoid duplications without losing the completeness of the output.

The idea of a canonical substring can be applicable when not all the pairs are compared, such as the interleave method in the previous section. Let $\mathcal{P}$ be a set of pairs of substrings of length l taken from strings T_1 and T_2. We suppose that $\mathcal{P}$ is the set of all similar substring pairs found by the interleave method. For maximal (l, d)-CIH substrings L_1 and L_2, the canonical similar substring pair is the pair $S_1 = L_1[i, i+l-1]$ and $S_2 = L_2[i, i+l-1]$ included in $\mathcal{P}$ such that for any $1 \leq j < i$, the pair $L_1[j, j+l-1]$ and $L_2[j, j+l-1]$ is not in $\mathcal{P}$. The canonicity of substring pair P can be checked by searching a string pair in $\mathcal{P}$, preceding P on the maximal (l, d)-CIH substring pair M, including P. This can be done by shifting the substring pair and updating the Hamming distance one by one, until we meet the end of M, or the preceding pair. This computation may take a long time for one check. However, by observing that each substring pair in M is accessed at most twice even if we test the canonicity for all string pair included in M and $\mathcal{P}$, we can see that the computation time for checking the canonicity for the pair included in M and $\mathcal{P}$ is bounded by $O(|M|)$.

When $\mathcal{P}$ is not the set of all pairs of substrings of length l taken from T_1 and T_2, we may miss some maximal (l, d)-CIH substring pairs. However, in the case that the strings to be compared are chosen in the manner described in the previous section, as the interleave method, we never miss the maximal (l, d)-CIH substring pairs with length at least $2l$, since such a pair always include a string pair in $\mathcal{P}$.

In practice, a maximal (l, d)-CIH substring pair can hardly be uniform. It is highly expected to include several string pairs of length l, with a Hamming distance smaller than d. In such cases, the idea of canonicity is still valid. We define $\mathcal{P}$ by the set of all string pairs with a Hamming distance of no greater than d', and define the canonicity. The total computation time for the canonicity check with respect to the maximal (l, d)-CIH substring pair M is still $O(|M|)$.

6 Computational experiments

This section shows the results of the computational experiments of our algorithm. The code was written in C, and compiled with gcc. The algorithm implemented is the first version of our algorithm, which is described in Sect. 3 with the techniques of avoiding duplications and reducing the cost of the radix sort. These experiments were done on a PC with Intel Core 2 Duo E8400 (3.0 GHz) with 4GB memory, with Linux and gcc. The implementation is available at the author's homepage; http://research.nii.ac.jp/~uno/index.html.

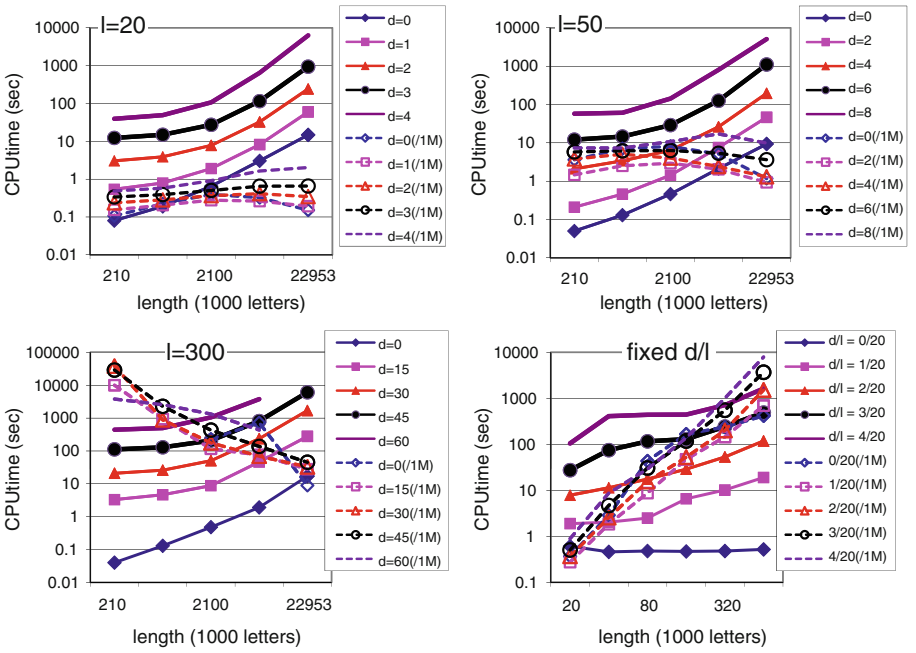

Fig. 7 Increase in computation time against increase in database size with fixed l and d: the *right-lower figure* is for fixed d/l inputting a string of 2.1 million letters

The instance was the set of substrings of fixed length taken from the Y chromosome of Homo sapiens. The data were taken from the National Center for Biotechnology Information (NCBI) genome repository, which is a database of genome sequences and genes. The data was downloaded from the Web site "ftp://ftp.ncbi.nih.gov/genomes/H_sapiens/". The data are of a text file composed of four letters A, T, G, and C, and some newline codes, to separate lines. We ignored the newline codes and the length of string is thereby 22, 952, 816.

The length of the short strings was set to 20, 50, and 300. Figure 7 shows the results.

Each solid line represents the computation time of a threshold value d, and each dotted line represents the computation time per 1,000,000 output similar string pairs. The X-axis is the number of input substrings, and Y-axis is the computation time. Both axes use log scales. We can see that the computation time increases slightly higher than linear time, but smaller than the square time.

We can also see that the computation time per 1,000,000 output pairs does not increase as the input size increases; thus we can say that the algorithm is output sensitive, i.e., the algorithm takes linear time in the output size.

We also show the increase in computation time against the increase in l with fixed d/l. The instance is fixed to that with 2.1 million strings, and the results are shown in the lower-right graph of Fig. 7. The left graph of Fig. 9 shows the number of executed radix sorts, which is the number of recursive calls. The horizontal axis is for the length of the input string, and the vertical axis is for the number of recursive calls. Each line corresponds to the results of fixed length l and threshold d. In this implementation, when the members in a group are sufficiently few, we execute a pairwise comparison immediately and do not execute further recursive calls. The results show the number of recursive calls is also robust for the increase

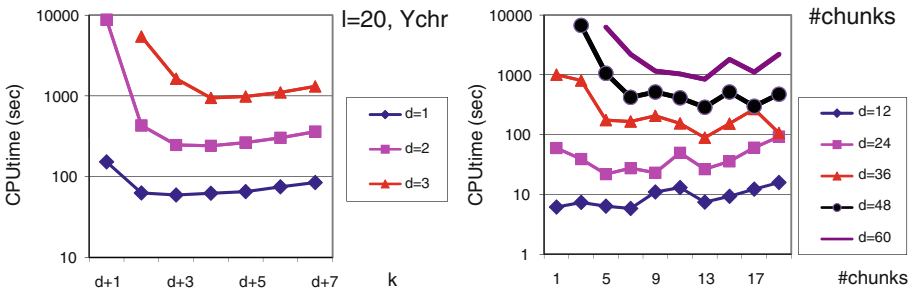

Fig. 8 *Left*: The experiments with the increase in k. *Right*: Experiments with the increase in the number of chunks

in the length when the length and the threshold are fixed. From these results, at least for genome sequences, our algorithm is quite scalable for the increase in the input string size.

The next experiment was to see the computation time against the increase in k. The instance was fixed to the Y chromosome, which has 22.9 million letters, and the length l was set to 20. We examined various k for $d = 0, 1, 2$, and 3, and the results are shown in the left graph of Fig. 8. The execution was terminated when it took more than 10,000 min, and the time is not plotted in the graph. Note that the case of $k = d + 1$ corresponds to the existing methods such as BLAST, since when $k = d + 1$, the algorithm finds string pairs having at least one same block. The number of letters included in a block is approximately ld/k; thus there is no big difference between the cases when $k = d + 1$ and the case when k is larger than $d + 1$, if d is small enough. However, when d is not small, that is $d > 1$, the computation time with $k = d + 1$ is quite long compared to the minimum computation time. Moreover, the computation time is close to the minimum for many ks; thus we can say that the choice of k is not a difficult problem.

In the right graph of Fig. 8, we present the results for the change of the number of chunks. The instance was fixed to that with 2.1 million strings, the length $l = 300$. We set the number of chunks to $1, 3, 5, \ldots, 19$, and examined them for the cases of $d = 0, 1, 2$, and 3. The execution was terminated when it took over 10,000 min, and the time is not plotted in the figure. We can observe that there is a big difference when d is large.

Table 1 shows the computation time with k^* which is chosen by the method described in Sect. 3.3, and the minimum computation time among all k. The references are the genome sequence of the Y chromosome of human, and Japanese Web texts. We can see that our choice k^* always attains almost minimum computation time. Table 2 shows the computation time by estimating the computation time described in Sect. 5.3, and the minimum computation time. The references are the first 2,100,000 letters taken from the genome sequence of the Y chromosome of human, and Japanese Web texts of 8,212,800 letters. Our choice also attains almost the minimum in many cases.

The Table 3 lists the results of determining the efficiency of the interleave method and maximal CIH. We used the Y chromosome of 22 million letters, and compared the number of similar string pairs and the number of maximal CIH substrings pairs. We fixed the length of the strings to 20, and examined for $d = 1, 2, 3$, and 4 and thus we found maximal $(20, d)$-CIH substrings for $d = 1, 2, 3$, and 4. By using the interleave method, the computation time and the number of solutions is reduced significantly. The additional computation time by introducing maximal CIH is not considerably long, while the number of solutions is drastically reduced. By using interleave, we missed many maximal CIH substrings, but their

Table 1 The choice of k and minimum computation time

	k^*	Time	min. k	min. time
Y chromosome $l = 20, d = 1$	3	55.17	3	55.17
Y chromosome $l = 20, d = 2$	6	335	4	281
Y chromosome $l = 20, d = 3$	7	944	7	944
Y chromosome $l = 50, d = 2$	3	45.5	3	45.5
Y chromosome $l = 50, d = 4$	6	218	6	218
Y chromosome $l = 50, d = 6$	9	1,036	8	843
Japanese Web text $l = 20, d = 1$	2	6.3	3	6.2
Japanese Web text $l = 20, d = 2$	3	11.0	3	11.0
Japanese Web text $l = 20, d = 3$	5	22.3	4	18.4
Japanese Web text $l = 50, d = 3$	4	15.9	4	15.9
Japanese Web text $l = 50, d = 6$	7	30.0	7	30.0
Japanese Web text $l = 50, d = 9$	10	82.1	11	60.7

Table 2 The choice of the number of chunks and minimum computation time

	Estimation	Time	min. k	min. time
Y chromosome (2.1 million letters) $l = 100, d = 5$	1	3.9	2	3.4
Y chromosome (2.1 million letters) $l = 100, d = 10$	1	61.3	4	20.3
Y chromosome (2.1 million letters) $l = 100, d = 15$	4	71.8	4	71.8
Y chromosome (2.1 million letters) $l = 300, d = 20$	1	31.5	7	11.4
Y chromosome (2.1 million letters) $l = 300, d = 30$	8	49.8	8	49.8
Y chromosome (2.1 million letters) $l = 300, d = 40$	11	144.8	14	118.4
Japanese Web text $l = 100, d = 5$	1	21.1	1	21.1
Japanese Web text $l = 100, d = 10$	1	39.3	1	39.3
Japanese Web text $l = 100, d = 15$	2	74.3	4	66.0
Japanese Web text $l = 300, d = 20$	1	79.1	3	75.2
Japanese Web text $l = 300, d = 30$	1	130	4	113
Japanese Web text $l = 300, d = 40$	1	233	7	155

lengths are no more than 39, since if the length of maximal CIH substrings is no less than 40, it includes at least one similar string pair even if we use the interleave method.

The right graph of Fig. 9 is the experimental results with Japanese texts taken from Web pages, crawled at 2007. The data was collected by Kawahara and Kurohashi [11]. The data were obtained by extracting the body of text, i.e., not including HTML commands or images. The size of the alphabet was 4,638, and the length was 42,915,000. The horizontal axis is the length of input data in log scale, and the vertical axis is the CPU time and CPU time per one million output pairs, in seconds in log scale. The lengths of input were from 1.3 to 40 million. We fixed the length l to 30, and evaluated the increase of the computation time for each $d = 0, \ldots, 4$. The computation time per pair found did not increase against the increase in the input length; thus we would say our multi-sorting algorithm also scales for this data.

Table 3 The number of solutions in the case of CIH

	$d = 1$	$d = 2$	$d = 3$	$d = 4$
Similar pairs; time	59.64	242.04	940.29	6,290.98
#solutions	308430876	701452360	1430762244	3110427918
Maximal CIH; time	70.15	269.08	1,018.49	
#solutions	10083533	45952952	129868305	333156637
Similar pairs with interleave; time	14.22	56.59	185.34	1,000.12
#solutions	30014243	68299127	139346032	303003424
Maximal CIH with interleave; time	19.36	70.99	220.96	1,080.48
#solutions	5650668	20338491	47922714	101905970

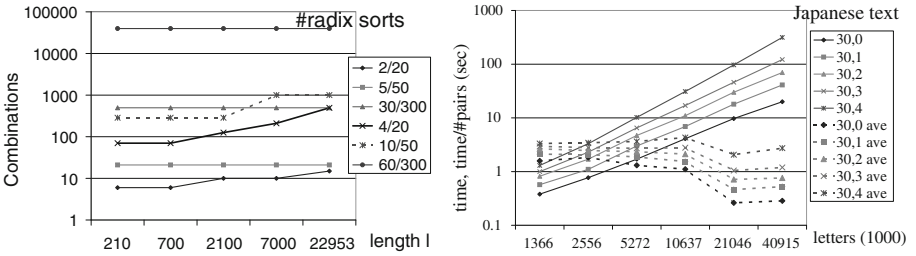

Fig. 9 *Left*: Number of radix sorts performed. *Right*: Experiments with Japanese Web texts

6.1 Parallelizing the algorithm

Of late, the price of multi-core CPU computers is decreasing, so we can easily use parallel computing systems. Our multi-sorting algorithm actually fits the multi-core CPU systems. Basically, the pairwise comparison in the classified groups can be operated in parallel. The classification is done by iterative radix sort. The radix sort itself is difficult to do in parallel, but the radix sorts in deeper levels can be performed in parallel. One iteration of the radix sort classifies strings into buckets, and the radix sorts in the next level will be performed for each group. Thus, in the levels other than the first level, the radix sorts in the groups can be performed in parallel. We implemented this parallelization by using thread library, which enables us to use multi-threads in a multi-core CPU. Note that it cannot be used for parallelization with cluster computers. After the first iteration of the radix sort, we make c threads, where c is the number of cores, and a queue of non-empty buckets which can be accessed from any thread. Each thread takes a bucket from the queue, with locking the queue counter, and operates further radix sort iterations concerned to the bucket. When a thread completes a bucket, it takes the next bucket from the queue. In summary, other than the inputting routine and the radix sort in the first levels, we can perform the computation in parallel.

We implemented a parallel version of our algorithm and evaluated the performance on a workstation with a quad core AMD Opteron processor of 4 cores, and an Intel Core 2 Duo E8400 of 2 cores. The instance was Y chromosome of homo sapience, and the length of the strings taken was 30. We changed the number of cores used, for several threshold values. The results are listed in the following table. Each cell shows the computation time (s).

d	AMD 1 core	AMD 2 core	AMD 4 core	Intel 1 core	Intel 2 core
0	20	17	15	10	7.9
1	47	34	27	27	17
2	138	89	64	88	51
3	353	237	142	285	151
4	1,507	835	490	1,111	576

When d is small, almost all computation time is spent for the initialization, the input of the problem, and the first level radix sorts. Hence, acceleration by parallelization is relatively small. However, for larger d, the computation time for these initialization processes is no longer the majority and thereby the computation time is reduced much. In particular, when $d = 4$ with Intel CPU, the acceleration is quite high, and close to twice the original by two cores. Although the factor differs depending on the type of CPU, the computation is accelerated by parallelization.

7 Conclusion

We proposed an efficient algorithm for enumerating all pairs of strings with a Hamming distance of at most given d from string set $\mathcal{S}$. We proposed multi-sorting algorithm whose computation time is practically linear time. We proved that the computation time of its variant is bounded by linear of the number of strings when the string length in the string set is constant. A simple modification of the algorithm adopts the edit distance and computation of mismatch tolerance. A new similarity continuous interval Hamming distance (CIH) was introduced to clearly define maximal similar substrings.

We proposed a method for 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 these strings from huge strings. From computational experiments for genome sequences, we demonstrated the practical efficiency of the algorithm and the efficiency of the parallelization. From the comparison of genome sequences, we found similar long substrings from human and mouse genomes in a practically short time.

Acknowledgments We gratefully thank 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 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. We appreciate the advice concerned with chunks given by Koji Tsuda of Advanced Institute of Science and Technology, Japan. For the parallelization of the implementation, we would like to thank Yasuhiro Ike of Ybeat, Japan, for his help on the implementation.

References

1. Abrahamson K (1987) Generalized string matching. SIAM J Comput 16:1039–1051
2. Altschul FS, Gish W, Miller W, Myers EW, Lipman DJ (1990) Basic local alignment search tool. J Mol Biol 215:403–410
3. Altschul SF, Madden TL, Schaffer AA, Zhang J, Zhang Z, Miller W, Lipman DJ (1997) Gapped BLAST and PSI-BLAST: a new generation of protein database search programs. Nucleic Acids Res 25:3389–3402
4. Amir A, Lewenstein M, Porat E (2000) Faster algorithms for string matching with k mismatches. In: Symposium on discrete algorithms, pp 794–803

5. Assent I, Krieger R, Glavic B, Seidl T (2008) Clustering multidimensional sequences in spatial and temporal databases. Knowl Inf Syst 16:29–51
6. Brown P, Botstein D (2000) Exploring the new world of the genome with DNA microarrays. Nat Genet 21:33–37
7. Faloutsos C, Barber R, Flickner M, Hafner J, Niblack W, Petkovic D, Equitz W (1994) Efficient and effective querying by image content. Intell Inf Syst 3:231–262
8. Feigenbaum J, Kannan S, Strauss M, Viswanathan M (1999) An approximate L1-difference algorithm for massive data streams. In: Proceedings of FOCS99
9. Fleischmann RD, Adams MD, White O, Clayton RA (1995) Whole-genome random sequencing and assembly of Haemophilus influenzae Rd. Science 28:496–512
10. Johnson DS, Mortazavi A, Myers RM, Wold B (2007) Genome-wide mapping of in vivo protein-DNA interactions. Science 31:1441–1442
11. Kawahara D, Kurohashi S (2006) Case frame compilation from the web using high-performance computing. In: Proceedings of the 5th international conference on language resources and evaluation (LREC2006), pp 1344–1347
12. Koga H, Ishibashi T, Watanabe T (2007) Fast agglomerative hierarchical clustering algorithm using locality-sensitive hashing. Knowl Inf Syst 12:25–53
13. Liu J, Goss S, Murray G (1994) Similarity comparison and analysis of sequential data. In: IEEE international conference on expert systems for development, pp 138–143
14. Manber U, Myers G (1993) Suffix arrays: a new method for on-line string searches. SIAM J Comput 22:935–948
15. Muthukrishnan S, Sahinalp SC (2000) Approximate nearest neighbors and sequence comparison with block operations. In: Proceedings of 32nd annual ACM symposium on theory of computing, pp 416–424
16. Muthukrishnan S, Sahinalp SC (2002) Simple and practical sequence nearest neighbors under block edit operations. In: Proceedings of CPM2002
17. Pearson WR (2000) Flexible sequence similarity searching with the FASTA3 program package. Methods Mol Biol 132:185–219
18. Popendorf K, Osana Y, Hachiya T, Sakakibara Y (2007) Murasaki-homology detection across multiple large-scale genomes. In: Fifth annual RECOMB satellite workshop on comparative genomics (San Diego, USA, 2007)
19. Shivakumar N, Garcia-Molina H (1996) Building a scalable and accurate copy detection mechanism. In: International conference on digital libraries, proceedings of the first ACM international conference on digital libraries, pp 160–168
20. Yamada S, Gotoh O, Yamana H (2006) Improvement in accuracy of multiple sequence alignment using novel group-to-group sequence alignment algorithm with piecewise linear gap cost. BMC Bioinform 7:524
21. Yamada T, Morishita S (2003) Computing highly specific and mismatch tolerant oligomers efficiently. Bioinformatics Conference 2003
22. Yamada T, Morishita S (2005) Accelerated off-target search algorithm for siRNA. Bioinformatics 21:1316–1324

Author Biography

Takeaki Uno was born in 1970, in Tokyo, Japan, and received a Dr of Science degree from Masakazu Kojima of Tokyo Institute of Technology, Japan, 1998, for the work of theoretical algorithmic research on speeding up enumeration algorithms. He was research associate of department of Industrial and Management Science of Tokyo Institute of Technology, from 1998 to 2001, and researched on discrete optimization algorithms and financial engineering. He has been an associate professor of National Institute of Informatics, Japan, from 2001, and researched on graph algorithms, graph classes, data mining algorithms, bioinformatics, and data analyze algorithms. Major prizes he received are Best Implementation award of FIMI04, Best Paper Runner-up Award of PAKDD2008, and Best paper award of ISAAC2008. His latest researches are on developing theoretically supported practically fast algorithms for processing huge data, and he provides efficient implementations on his homepage, including those that got awards in international conferences.