Vol.2013-AL-145 No.2
2013/11/6

An Algorithm for Finding Frequently Appearing Long String Patterns from Large Scale Databases

Takeaki Uno[1], Juzoh Umemori[2] and Tsuyoshi Koide[3]

[1] uno@nii.jp, National Institute of Informatics
2-1-2, Hitotsubashi, Chiyoda-ku, Tokyo 101-8430, Japan
[2] , Fujita Health University, Aichi 470-1192, Japan
[3] , National Institute of Genetics, Mishima, Shizuoka 411-8540, Japan

Abstract. We propose a new algorithm for solving frequent string mining problem with allowing approximate matches. The algorithm first computes the similarity between the strings in the database, and enumerate clusters generated by similarity. We then compute representative strings for each cluster, and the representatives are our frequent strings. Further, by taking majority votes, we extend the obtained representatives to obtain long frequent strings. The computational experiments we performed show the efficiency of both our model and algorithm; we were able to find many string patterns appearing many times in the data, and that were long but not particularly numerous. The computation time of our method is practically short, such as 20 minutes even for a genomic sequence of 100 millions of letters.

1 Introduction

String data is one of the most popular data types, including texts, word sequences, genome sequences, and so on. When we analyze string data, we sometimes want to find frequently appearing string patterns, called frequent strings, in the data. In the bioinformatics field, such frequent strings are called "motifs", and many studies have been done on the algorithms for finding motifs. Frequent string mining is the task to find frequent strings. The problem of finding patterns frequently appearing in a given database is called frequent pattern mining, and widely studied. The first problem was to find itemsets from a transaction (subset family) database [1]. The recent development of algorithms enables us to handle large scale databases, as shown in the implementation competitions [5]. More structured patterns such as sequences and graphs are also considered, and many algorithms are proposed [7, 11, 16].

Actually, there are at most $n(n+1)/2$ substrings for a string of n letters, thus frequent strings can be enumerated in polynomial time. Moreover, constructing a suffix array [8] makes it possible to obtain all the frequent strings in a compact form in $O(n)$ time. This computational efficiency has enhanced development in many areas in this field, such as natural language processing.

On the other hand, for application to real-world data, we have to allow approximate matches, i.e., matches that include errors. In natural languages, many ambiguities occur in writing, and numerous human errors such as typos also occur. As a basic principle, genome sequences include errors. Consequently, it is exceedingly unlikely that long subsequences coming from the same origin will be exactly identical. In such cases, exact matches can give only short and trivial patterns. Unfortunately, frequent pattern mining cannot well deal with approximate matching for finding large patterns. For example, if we allow patterns to have at most θ errors in matching, huge number of short strings become frequent, since any string of θ letters can match anything in the string data. This disturbs the usual hill climbing algorithms, since they in principle have to explore all small patterns.For example, the algorithm of Mitasiunaite and Boulicaut[9] took one hour to one day, to extract frequent strings of length 5 to 10 with 2 or 3 errors, from string data of millions of letters. It would be very hard to find string patterns of length more than 100 with up to 10 errors by this approach.

On the other hand, if we allow errors according to the pattern length, such as ρl mismatches for strings of length l with a ratio ρ, hill climbing algorithms do not work since patterns do not satisfy the anti-monotone property. Fundamentally, frequent pattern mining with approximate match (soft occurrence) has a difficulty on the definition of the problem, since even if we ignore small patterns, there might be huge number of large frequent patterns. For example, when string "AAAA...AAA" is included in many times in the data, any string with short distance from it, such as "BBA...AAA" and "BAZ...AA", can be a frequent string. We should formulate the problem so that only some representatives of such strings will be output, however, defining representatives is difficult. Pattern mining aims to enumerate patterns interesting/valuable to human being, and its good mathematical definition is hard.

In this paper, we propose a new "method" for generating string patterns that may appear in the given string data many times. That is, we define our task, but do not formulate the problem precisely. Instead of that, we propose a process to generate string patterns from the input string data. Actually, many algorithms in bioinformatics take this approach (see a survey [14]). This approach is often taken in areas with mathematically undefined objectives, such as information retrieval, and web science. In algorithmic area, heuristic algorithms can be considered as this approach. For example, greedy algorithms have well-defined objectives, but their output can not be formulated as a simple mathematical terms. The algorithms are description of the output.

To design our algorithm, we observe that if a string approximately matches some substrings in the string data, the substrings are relatively similar to each other. This observation leads us that clusters of substrings generated by similarity can be regarded as the set of substrings that some frequent patterns match. On this basis, we enumerate the clusters first and then compute their representatives. The representatives are our model of frequent strings, and we show its completeness. We can prove that for any frequent string defined in a usual way, there is a representative not so different from the pattern. Furthermore, by taking majority votes, we extend the frequent strings in both directions to obtain longer frequent strings.

Actually, there are similar approaches in bioinformatics [6, 13, 14] However, because of the computational difficulty, they can deal only with small data, can find few patterns, or have some heuristics such as removing very frequent small strings from the data, that loses completeness and disables to deal with other string data. Moreover, they do not use enumerational approaches, thus the obtained patterns usually do not have the completeness. For example, let us see the algorithm HomologMiner [6] that is designed to find clusters of similar substrings. It first removes all frequently appearing short strings called *repeat sequences* from the input genome sequences, by looking at the library of repeat sequences. It then computes the similarity between all substrings, and cluster the substrings. Because of the heavy pairwise comparison (even though they avoid heavy DP), they needed up to ten hours. δ-free pattern mining aims to reduce the number of patterns by pruning patterns including patterns with similar occurrences[4], but the reduction ratio is limited, say 1/10.

To achieve a high computational performance, we keep our task to be simple. We address two tasks in this paper; find short string patterns that have Hamming distances of at most the given threshold d to many substrings of the input string data, and to find many string patterns that have short edit distances to many substrings of the input string data. The former can be seen as motifs, and the latter can be seen as consensus sequences, that have been extensively studied in bioinformatics. By using Hamming distance as error criterion, we can find similar substring pairs in a very short time. There are several algorithms for this task, especially in bioinformatics such as BLAST [2, 3] and FASTA [10]. Among these, we chose the algorithm of [15] because of its high performance. Actually, the other algorithms basically use exact matches to find approximate matches, thus they do not fit our purpose; if we use these algorithms, we should use exact match directly, to make everything clear and simple. The algorithm we chose can terminate in a few minutes even for 10 million of genome sequences of 30 letters with Hamming distance threshold 2.

Our computational experiments show that our algorithm runs in a practically short time, say 10 minutes, even for large scale data of up to one million letters, for many kinds of real world data such as genome sequences, natural language texts, and word sequences. The patterns found by our method are long, and their number is quite small compared to those found by usual frequent pattern mining. The number of patterns found is usually in the order of 100 or 1,000. This range is suitable for the practical use of pattern mining, i.e., ten is too small (it should be solved by optimization approaches), and one million is too much.

2 Preliminaries

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 $|S|$. 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 a string $S[i]S[i+1]\ldots S[j]$, and is denoted by $S[i, j]$. When $i < 1$ or $i > |S|$, $S[i]$ is a special letter '\$' that is not in Σ. For example, when string S is $ABCDEFG$, $S[3] = C$, and $S[4, 6] = DEF$. When $j < i$, we define $S[i, j]$ to be the empty string, and when $i < 1$, the $S[i, j]$ starts with '\$'. For two strings S_1 and S_2, the *concatenation* of S_2 to S_1 is a string S given by appending S_2 to the tail of 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.

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 $S_1[i] \neq S_2[i]$. Such letters are called the *mismatches* of S_1 and S_2, and the positions of mismatches are called the *mismatch positions* of S_1 and S_2. The *edit distance* (Levenshtein distance) between S_1 and S_2 is defined by the minimum number of operations to transform S_1 to S_2, where the operations are to change/insert/delete one letter. Suppose that we are given a distance measure. For a string S, the substrings in the string data that have a short distance to S are called the *occurrences* of S. A set of occurrences of S is called an *occurrence set* of S.

3 Model and Method

We formulate our frequent string pattern mining problem as follows.

Problem: Cluster-based Frequent String Mining
input string set $\mathcal{D} = \{S_1, \ldots, S_m\}$
1. (Cluster enumeration) Find groups (clusters) of substrings of strings in $\mathcal{D}$ such that the substrings in the group are similar to each other.
2. (Centroid computation) Compute a string similar to all the substrings in a cluster, for each cluster.

This would involve a computational difficulty on the cluster enumeration step if the length of strings to be found was long. To cope with this difficulty, we observe that "long similar strings have a common short string that is similar to substrings of each long string." In particular, if the long strings are similar in terms of a distance that allows more general errors, the short string might be similar in terms of a distance that allows more specific errors. Here "errors" includes mismatches, insertions, deletions, copies, duplications, moves, etc. "Specific errors" means a class composed of few of these, and "general errors" a class comprising many of these. The latter is observed frequently in genome sequences; two long genome sequences similar in the edit distance usually have substrings similar in the Hamming distance. For example, BLAST [2, 3] and FASTA [10] that are de facto standard similarity search tools in bioinformatics, use this principle to find long similar strings; they find a pair of substring that are exactly the same, and check whether they are a part of similar long substrings. This principle motivates us to use short frequent substrings as seeds of long frequent substrings. That is, we first find short frequent substrings and then extend each short string in both directions until it becomes not similar to its occurrences. Accordingly, the problem for long strings is formulated as follows.

Problem: Cluster-based Long Frequent String Mining
input string set $\mathcal{D} = \{S_1, \ldots, S_m\}$
1. (Seed enumeration) Find groups (clusters) of substrings of short length in the set of strings such that the substrings in the group are similar to each other.
2. (Centroid computation) For each cluster, compute a string similar to all the substrings in a cluster, and extend the string in both directions until it is not similar to many substrings in the cluster.

If a seed is too short in length, it may be included in many different frequent string patterns. On the other hand, the seeds have to be shortened as much as possible to achieve the computational efficiency. This trade-off is a key to achieving practical efficiency with this model.

4 Algorithm

We propose an efficient algorithm for solving these problems. The key idea is using a multi-sorting algorithm [15] for finding similar short substrings. This algorithm inputs a set of strings of the same length l, and finds all the string pairs having a Hamming distance of at most d. Applying this algorithm to all substrings in the string data gives a graph representation of similarity among all substrings. A number of clustering algorithms can be used to find clusters, but here we use a simple method for adapting large-scale data.

Hereafter, we assume that a substring has length l. For a substring of length l, let $N(S)$ be the set of substrings of length l of a string in the string data such that the Hamming distances to S is at most d. Note that S is included in $N(S)$. To make a cluster, we choose a substring S maximizing $|N(S)|$ which corresponds to a maximum degree vertex in the graph representing the similarity relation among the substrings. We then make a cluster by $N(S)$. We repeat this after removing the substrings in the cluster from the candidates of S. The process ends when $|N(S)|$ achieved a value of less than σ, that is a given threshold value.

For each cluster, we compute its representative string $C(N(S))$, which we call *centroid*, by using a majority voting algorithm described as follows. Here, $v(\{s_1, \ldots, s_k\}, \rho) = c$ if $k\rho$ letters s_j in $\{s_1, \ldots, s_k\}$ are c, and is the wildcard otherwise.

```
Algorithm StringVote ($\{S_1, \ldots, S_k\}, \rho(> 0.5)$)
1. for $i := 1$ to $l$, set $S^*[i]$ to $v(\{S_1[i], \ldots, S_k[i]\}, \rho)$
2. return $S^*$
```

The length of $C(N(S))$ is l, and its ith letter is determined by the voting, i.e., the letter appearing most often in all the ith letters of strings in $N(S)$. For example, if the first letters of three strings in a cluster are all A, B, and A, the majority is A, and the first letter of the centroid $C(N(S))$ becomes A. For the case in which the majority letter is not especially significant, we use a threshold value ρ so that if the majority letter appears less than $\rho|N(S)|$ in the substrings in $N(S)$, we set the letter to the "wildcard", instead of the majority letter.

Instead of the voting, we can simply use S itself as the centroid. However, in some case, S can be different from the common string. For example, $S = BBAAAAAA$, and all strings in $N(S)$ are $AAAAAAAA$ except for one, $BBBBAAAA$. The representative of this cluster should be $AAAAAAAA$, but S is different from it. However, in the analysis explained in the next section, choosing S has an advantage. Using the voting algorithm, our algorithm for Cluster-based Frequent String Mining problem is described as follows.

```
Algorithm FreqString ($\mathcal{D}, l, d, \sigma, \rho$)
    $\mathcal{D}$: string set, $l$: substring length, $d$: Hamming distance threshold, $\sigma$: degree threshold, $\rho$: voting threshold
1. $\mathcal{S} :=$ all substrings of length $l$ in a string in $\mathcal{D}$
2. call multi-sorting algorithm to compute $N(S)$ for all substrings in $\mathcal{S}$
3. choose a substring $S$ in $\mathcal{S}$ maximizing $|N(S)|$
4. if $|N(S)| < \sigma$ stop
5. compute $C(N(S))$ by StringVote ($N(S), \rho$)
6. $\mathcal{S} := \mathcal{S} \setminus N(S)$
7. if $\mathcal{S} \neq \emptyset$ go to 3
```

Let N denote the number of output pairs by the multi-sorting algorithm, i.e., $N = (\sum_{S \in \mathcal{S}} |N(S)|)/2$. In practical terms, the multi-sorting algorithm terminates in $O(l(|\mathcal{S}| + N))$ time if l is sufficiently large and d is sufficiently small, e.g., in the case where $l = 30$ and $d = 3$ for genome sequences [15]. Steps 1, 3, 4 and 7 can be done in $O(|\mathcal{S}|)$ time in total. Since step 6 can be done by removing all pairs including a substring in $N(S)$, and no pair is removed twice, thus accounting computation time for step 6 results $O(N)$. Step 5 can be done in $O(l\ |N(S)|)$ time, thus needs $O(lN)$ time in total. In summary, the computation time depends on the time for multi-sorting algorithm, and can be expected to be $O(l(|\mathcal{S}| + N))$ in practice. We can see this by looking at the result of our experiments; the memory usage that is linear to N and the computation time.

For finding longer frequent substrings, we extend the seeds as follows. Suppose that we obtain centroid S^* from a cluster $N(S)$. Then, for all substrings in the cluster, we compute the majority letter in the same way as given above, among all letters following the substrings in strings in $\mathcal{D}$. Note that if some substrings are located at the end of some texts, we consider that any following letter is a special letter '\$' meaning "void". The majority letter, or the wildcard, is the $l + 1$ th letter of the extended centroid. We further compute the second following letter, the third following letter and so on. We want to stop this when the majority is often not significant. We stop the extension if there are more than θ wildcards in the last l letters of the extending centroid, where θ is a given threshold. The stopping criteria, majority threshold, and some other parameters or methods of the extension can be changed as the user likes, but to make the similarity of the strings uniform, we propose to use the same setting as that for the centroid computation and the multi-sorting algorithm. The algorithm is described as follows.

```
Algorithm StringVoteExt ($S, \{p_1, \ldots, p_k\}, l, \rho(> 0.5), \theta$)
1. $i := 0$; $j := l - 1$; $S^* :=$ StringVote ($\{S[p_1, p_1 + l - 1], \ldots, S[p_k, p_k + l - 1]\}, \rho$)
2. $i := i - 1$; set $S^*[i + 1]$ to $v(\{S[p_1 + i], \ldots, S[p_k + i]\}, \rho)$
3. if $S^*[i + 1] \neq \$$ and $S^*[i + 1, i + l]$ has wildcards of at most $\theta$, go to 2.
4. $j := j + 1$; set $S^*[j + 1]$ to $v(\{S[p_1 + j], \ldots, S[p_k + j]\}, \rho)$
5. if $S^*[j + 1] \neq \$$, and $S^*[j - l + 1, j + 1]$ has wildcards of at most $\theta$, go to 4.
6. return $S^*[i + 2, j]$
```

After computing the centroid, we remove all substrings in the cluster $N(S)$ from $\mathcal{S}$. We further remove the substrings from $\mathcal{S}$ that are included in the substring S' obtained by extending a substring in the cluster in both directions as the same length as the extent of the centroid. This is because in the next phase, the substrings removed would make a cluster, and the extension of the seed obtained by the cluster would be the same as that for the currently obtained centroid. The algorithm FreqLongString for Cluster-based Long Frequent String Mining is obtained by modifying steps 5 and 6 of FreqString as follows.

```
5. compute $C(N(S))$ by StringVoteExt ($\mathcal{D}$, { positions of $N(S)$ in $\mathcal{D}$}, $l$, $\rho, \theta$)
6. extend the substrings in $N(S)$ in the same length as for Step 5, and remove
all substrings included in the extended substrings from $\mathcal{S}$.
```

The additional computation time needed for this process is for the extension and the removal of the substrings. This needs, roughly, $O(Nh)$ time where h is the average length of the extension. Thus, not so many centroids are extended in long lengths, and therefore we may not lose computational efficiency.

To achieve efficiency of the computation, we are motivated to use exact matches as seeds, i.e., substrings having no mismatches, since finding exactly the same substrings is much easier than finding similar substrings. However, this may result in a loss of model efficiency. If we use identical substrings, the cluster size will be small, and the voting may not work well. If we use much shorter identical substrings, they are possibly included in other non-similar strings, thus the voting will be done for non-similar strings. Using similar substrings may exclude non-similar strings, and increase the cluster size.

5 Completeness of the Model

This section shows a completeness of the model, by stating that any frequently appearing string is similar to at least one output pattern. Suppose that $\mathcal{C}$ is the set of strings of length l obtained by FreqString, from a string set $\mathcal{D} = \{S_1, \ldots, S_m\}$.

Table 1. Results for X chromosome; length of 1,000 letters

	d	200	400	800	1600	3200	6400	12800	25600	51200	102400	204800	291572
time (sec)	0	0.45	0.9	1.84	3.71	7.23	14.9	30	65	141	339	889	1559
	1	0.51	1.08	2.23	4.63	9.23	19.5	42	99	296	1197		
	2	0.58	1.3	3.09	7.22	15.4	34	76	203	804			
	3	0.91	2.05	5.18	13.1	28	66	155	456	2082			
memory (MB)	0	72	72	72	72	80	112	190	350	659	1284	1983	2705
	1	72	72	72	72	81	112	190	350	1217	3684		
	2	72	72	72	72	72	81	112	195	2879			
	3	72	72	72	72	72	89	126	316	5212			
#short patterns	0	7	19	25	25	68	83	178	461	1603	4879	7003	15021
	1	42	101	179	291	471	803	1696	5935	15836	41939		
	2	53	166	288	433	1168	1820	3770	14874	31193			
	3	88	415	1022	2537	1978	4239	6945	15879	58460			
#long patterns	0	0	1	2	3	4	7	11	26	56	111	273	341
	1	1	5	13	28	45	66	107	234	569	1019		
	2	2	4	20	38	52	136	229	493	980			
	3	3	7	29	55	64	130	254	349	398			
average length	0	0	1123	667	429	301	335	220	218	193	244	304	277
	1	87	302	205	158	142	142	142	173	180	204		
	2	106	432	191	167	161	133	130	146	152			
	3	126	278	161	122	121	130	113	123	219			

Lemma 1. *For any S and $N(S)$, the Hamming distance between S and $C(N(S))$ is less than d/ρ. In particular, it is less than $2d$ if $\rho > 1/2$.*

Proof. Let P be the set of mismatch positions of S and $C(N(S))$. For each p in P, let $Z(p)$ be the set of $S' \in N(S)$ such that at the position p, S' has a letter different from S. We then see that $|Z(p)| \geq \rho(|N(S)|)$. Since each $S' \in N(S)$ can have at most d positions mismatching S, $\sum_{p \in P} |Z(p)| \leq d(|N(S)| - 1)$. Thus

$$d|N(S)| - d \geq \sum_{p \in P} |Z(p)| \geq \sum_{p \in P} \rho|N(S)| = |P|\rho|N(S)|,$$

thus we have $|P| < d/\rho$ $\square$

Lemma 2. *Suppose that at least σ substrings in strings in $\mathcal{D}$ have Hamming distances at most $d/2$ to S. If S is a string in $\mathcal{S}$, there is always a string $S^* \in \mathcal{C}$ s.t. the Hamming distance between S and S^* is at most $(1 + 1/\rho)d$.*

Proof. From the assumption we have that $|N(S)| \geq \sigma$. Thus, S is included in $N(S')$ for some S' in Step 5 of algorithm FreqString. Since the Hamming distance between S and S' is at most d, and from Lemma 1, the Hamming distance between S' and $C(N(S'))$ is at most d/ρ, we have that the Hamming distance from S to $C(N(S'))$ is at most $(1 + 1/\rho)d$. $C(N(S'))$ is a member of $\mathcal{C}$, therefore the statement holds. $\square$

Lemma 3. *Suppose that d is even and at least σ substrings in strings in $\mathcal{D}$ have Hamming distances at most $d/2$ to S. There is always a string $S^* \in \mathcal{C}$ s.t. the Hamming distance between S and S^* is at most $(1.5 + 1/\rho)d$.*

Proof. Let $\mathcal{H}$ be the set of substrings of strings in $\mathcal{D}$ such that the Hamming distance to S is at most $d/2$. Since $|\mathcal{H}| \geq \sigma$ and any two strings in $\mathcal{H}$ have Hamming distance of at most d, at least one of $\mathcal{H}$ belongs to $N(S')$ that induces a string $S^* \in \mathcal{C}$. From Lemma 1, the Hamming distance from S to S^* is at most $(1.5 + 1/\rho)d$. $\square$

From these lemmas, we can see that the output of our algorithm covers all frequent strings. When we choose S as the centroid of $N(S)$, the upper bound in the statement of the lemma can be improved. From the proof of the above lemma, there is a string S' such that Hamming distance to S is at most $d/2$, and included in some cluster $N(S^*)$. Thus, the Hamming distance from S to S^* is bounded by d if S is a substring of a string in $\mathcal{D}$, and $1.5d$ otherwise.

6 Experiments

Our implementation is coded in C, and no sophisticated libraries such as binary trees are used. All experiments were done on a 3.2 GHz Core i7-960 computer with a Linux operating system having 24GB of RAM memory. Note that we did not use multi-cores in the experiment. The codes and the instances we used can be found available at the author's website (`http://research.nii.ac.jp/~uno/codes.html`).

We used a genome sequence and a word sequence as instances. The genome sequence was the human X chromosome taken from National Center for Biotechnology Information (NCBI) database. It was 143,733,266

Table 2. Results for word sequence; length of 1,000 letters

	d	25	50	100	200	400	800	1600	3200	4700
time	0	0.1	0.19	0.46	1.26	2.93	22.9	43	175	267
(sec)	1	0.22	0.6	1.93	7.22	22.4	110	388	2978	
	2	1.72	6.23	27	120	466	2155			
memory	0	36	36	36	36	36	62	99	257	354
(MB)	1	36	36	36	72	52	126	330	1575	
	2	36	36	43	107	280	885			
#short	0	82	145	499	1905	3717	8673	20773	66915	86540
patterns	1	296	949	3660	12639	32825	92408	228485	734335	
	2	1898	4572	10242	21067	40413	85325			
#long	0	36	74	271	696	1409	3128	6417	16526	19665
patterns	1	72	168	339	762	1493	3226	6425	15571	
	2	89	177	347	715	1455	3224			
average	0	13	16	14	17	18	17	17	17	18
length	1	16	15	16	16	16	14	14	15	
	2	14	14	13	13	13	12			

Table 3. Results for change in pattern length (X chromosome); degree is average number of similar patterns, and deg<2 is ratio of patterns that have at most two occurrences

l, d	10,0	15,0	15,1	15,2	20,0	20,1	20,2	30,0	30,1	30,2	30,3	50,0	50,2	50,4	50,6
time(sec)	690	37	192	1391	22	62	174	30	42	76	155	63	82	151	366
memory (MB)	1072	150	439	1551	97	181	340	112	112	112	197	121	124	124	106
#short patterns	2466	512	11915	164853	314	4168	6808	83	834	1975	4743	34	113	586	1404
#long patterns	1388	150	1011	770	86	302	0	27	107	229	254	2	11	88	106
average length	27	141	75	45	167	113	0	220	142	130	113	360	157	69	57
degree	0	5	0	0	11	5	0	8	14	13	9	1	1	90	95
deg<2	98	37	69	89	22	25	0	72	13	11	17	0	0	0	0

letters in length, and comprised 4 kinds of letters and a special letter 'N' meaning the wildcard (except comment lines). The instances we generated from the X chromosome were doubled by attaching its reverse, since genome sequences are often similar to the reverse of subsequence of the other sequences. The word sequences were generated from the benchmark data named "20 Newsgroups" (`http://people.csail.mit.edu/jrennie/20Newsgroups/`). With the exception of dashes and periods, all symbols were removed from the data, and each word was assigned a unique ID. The word sequence was the sequence of ID's of the words in the data. The alphabet size of the word sequence was 132,859, and the length was 5,454,672.

Tables 1 and 2 show the experimental results for the performance in terms of the changes of the input size and d. Since the computation time and the memory usage does not differ much between FreqString and FreqLongString, we show those of FreqLongString. We set the pattern length l to 30 on Table 1 and 12 on Table 2, σ to 10, ρ to $1/2$, θ to 2, and found the extended centroids for the cases of $d = 0, 1, 2, 3$, and $d = 0, 1, 2$, respectively. The instances were generated by taking substrings starting from the beginning, and were written in length of 1,000 letters. Tables 3 and 4 show the results obtained by the changes of the seed length l. We use the instances of length 6,400,000 for the X chromosome, and 1,600,000 for the word sequence.

The computation time was almost linear in the length except for large instances with large d, and even though the length is hundred millions, the computation time was comparatively short. For large scale data, the number of similar substrings was quite huge, thereby the computation time was long. In such cases, we should use much larger l and d, for not losing the efficiency as shown below. The number of patterns did not substantially explode, and the average pattern length was stable. In testing the appearance of the patterns we obtained in the genome sequence, we counted the substrings such that the edit distance to the pattern string was at most the 10% of the pattern length. Few patterns appeared few times, say 5–10 times, but other patterns appeared more than 30 times. For the increase of d, the increase in the pattern number and pattern length appeared to saturate at some point. This implies that in practice we do not have to care about large d.

We can find much more patterns by increasing d, especially for genome sequences, but too large d results in a long computation time. Thus, using small d would be a good solution. In genome sequences, we found many short patterns with small l, and few long patterns with large l. This is caused by that for small l, there are too many similar substrings for a substring. Such many substrings would be a part of non-similar long strings, thereby we could find only their common short substrings. For large l, we might not unify the groups, but when d is small, especially $d = 0$, we could find only few groups composed of quite similar substrings, and thus obtained very few patterns. With slightly large d and non-short l, we can find many sufficiently long patterns within quite short

Table 4. Results for change in pattern length; word sequence

l, d	8,0	12,0	12,1	18,0	18,1	18,2	18,3	25,0	25,2	25,4	40,0	40,2	40,4	40,6
time(sec)	2573	43	387	17.6	32	121	872	14	33	148	27	36	65	2731
memory (MB)	1557	110	380	3398	60	138	558	61	62	69	63	63	64	64
#short patterns	9380	6041	5351	6031	2484	3419	3357	3262	1143	2263	3261	428	562	804
#long patterns	6585	6417	6425	2499	3799	3754	3845	983	1696	2277	425	533	731	804
average length	14	17	14	18	20	17	18	15	14	16	17	17	15	13
degree	0	0	0	0	0	0	0	0	0	0	0	0	0	0
deg< 2	100	100	100	98	99	100	100	99	99	97	100	100	96	100

Table 5. Results with using interleaving sampling; X chromosome

	d	3200	6400	12800	25600	51200	102400	204800	291572
time	0	2.53	5.23	10.7	22.6	47	102	246	386
(sec)	1	3.04	6.38	13.7	30	75	216	680	1316
	2	3.85	9.46	20.8	51	165	701	2550	
	3	6.21	13.9	33	92	374	2108		
#long	0	1	3	4	37	152	399	995	1313
patterns	1	31	48	72	218	563	1116	2566	3330
	2	46	115	193	454	1102	1454	4278	
	3	104	171	284	614	1052	1929		
average	0	1710	977	769	353	378	393	413	386
length	1	189	200	178	234	262	262	293	280
	2	160	164	161	215	215	217	227	
	3	139	135	136	181	193	211		

time. This result supports our motivation to use similar short substrings for frequent string mining. In particular, we can see many copies/quotations in the word sequences, and consequently we found many groups of copies for the case of $d = 0$, and the large average length of patterns. By introducing similarity, we could find shorter patterns, that would not come from the groups of "copy and paste".

We also tested the similarity between patterns discovered. For each pattern S, we counted the number of strings S' similar to it; such that S (or S') had an edit distance of at most $0.1|S|$ to a substring of S'. We tested the output in the second experiment, and show results in the last two rows of Tables 3 and 4. The first row shows how many patterns are similar to one pattern on average, and the second row shows the fraction of patterns having at most two other similar patterns. In all cases, each pattern is similar to at most 15% of other patterns on average, and there are several patterns not similar to other patterns. The patterns we obtained from the data are in some sense independent to each other. In word sequences, and genome sequences with $d = 0$, almost all patterns have no similar pattern. It comes from that we found only groups of quite similar substrings, and "copy and paste", thus the strings in different groups are not similar to each other.

The computation time can be reduced by using "interleave sampling" of substrings proposed in [15]. By using the method, we can find continuous pairs of substrings having small Hamming distance, i.e., we find pairs of substrings S_1 and S_2 such that $S_1[i, i + l]$ and $S_2[i, i + l]$ have a Hamming distance of at most d for any i. When we find such string pairs from short string pairs, we never miss sufficiently long continuously similar substrings, even if we take only $1/c$ of all substrings, for some constant c such as $c = l^{1/2}$. This enables us to fasten the computation without losing the efficiency. In Table 5 and Table 6, we could reduce the computation time to 1/5, without decreasing the number of patterns found, and the length of the patterns.

Comparison with other methods

The comparison of the methods for this problem is actually very difficult, since they share the same task, but the problems formulations are different. For example, many algorithms in bioinformatics finds the frequent string from exact matches. They are faster than those of approximate matches, and the accuracy and completeness might be less, but no one can be sure about this. Some algorithms reduce the input data by domain specific background knowledge, and/or some corpus for efficient computation. It is hard to have a good criteria for the evaluation of algorithms. Instead of a good evaluation criteria, we just show the results of the computational experiments in Table 7, that are reported in literature. The numbers are approximations.

7 Conclusion

We formulate a simple frequent string pattern mining problem with approximate match, and propose an algorithm. By using the multi-sorting algorithm for the similarity computation, our algorithm could perform quite well even for large-scale real-world string data. The number of patterns we obtained in this approach was not excessively

Table 6. Results with using interleaving sampling; X chromosome

	d	400	800	1600	3200	4700
time	0	1.21	10.9	18.4	56	93
(sec)	1	6.41	35	108	677	
	2	125	547	2191		
#long	0	777	1733	4881	13409	16655
patterns	1	1351	3095	6182	14798	
	2	1338	3164	5908		
average	0	16	4881	17	17	18
length	1	16	6182	15	15	
	2	15	5908	14		

Table 7. Comparison of the performance of the algorithms

algorithm	distance	errors	problem size	#patterns	CPU time
FreqLongString	edit distance	10%	100MB	1000	1000
(hill climbing) [9]	edit distance	2 or 3	>10MB	huge	1 day
HomologMiner [6]	Hamming distance	-	30MB	few	10 hours
RepeatScout [12]	edit distance	-	50MB	few	2 hours

large such that the patterns obtained are tractable. The experimental results showed that by introducing the similarity, we can obtain much more long patterns or non-trivial patterns without drastically increasing the computational cost. We used Hamming distance for the similarity evaluation, but the obtained patterns are actually frequent even in the term of the edit distance. It is noteworthy that using simple criteria makes it possible to find interesting, large patterns with more sophisticated criteria. Interesting future research is to extend this algorithm to handle more sequence-like string patterns to accept more ambiguous data such as natural language texts.

Acknowledgments

We gratefully thank to Prof. Hiroki Arimura of Hokkaido University, and Prof. Takashi Washio of Osaka University for their valuable comments. This research is supported by JST PRESTO and Funding Program for World-Leading Innovative R&D on Science and Technology, Japan.

References

1. R. Agrawal, H. Mannila, R. Srikant, H. Toivonen and A. I. Verkamo, “Fast discovery of association rules”, Advances in Knowledge Discovery and Data Mining, Chapter 12, AAAI Press / The MIT Press (1996).
2. S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman, “Basic local alignment search tool”, Journal on Molecular Biology **215**, 403–410 (1990).
3. S. F. Altschul, T. L. Madden, A. A. Schäffer, J. Zhang, Z. Zhang, W. Miller, D. J. Lipman, “Gapped BLAST and PSI-BLAST: a new generation of protein database search programs”, Nucleic Acids Research, **25**, 3389–3402 (1997).
4. C. Hebert, B. Cremilleux, “Mining Frequent δ-Free Patterns in Large Databases”, LNAI **3735**, 124–136 (2006).
5. B. Goethals, “The FIMI repository”, `http://fimi.ua.ac.be/` (2003).
6. M. Hou, P. Berman, C. H. Hsu and R. S. Harriset, “HomologMiner: Looking for Homologous Genomic Groups in Whole Genomes”, Bioinformatics **23**, 917–925 (2007).
7. A. Inokuchi, T. Washio and H. Motoda “An Apriori-Based Algorithm for Mining Frequent Substructures from Graph Data”, LNAI **1910**, 13-23, (2000).
8. U. Manber and G. Myers, “Suffix Arrays: A New Method for On-line String Searches”, SIAM J. on Comp., **22**, 935–948 (1993).
9. I. Mitasiunaite and J.-F. Boulicaut, “Introducing Softness into Inductive Queries on String Databases”, Databases and Information Systems IV, 117–132. IOS Press (2007).
10. W. R. Pearson, “Flexible sequence similarity searching with the FASTA3 program package”, Methods in Molecular Biology **132**, 185–219 (2000).
11. J. Pei, J. Han, B. Mortazavi-Asl, H. Pinto, Q. Chen, U. Dayal, and M.-C. Hsu, “PrefixSpan: Mining Sequential Patterns Efficiently by Prefix-Projected Pattern Growth”, ICDE 2001, 215–224 (2001).
12. A. L. Price, N. C. Jones, P. A. Pevzner, “De novo Identification of Repeat Families in Large Genomes”, Bioinformatics **21** (Suppl 1), 351–358 (2005).
13. F. P. Roth, J. D. Hughes, P. W. Estep, and G. M. Church, “Finding DNA Regulatory Motifs within Unaligned Noncoding Sequences Clustered by Whole-genome mRNA Quantitation”, Nature Biotechnology **16**, 939–945 (1998).
14. S. Saha, S. Bridges, Z. V. Magbanua, D. G. Peterson, “Computational Approaches and Tools Used in Identification of Dispersed Repetitive DNA Sequences”, Tropical Plant Biol. (2008) doi: 10.1007/s12042-007-9007-5.
15. T. Uno, “Multi-sorting Algorithm for Finding Pairs of Similar Short Substrings from Large-scale String Data”, Knowledge and Information System **25**, 229–251 (2010).
16. J. Wang and J. Han, “BIDE: Efficient Mining of Frequent Closed Sequences”, ICDE 2004, 79–90 (2004).