An Efficient and Fast Algorithm for Mining Frequent Patterns on Multiple Biosequences

Wei Liu, Ling Chen

An Efficient and Fast Algorithm for Mining Frequent Patterns on Multiple Biosequences

Wei Liu[1,2], Ling Chen[1,3]

[1] Institute of Information Science and Technology, Yangzhou University , Yangzhou, China
[2] School of Information Technology, Nanjing Xiaozhuang University , Nanjing, China
[3]National Key Lab of Novel Software Tech, Nanjing University, Nanjing, China
yzliuwei@126.com, lchen@yzcn.net

Abstract. Mining frequent patterns on biosequences is one of the important research fields in biological data mining. Traditional frequent pattern mining algorithms may generate large amount of short candidate patterns in the process of mining which cost more computational time and reduce the efficiency. In order to overcome such shortcoming of the traditional algorithms, we present an algorithm named MSPM for fast mining frequent patterns on biosequences. Based on the concept of primary patterns, the algorithm focuses on longer patterns for mining in order to avoid producing lots of short patterns. Meanwhile by using prefix tree of primary frequent patterns, the algorithm can extend the primary patterns and avoid plenty of irrelevant patterns. Experimental results show that MSPM can achieve mining results efficiently and improves the performance.

Keywords: Biological sequence; Frequent Pattern Mining; Primary Patterns

1 Introduction

Biosequence patterns usually correspond to some important functional (or structural) elements[1] such as conserved sequence patterns, repeated patterns or combinative patterns etc. Hence it is very significative to find such patterns in protein family analysis, transcriptional regulation analysis, and genome annotation etc. The task of biosequence pattern mining [2] is also the key technique for gene recognition, biosequence functional prediction and interactions explanation between sequences. It is one of the most important research areas in biosequence data mining.

In the area of data mining, lots of sequential pattern mining algorithms have been proposed in recent years. At present the sequential pattern mining algorithms are mainly classified as two categories: one is for frequent patterns mining on single sequence; the other is for mining in multiple sequences. The former can mine frequent patterns only for single sequence[3-4], and is unable to synchronously analyze the relation between frequent patterns from a certain sequence and those contained in the other sequences. Such analysis is common and necessary in biosequence data mining. For the latter, according to the definition[5] by Agrawal and Srikan in 1995 based on the analysis of transaction data: given a sequence set and a user-specified

support threshold, the problem of sequential pattern mining is to find all frequent subsequences, that is to say, the counts of the subsequence appeared in the sequence set are not less than the minimal support threshold. In 1996, Strikant et al. proposed GSP (generalized sequential pattern mining)[6] which introduced the concept of time and level-wise constraints based on Apriori algorithm. It mines all frequent patterns by the use of bottom-up and breadth-first search strategy. But when the sequence database is a large-scale one, large amount of candidates could be produced and the database should be frequently scanned. Especially when the sequences contain long patterns, large amount of short candidate patterns may be generated , which could cause the problem would be intractable and of lower efficiency. In order to solve this problem, in 2000 Pei et al. put forward an algorithm named Prefixspan[7] based on pattern growth approach. It adopts divide and conquer method and continuously produces much smaller projected databases so as to mine frequent patterns. Since no candidates are produced in the algorithm, search space is greatly reduced. Its main cost is on the construction of projected databases and its performance is much higher than Apriori-based algorithms. Other recent works on sequential pattern mining algorithms have been surveyed in [8] by Han et al.

However, because of the particularity and variety of mining requirements for biological data, the previous developed methods can not be applied directly to the large-scale biological data mining. Therefore extensive efforts have been devoted to developing some special mining algorithms for biological data, such as PTR-based algorithms[9-10] by Apostolico et al., ATR-based algorithms[11-18] by Delgrange et al. and TRFinder algorithm[19] by Beason. Later Kurtz presented REPuter[15] algorithm based on suffix tree which overcame the limitation of the length of input sequences. It was based on sequence alignment technique but could hardly find those frequent repeats among DNA sequences. In 2007, Wang et al made researches on searching for the similar repeated segments[20] and then introduced a new criteria of similarity and the concept of SATR(segment-similarity based approximate tandem repeats). They designed an algorithm SUA_SATR [21] based on SUA with no limitations on pattern length during the searching process. Moreover, with the same similarity, the algorithm is faster than other traditional algorithms for the same DNA sequence, although its efficiency should be improved. In 2007 Xiong et al. proposed BioPM algorithm[22] specially for protein sequence mining. They introduce the concept of multiple supports so as to overcome the disadvantages of traditional algorithms and improve its performance and efficiency. But when the minimal support becomes lower, it can not keep its high efficiency since numerous projected databases are constructed. In addition, the algorithm still produces large numbers of irrelevant short patterns during the mining process. In 2009, Guo et al. addressed MBioPM algorithm[23] which is an improvement of BioPM algorithm. Based on a pattern partitioning scheme, the algorithm successfully avoids constructing large amount of projected databases. But when the lengths of the patterns exceed k, it requires a large buffer for frequent patterns mining which resulted in huge memory space cost. Moreover, it also takes large amount of time to align the existing patterns with those in the buffer. All these large time-space costs will cause the low efficiency of the algorithm.

To overcome the problems of traditional sequential pattern mining algorithms mentioned above, we present a fast and efficient algorithm named MSPM for multiple

biosequence mining. The algorithm mines all frequent patterns rapidly based on the prefix tree of primary frequent patterns which reflects more biological meanings. Our empirical studies on the tested data from pfam protein database show that MSPM algorithm can obtain higher performance and efficiency than the traditional mining algorithms.

2 Definitions and Concepts

2.1 The primary patterns

Definition 1: Let $\sum$ be the alphabet, and $S = \{S_1, S_2, ..., S_n\}$ be a string of $\sum$. Assuming x is a character in $\sum$, if for string S, there exists integers $1 \le i_1 < i_2 < ... < i_m \le n$ such that $s_{i1} = s_{i2} = ... = s_{im} = x$, then we call $s_x(k) = "s_{ik} s_{ik+1} ..... s_{i(k+1)-1}"$ the k^{th} primary pattern of S with respect to x.

Example1: Let $\sum = \{a, b, c\}$ and $S = "bacaabcbab"$, then the primary patterns of S with respect to character a are $s_a(1) = "ac"$, $s_a(2) = "a"$, $s_a(3) = "abcb"$, $s_a(4) = "ab"$.

From the definition, we can easily get the following lemma.

Lemma 1 For a string S, let its character set be $C(S) \subseteq \sum$. For an $x \in C(S)$, suppose there are n_x primary patterns with respect to x in S, then we have $\sum_{i=1}^{n_x} |s_x(i)| \le |S|$ and $\sum_{x \in C(S)} n_x = |S|$.

Lemma 2 For a string S, summation of the lengths of the primary patterns with respect to all characters in $C(S)$ will satisfy:

$$\sum_{x \in C(S)} \sum_{i=1}^{n_x} |s_x(i)| \le |C(S)| \bullet |S|.$$

Proof: By Lemma 1, we can see that $\sum_{x \in C(S)} \sum_{i=1}^{n_x} |s_x(i)| \le \sum_{x \in C(S)} |S| = |C(S)| \bullet |S|$

Q.E.D

Because $C(S) \subseteq \sum$ and $|C(S)| \le |\sum|$, it is can be deduced that $\sum_{x \in C(S)} \sum_{i=1}^{n_x} |s_x(i)| \le |\sum| \bullet |S|$.

Lemma 3 For a string S, the average length of the primary patterns with respect to all characters in S will be not more than $|C(S)|$.

Proof: From lemma 2, we know that the summation of the lengths of all the primary patterns of S satisfies $\sum_{x\in C(S)}\sum_{i=1}^{n_x}\left|\sigma_x(i)\right| \le \left|C(S)\right| \bullet \left|S\right|$. Furthermore, by lemma 1 we also know that the number of total primary patterns of S is equal to $\left|S\right|$. Therefore the average length of all the primary patterns of S will be not more than $\left|C(S)\right|$.

Q.E.D

From the lemmas mentioned above, we can see that all primary patterns can be intercepted in $O(\left|S\right|)$ time by scanning S. The framework of the algorithm for intercepting the primary patterns is described as follows:

Algorithm Intercept（S）
Input: string S;
Output: the primary patterns of S;
begin
For every $x \subseteq C(S)$ do
k=1;
$\sigma_x(k) = \varnothing$;
Let the first position of x appeared in S be l;
$\sigma_x(k) = \sigma_x(k) \cup \{\sigma_i\}$
i=l;
repeat
While $(\sigma_i \neq x \cup \sigma_i \neq \varnothing)$ do
$\sigma_x(k) = \sigma_x(k) \cup \{\sigma_i\}$;
i=i+1;
End while
k=k+1 ;
$\sigma_x(k) = \{\sigma_i\}$;
i=i+1;
Until $\sigma_i = \varnothing$
End for
End

2.2 The table of primary patterns

After getting all primary patterns of S, we can further build a table of primary patterns for S. All the primary patterns are listed in the table in the lexicographic order so as to conveniently search.

Example 2 For the sequence S = "*bacaabcbab*" in example 1, after sorting all primary patterns of S, the table of primary patterns can be built as shown in Table 1.

Table1. The table of primary patterns for *s*

Num	s_m	*loc*
1	*a*	4
2	*ab*	9
3	*abcb*	5
4	*ac*	2
5	*b*	10
6	*ba*	8
7	*bacaa*	1
8	*bc*	6
9	*caab*	3
10	*cbab*	7

In the table, each entry is a vector (Num, s_m, loc), where *Num* is the index of the entry in the table, s_m is the primary pattern and *loc* denotes the start position of s_m in *S*.

All the primary patterns obtained by algorithm intercept(*S*) should be sorted so as to be arranged in lexicographic order. By Lemma 1, we know that there are $|S|$ patterns. Suppose that $|S| = n$, it costs $O(n \cdot \log n)$ time for sorting. Fortunately, for biosequences, $|\Sigma|$ is a constant integer. For instance, for gene sequences, $|\Sigma| = 4$ whereas for protein sequences, $|\Sigma| = 20$. Hence we can use radix sorting method. Obviously by lemma 3 we know that the average length of primary patterns is not more than $|\Sigma|$ and their length is imbalance. Because of each pattern with different lengths, the traditional radix sorting algorithm can't be applied straightforwardly. Therefore, we present the following sorting algorithm.

Algorithm2 Sort(s_m, s_m")

Input: s_m : the primary patterns of *S*;

Output: s_m" : the ordered primary patterns table;

Begin

Let the initial character of s_{mi} be χ_i and $l = |\Sigma|$. If there is $\chi_1 < \chi_2 < ... < \chi_l$ in Σ, then according to the initial characters of all patterns in s_m, we can divide them into some buckets as s_{m1}, s_{m2}, ⋯ , s_{ml};

$s_m' = \varnothing$

For *i*=1 to *l* do

After emitting the first character of s_{mi}, we can get a new string set s_{mi}'. Delete all the empty strings from s_{mi}' .

If $s_{mi}' = \varnothing$ then $s_m' = s_m' \cup s_{mi}'$ End if

End for

Sort($\boldsymbol{S}_m'$, $\boldsymbol{S}_m''$);

For i=1 to l do

For each string y in $\boldsymbol{S}_{mi}''$ do $y = y \& \boldsymbol{\chi}_i$; end for

endfor

Group $\boldsymbol{S}_{m1}''$, $\boldsymbol{S}_{m2}''$, … , $\boldsymbol{S}_{ml}''$ into $\boldsymbol{S}_m''$;

End

By lemma 2, we know that the summation of the lengths of all primary patterns is not more than $|C(S)| \bullet |S|$. Since the algorithm Sort classifies all the characters of every primary pattern exactly once, its complexity is $O(|C(S)| \bullet |S|)$. Because $|C(S)| \leq |\Sigma|$ is a constant, its complexity is just $O(|S|)$.

Example 3 Let D={S_1, S_2, S_3, S_4} be a set of strings, $\boldsymbol{S}_1$="*abcbac*", $\boldsymbol{S}_2$="*acbca*", $\boldsymbol{S}_3$="*bcbabc*" and $\boldsymbol{S}_4$="*acbabc*". Their primary pattern tables are shown in Table 2.

Table2. The primary pattern table of four sequences

The primary pattern table of S_1

Num	$\boldsymbol{S}_m$	*loc*
1	*abcb*	1
2	*ac*	5
3	*bac*	4
4	*bc*	2
5	*c*	6
6	*cba*	3

The primary pattern table of S_2

Num	$\boldsymbol{S}_m$	*loc*
1	*ab*	5
2	*acbc*	1
3	*b*	6
4	*bca*	3
5	*cab*	4
6	*cb*	2

The primary pattern table of S_3

Num	$\boldsymbol{S}_m$	*loc*
1	*abc*	4
2	*ba*	3
3	*bc*	1,5
4	*c*	6
5	*cbab*	2

The primary pattern table of S_4

Num	$\boldsymbol{S}_m$	*loc*
1	*abc*	4
2	*acb*	1
3	*ba*	3
4	*bc*	5
5	*c*	6
6	*cbab*	2

2.3 Merging the Primary Pattern tables

After getting all primary pattern tables of multiple sequences, we can merge them in order to obtain a merged primary pattern table as shown in Table 3.

Table3. The merged primary pattern table of multiple sequences

Num	S_m	*Seq*
1	*ab*	2
2	*abc*	3,4
3	*abcb*	1
4	*ac*	1
5	*acb*	4
6	*acbc*	2
7	*b*	2
8	*ba*	3,4
9	*bac*	1
10	*bc*	1,3,4
11	*bca*	2
12	*c*	1,3,4
13	*cab*	2
14	*cb*	2
15	*cba*	1
16	*cbab*	3,4

In Table3, each entry is denoted as a vector (Num, S_m, Seq), where *Num* is its index in the table, S_m is the primary pattern and *Seq* denotes the sequences where S_m appears.

2.4. The primary frequent patterns mining

Definition 2(Distribution support[22]):Given a set of biosequences *D* and a subsequence *T*, the distribution support of subsequence *T* in *D* is defined as $dis\_\sup D(T) = |\{s \mid s \in D, T \subset s\}|$, which is the number of strings in *D* containing the subsequence *T*.

Definition 3: A primary pattern *P* is called a frequent primary pattern if *dis_supD(P)*≥ *mindis_sup*, where positive integer *mindis_sup* is the minimum support.

By the above definitions, we can build a frequency table for a string set *D*. For instance the frequency table for primary patterns of string set *D* in Example 3 is as shown in Table 4..

Table4. The frequency table for primary patterns on multiple sequences

S_m	*Freq*
a	4
ab	4
abc	3
abcb	1
ac	3

acb	2
acbc	1
b	4
ba	3
bac	1
bc	4
bca	1
c	4
cab	1
cb	4
cba	3
cbab	2

In table4, each entry is denoted as$\left(\boldsymbol{s}_m, \mathit{Freq} \right)$, where $\boldsymbol{s}_m$ is a primary pattern and *Freq* denotes the distribution support of $\boldsymbol{s}_m$ in the set of multiple sequences *D*.

Let *dis_sup* be 2, and then based on Table 4, we can easily mine all primary frequent patterns:

a（frequency: 4）, *ab*（frequency: 4）, *abc*（frequency: 3）, *ac*（frequency: 3）, *acb*（frequency: 2）; *b*（frequency: 4）, *ba*（frequency: 3）, *bc*（frequency: 4）; *c*（frequency: 4）, *cb*（frequency: 4）,*cba*（frequency: 3）,*cbab*（frequency: 2）.

Based on above primary frequent patterns and table2, we can easily build the following two-dimension table respect to primary frequent patterns on multiple sequences.

Table5. The two-dimension table of primary frequent patterns on multiple sequences

Sequence / Frequent patterns	S_1	S_2	S_3	S_4
a	{1,5}	{1,5}	{4}	{1,4}
ab	{1}	{5}	{4}	{4}
abc	{1}	∅	{4}	{4}
ac	{5}	{1}	∅	{1}
acb	∅	{1}	∅	{1}
b	{2,4}	{3,6}	{1,3,5}	{3,5}
ba	{4}	∅	{3}	{3}
bc	{2}	{3}	{5}	{5}
c	{3,6}	{2,4}	{2,6}	{2,6}
cb	{3}	{2}	{2}	{2}
cba	{3}	∅	{2}	{2}
cbab	∅	∅	{2}	{2}

The element$\{ l_1 , l_2 , ..., l_k \}$ in the table denotes the set of starting positions of primary pattern in *S*. For example, the element {1, 5} in the first row of table5

denotes the primary pattern *"a"* are respectively in the 1st and 5th positions of S_1. Furthermore, we can build the following table similar to Table1:

Table 6. The primary frequent pattern table of *D*

Num	S_m	*Loc_set*
1	*a*	{1,5}, {1,5}, {4}, {1,4}
2	*ab*	{1}, {5}, {4}, {4}
3	*abc*	{1}, ∅, {4}, {4}
4	*ac*	{5}, {1}, ∅, {1}
5	*acb*	∅, {1}, ∅, {1}
6	*b*	{2,4}, {3,6}, {1,3,5}, {3,5}
7	*ba*	{4}, ∅, {3}, {3}
8	*bc*	{2}, {3}, {5}, {5}
9	*c*	{3,6}, {2,4}, {2,6}, {2,6}
10	*cb*	{3}, {2}, {2}, {2}
11	*cba*	{3}, ∅, {2}, {2}
12	*cbab*	∅, ∅, {2}, {2}

In Table 6, each entry is denoted as a vector (Num, S_m, loc_set), where *Num* is the index of this entry in the table, S_m is a primary pattern and *loc_set* denotes the set of start positions of S_m in each sequence.

3 The Prefix Tree of Primary Patterns

3.1 Aggregation Vector and Its Operations

To mine all the frequent patterns in a given biosquence set, a prefix tree is used. In order to fully comprehend the construction process of the prefix tree, first we give the following definitions.

Definition 4: Let $T = (T_1, T_2, \ldots, T_n)$ be a vector, where T_i is a set, we call *T* is an aggregation vector.

Definition 5: Given an aggregation vector *T*, its support function *D*(*T*) can be defined as the number of nonempty sets in *T*, that is: $D(T) = \left|\{T_i | T_i \neq \varnothing; 1 \le i \le n\}\right|$.

Definition 6: Assuming $T = (T_1, T_2, \ldots, T_n)$ and $S = (S_1, S_2, \ldots, S_n)$ are two aggregation vectors , their intersection is defined as

$$T \cap S = (T_1 \cap S_1, \ldots, T_n \cap S_n).$$

Definition 7: Given a set $t = (t_1, t_2, \ldots, t_k)$ and a number *l*, the addition operation of them is defined as $t + l = \{t_1 + l, t_2 + l, \ldots, t_k + l\}$.

Definition 8: The addition operation on a given aggregation vector $T = (T_1, T_2, \ldots, T_n)$ and a number l is defined as $T + l = \{T_1 + l, T_2 + l, \ldots, T_n + l\}$.

3.2 The Prefix Tree Construction of Primary Frequent Patterns

By algorithm 2 can not only sort the primary patterns, but also can obtain the number of each primary pattern. And during the recursive process, the number of primary patterns with the prefix of substring y can also be obtained. By summing up the numbers of the primary patterns with the prefix of a certain character or substring y, the scope $[l, u]$ of those primary patterns in the sorted primary pattern table can also be computed. Therefore, while implementing algorithm 2, we can also expediently construct a prefix tree of primary patterns.

Definition 9 For a table of primary patterns, its corresponding prefix tree is a rooted tree. The path from the root to each leaf denotes a primary pattern and each edge in the path denotes a character. Children of the same node denote different characters.

Once the prefix tree is construced, all the primary pattern can be obtained by listing all characters in sequence of each path, one.

From the definition of the prefix tree, we can design an algorithm for building the prefix tree of primary patterns. It is a recursive process. The algorithm repeatedly builds the prefix subtree for the nodes from the root to the leaves level by level. Therefore, we present an algorithm named node-extend (*S*, *d*) which constructs a prefix tree rooted as *d* for the set *S* of primary patterns. The framework of the algorithm node-extend is shown as follows.

Algorithm3:Algorithm node-extend(*S, d*);
Input: *S*: the set of primary patterns and their starting positions in the alphabetical order table. Each primary pattern in *S* takes the form of（*s*, *e*）which denotes an entry of the primary pattern table, where *s* is a primary pattern and *e* is the starting position set of *s*.
d : the node to be extended.
mindis-sup: the minimal support threshold;
Output: the prefix tree rooted at *d* for the set *S* of primary patterns;
Begin
Suppose there are *r* different first characters in the strings of *S*; then the strings of *S* can be divided into *r* groups according to their first characters;
Let the group with the first character x_i be S_i;
For *i*= 1 to *r* do
T_i=Φ;
For all strings *P* of S_i do
Emit the first character x_i of *P* and obtain a string *P*';
If $P' \neq \Phi$ then $T_i = T_i \cup \{P'\}$ End if
If $|T_i| \geq \min\text{dis-sup}$ then
Generate a child d_i for *d*;
Label the edge(*d*, d_i) using character x_i;

The starting position set $E^{(d)}$ on the node d defines assigned as the starting position set of strings in T_i;
Node-extend(T_i, d_i);
End if
End for
End for
End

Let the primary patterns table of D be $D(H)$, the prefix tree of primary patterns of D can be built by calling the recursive process Node-extend($D(H)$, *root*).

Example 4 For the frequent primary patterns in Table 6, the prefix tree of primary frequent patterns by the above recursive algorithm can be built as shown in Fig. 1.

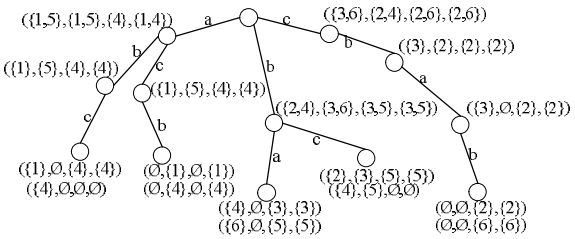

Fig.1. Prefix tree of primary frequent patterns for D

In the prefix tree shown in fig.1, each character x in the edge denotes the character or substring that the node represents, $\{l_1, l_2, ..., l_k\}$ denotes the set of starting position sets of primary patterns with the prefix of x in D. Each inner node b stores a starting node aggregation vector $T^{(b)} = \{T_1^{(b)}, T_2^{(b)}, ..., T_k^{(b)}\}$, where $T_i^{(b)}$ is the starting positions set of the substring corresponding to the i^{th} string.

Each leaf node b stores a node-depth D_b. Moreover, there are a start-node aggregation vector $T^{(b)} = \{T_1^{(b)}, T_2^{(b)}, ..., T_k^{(b)}\}$ and a terminative node aggregation vector $E^{(b)} = \{E_1^{(b)}, E_2^{(b)}, ..., E_k^{(b)}\}$, where $E^{(b)} = T^{(b)} + D_b$. $E_i^{(b)}$ is the terminative positions set of the substring corresponding to the i^{th} string. Listing all characters on the edges of each path from the root to each leaf, one primary frequent pattern can be obtained.

4 Mining Algorithm

4.1 The Generic Frequent Patterns Mining

The limitation of primary frequent pattern is that there is no same character as the first character in the pattern. The primary frequent pattern table on multiple sequences

mentioned above can only mine primary frequent patterns, but can not mine all the frequent patterns. We can use the prefix tree of frequent primary patterns to mine larger frequent patterns by aggregation vector operations on the vectors at each node. For example, in Fig.1 the frequent pattern *"cbabc"* can be obtained by expanding the primary frequent pattern *"cbab"*. After getting the prefix tree as fig.1, we can easily find the path of pattern *"cbab"* and then its terminative node vector ($\varnothing$, $\varnothing$, {6}, {6}) also can be obtained at the leaf. Next we return to the root of the tree and perform intersection operations respectively on the starting node vectors of the root's sons and the terminative node vectors of the primary pattern. For instance in fig.1, after the intersection operation on the child *c* , ({3,6}, {2,4}, {2,6}, {2,6}) $\cap$ ($\varnothing$, $\varnothing$, {6}, {6})=($\varnothing$, $\varnothing$, {6}, {6}), its support is 2. This means the pattern is frequent. Therefore we can expand frequent pattern *"cbab"* by the operation $\{cbab\} \cup \{c\} = \{cbabc\}$ and a longer frequent pattern *"cbabc"* is obtained.

From the analysis mentioned above, based on the prefix tree of primary frequent patterns we propose an algorithm named Span(*s*, $E^{(s)}$, *a*) for mining all frequent patterns. It is a recursive process. The algorithm starts from the root and extends patterns for each node level by level. The framework of the algorithm Span(*s*, $E^{(s)}$, *a*) is as follows:

Algorithm 4:Algorithm Span(*s*, $E^{(s)}$, *a*)
Input: *a*: the node where *a* is; *s*: the string that be extended;
$E^{(s)} = \{E_1^{(s)}, E_2^{(s)}, ..., E_k^{(s)}\}$: the terminative vector set of *s*;
Freq-set: the set of frequent substrings;
mindis-sup : the minimal support threshold;
Output: the *Freq-set* after update
Begin
For each child *b* of *a* do
Assuming that the starting vector of *b* is $T^{(b)} = \{T_1^{(b)}, T_2^{(b)}, ..., T_k^{(b)}\}$ and the character on edge (*a, b*) is *x*.
If *b* is not an leaf node then
num=0;
For *i*= 1 to *k* do
$F_i = T_i^{(b)} \cap E_i^{(s)}$;
If $F_i \neq \Phi$ then *num*=*num*+1; endif
End for
Assuming that the set is $F = \{F_1, F_2, ..., F_k\}$
If $|num| \geq dis\min - \sup$ then
If $x \neq \varnothing$ then
Freq-set=*Freq-set* $\cup \{s * x\}$;
Span $(s * x, F, b)$;
Else
Span (s, F, b) ;
End if

End if
Else
$E^{(b)} = \{E_1^{(b)}, E_2^{(b)}, ..., E_k^{(b)}\}$ */
$E^{(s)} = E^{(b)}$;
Span (s, $E^{(s)}$, root);
End for
End

Based on algorithm 4, we present an algorithm Freq-Mining for mining frequent patterns on the string set *S*.

Algorithm 5: Algorithm Freq-Mining（*S,root*）
Input: *S*: the strings for mining;
root: the root of prefix tree of primary sub-patterns for *S*;
Output: *Freq-set*: the set of frequent pattern of *S*;
Begin
Freq-set= Φ;
$E=\{T_1, T_2, \ldots, T_k\}$; where $T_i=\{1,2,\ldots,|S_i|\}$;
Span (Φ, *E*, *root*) ;
End.

4.2. MSPM Algorithm

In this section, we present algorithm MSPM（Multiple Sequential Pattern Mining for Biological Data）for mining the frequent patterns in a biosequence set *S*. Algorithm MSPM first mine all frequent primary patterns. Then these frequent primary patterns can be extended to get all the frequent patterns. Framework of algorithm MSPM is as follows.

Algorithm MSPM（*S, minloc_sup*）
Input: *S* ; an biosequence set;
mindis_sup: the minimal support threshold;
Output: *Freq-set* : the set of all frequent patterns;
Begin
For each biosequence S_i in *S* do
Intercept (S_i);
Sort(S_{mi}, S_{mi}"");
End for
Merging all primary pattern tables of multiple sequences and sorting all patterns by calling algorithm 2;
Building the frequency table of primary patterns and getting set *H* of all primary frequent patterns;
Building the two-dimension table respect to frequent primary patterns and then the frequent primary pattern table of *S* can be obtained;

Constructing the prefix tree *T* of frequent primary patterns by calling the recursive algorithm Node-extend(*S(H), root*);

Freq-Mining（*S,root*）;

End.

5 Experimental Results and Analysis

To test the efficiency of our algorithm MSPM, we made an experiment on two groups of data for comparisons. In the experiment of first group, the traditional Apriori algorithm, BioPM algorithm, MbioPM algorithm and our algorithm MSPM are tested to compare their performance. The experimental results show that the change of minimal support threshold makes a slight impact on our algorithm MSPM. The test on the second group compares the computation times of algorithm BioPM, MbioPM algorithm and MSPM to validate that our algorithm is more effective and faster than other ones under the same minimal support threshold.

5.1 Test Data and Computational Environment

All testing data in the experiments are from *pfam* protein database （http://pfam.sanger.ac.uk/）, and the average length of our selected protein sequences is 1000[24] which ensures the experimental validity. All experiments were conducted on a 3.00GHZ Pentium 4 with 2.00GB memory. All codes were complied using Microsoft Visual C++6.0.

5.2 Comparison of the Performance

The purpose of the first group experiment is to prove that the change of minimal support threshold makes a slight impact on our algorithm MSPM. We took protein sequence samples from three protein families (*G-alpha, Calici Coat, Glyco_hydro*_19）of *pfam* database as testing data. During the process of experiment, we chose 50 protein sequences with the similar length respectively from three protein families as a testing data set so as to ensure the diversity of the test data. With each specified support, we tested 50 sequences using three algorithms. Fig.2 shows the running time of algorithms Aprior[6], BioPM[22], MbioPM[23] and MSPM for the same biological data set with different minimal supports.

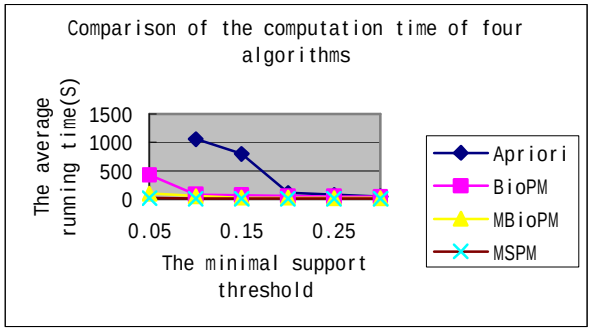

Fig.2 Comparison of the computational times by the four algorithms under different minimum supports

From fig.2 we can see that four algorithms becomes faster when the minimal support increases. But MSPM algorithm is rather stable and always faster than the other three ones especially with the lower minimal supports. That is because with lower minimal supports, Apriori algorithm would produce more short candidate patterns which necessarily result in the growth of computation time and inefficiency. Similar to Apriori, the inefficiency of BioPM algorithm is caused by the fact that it must spend more time in frequently building projective databases. MBioPM algorithm mines from the patterns with a certain length but it takes large amount of time for comparing the existing patterns with the patterns in buffer zone. While MSPM uses primary patterns with longer length for mining, which avoids producing lots of short patterns and more candidates. Through the merging operation and pattern growth method based on the prefix tree of primary frequent patterns, the algorithm can accelerate mining procedure. The above analysis shows that our algorithm MSPM is more efficient, stable and faster.

5.3 Comparison of the computational Time

We test the algorithms on the second data group to compare their computational time with the same minimal support threshold. The experimental results show our algorithm MSPM is more effective. We took protein sequence samples with the similar length respectively from ten protein families（*Globin, short chain dehydrogenase, SBP_bac*_9, *Acety-ltransferase, GNAT* family, *ATPase* family, *Glyco_hydro*_19, *G-alpha, Calici coat, Birna* VP2）of *pfam* database. We fixed the minimal support as 15%, and compared the computational times by the algorithms BioPM, MBioPM and MSPM on data sets consisting of different number of sequences from 100 to 600. Fig.3 shows the comparison of their computation times.

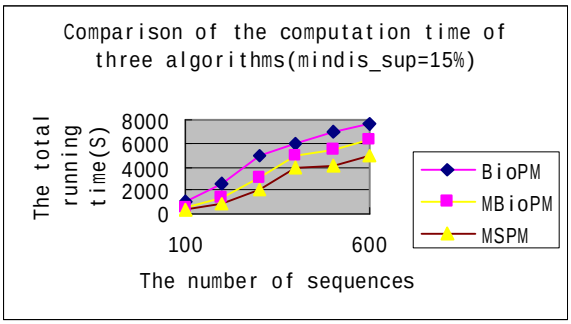

Fig.3. Comparison of the computational times by the three algorithms on data sets consisting of different number of sequences

It can be observed from fig.3 that the running speeds of three algorithms become slower when the number of sequences increases. But the MSPM algorithm is rather stable and always faster than other two algorithms. The reason is BioPM algorithm produces lots of short patterns during the process of iterations at first and then frequently builds projective databases. All of these operations will necessarily increase the time cost of the algorithm. MBioPM algorithm mines from the patterns with a certain length but it takes large amount of time to align the existing patterns with the patterns in buffer zone when mining the k-length frequent patterns. It also repeatedly creates or eliminates buffer zone during the process of pattern growth. All of these cause the algorithm be less efficient. MSPM mines frequent patterns from primary patterns with longer length, which avoids producing lots of short candidate patterns and reduces the computation time. The merged operation and pattern growth based on the prefix tree of primary frequent patterns also avoid producing the redundant patterns and greatly improve mining efficiency of our algorithm.

6 Conclusion

Based on the mining requirements and characteristics of biosequential patterns, we present an efficient and fast algorithm MSPM for multiple biosequence mining. The algorithm first builds multiple primary pattern tables according to all primary patterns of multiple sequences. Then through merging operation, the corresponding merged primary pattern table can be obtained. Based on this merged primary pattern table, all primary frequent patterns can be easily mined. Furthermore, we also present an algorithm for general frequent patterns mining on multiple sequences. The algorithm constructs a prefix tree of primary frequent patterns based on primary frequent patterns table and thereby mines all frequent patterns rapidly by the use of aggregation vector operations on the prefix tree. During the process of pattern growth, the algorithm neither produces any candidates nor constructs a mass of projective databases. This makes our mining results reflect more biological meanings and also improves mining efficiency. Our empirical studies on the tested data from *pfam*

protein database show that MSPM algorithm can obtain higher performance and efficiency than the traditional mining algorithms.

Acknowledgement: This research was supported in part by the Chinese National Natural Science Foundation under grant No. 60673060, Natural Science Foundation of Jiangsu Province under contract BK2008206.

References

1. Brejova B,DiMarco C,Vinar T,et al. Finding Patterns in Biological Sequences[R]. Technical report, University of Waterloo, 2000.
2. Bajesy P,Han J w,Liu L et.al. Survey of biodata analysis from a data mining Perspective[C]. In:Wang J T L,Zaki M J,Toivonen H T T,et al,eds. Data Mining in Bioinformaties. England: Springer-Verlag London LTD,2005:9-39.
3. Mannila H, Toivonen H. Discovering generalized episodes using minimal occurrences[C]. In Proceedings of the Second International Conference on Knowledge Discovery and Data Mining (KDD '96). Portland, AAAIPress, 1996: 146-151.
4. Mannila H, Toivonen H, Verkamo A I. Discovery of frequent episodes in event sequences[J]. Data Mining and Knowledge Discovery, 1997,1:259-289.
5. Agrawal R,Srikant R. Mining sequential patterns.In:Yu PS,Chen ALP, eds. Proc. Of the 11th Int'l Conf. on Data Engineering [J]. Taipei:IEEE Computer Society,1995:3-14.
6. Srikant R, Agrawal R. Mining sequential patterns: Generalization and performance improvements [C]. In: Apers PMG, Bouzeghoub M, Gardarin G, eds. Advances in Database Technology, Proc. of the 15th Int'l Conf. on Extending Database Technology. London: Springer-Verlag, 1996:3-17.
7. Pei J, Han JW, Mortazavi-Asl B, Pinto H. Prefixspan: Mining sequential patterns efficiently by prefix-projected growth[C]. In: Proc. of the 17th Int'l Conf. on Data Engineering. Washington: IEEE Computer Society, 2001. 215-224.
8. Han J W, Cheng H, Xin D, Yan X. Frequent Pattern Mining: Current Status and Future Directions[J]. Data Mining and Knowledge Discovery. 2007,15(1):55-86.
9. Apostolico A, Prefarata F. Optimal off-line detection of repetitions in a string[J]. Theoretical Computer Science, 1983,22(3): 297-315.
10. Kolpakov R, Kucherov G. Finding maximal repetitions in a word in linear time[C]. In: Proc. of the 1999 Symp. on Foundations of Computer Science. Washington: IEEE Computer Society, 1999. 596-604.
11. Delgrange O, Rivals E. STAR: An algorithm to search for tandem approximate repeats[J]. Bioinformatics, 2004,20(16):2812-2820.
12. Kolpakov R, Kucherov G. Finding repeats with fixed gap[C]. In: Proc. of the 7th Int'l Symp. on String Processing and Information Retrieval (SPIRE). Washington: IEEE Computer Society, 2000. 162-168.
13. Kolpakov R, Kucherov G. Finding approximate repetitions under hamming distance[J]. Theoretical Computer Science, 2003,303(1): 135-156.

14. Krishnan A, Tang F. Exhaustive whole-genome tandem repeats search[J]. Bioinformatics, 2004,20 (16): 2702-2710.
15. Kurtz S, Choudhuri JV, Ohlebusch E, Schleiermacher C, Stoye J, Giegerich R. REPuter: The mani fold applications of repeat analysis on a genomic scale[J]. Nucleic Acids Reseach, 2001, 29(22):4633-4642.
16. Landau GM, Schmidt JP, Sokol D. An algorithm for approximate tandem repeats[J]. Journal of Computational Biology, 2001,8(1): 1-18.
17. Hertz GZ, Stormo GD. Identifying DNA and protein patterns with statistically significant alignments of multiple sequences[J]. Bioinformatics, 1999,15(7): 563-577.
18. Pevzner PA, Sze SH. Combinatorial approaches to finding subtle signals in DNA sequences[C]. In: Proc. of the 8th Int'l Conf. Intelligent System for Molecular Biology. San Diego: AAAI Press, 2000. 269-278.
19. Benson G..Tandem repeats finder: A program to analyze DNA sequences[J]. Nucleic Acids Research, 1999,27(2):573-580.
20. Wang D, Wang G, Wu QQ, Chen BC. Finding LPRs in DNA sequence based on a new index SUA[C]. In: Proc. of the IEEE 5th Symp. on Bioinformatics and Bioenginerering (BIBE 2005). Washington: IEEE Computer Science, 2005. 281-284.
21. Wang D, Zhao Y, Chen BC, Wang GR. SUA-Based algorithm for finding SATRs in DNA sequence[J]. Journal of Northeastern University (Natural Science), 2007,28(2):209-212 (in Chinese with English abstract).
22. Xiong Yun, Zhu Yangyong. BioPM: An Efficient Algorithm for Protein Motif Mining[C]. In: Proc. of ICBBE'07. [S. l.]: IEEE Press,2007. 394-397.
23. Guo Shun, Jiang Qingshan, Wang Beizhan, Shi Liang. A new pattern mining algorithm for protein sequences[J]. Computer Engineering, 2009.4,35(8), pp:208-210.
24. Bateman A, Birney E, Cerruti L, et al. The Pfam Protein Families Database[J]. Nucleic Acids Res., 2002, 30(1): 276-288.