Efficient Method for Mining Patterns from Highly Similar and Dense Database based on Prefix-Frequent-Items

Meng Han\textsuperscript{1,2}, Zhihai Wang\textsuperscript{1}, Jidong Yuan\textsuperscript{1}
\textsuperscript{1}School of Computer and Information Technology
Beijing Jiaotong University, Beijing, 100044, P.R. China
\textsuperscript{2}School of Computer Science and Engineering
Beifang University of Nationalities, Yinchuan, 750021, P.R. China
Email: compute2006_2@126.com

Abstract—In recent years, there are a great deal of efforts on sequential pattern mining, but some challenges have not been resolved, such as large search spaces and the ineffectiveness in handling highly similar, dense and long sequences. This paper mainly focuses on how to design some effective search space pruning methods to accelerate the mining process. We present a novel structure, Prefix-Frequent-Items Graph (PFI-Graph), which presents the prefix frequent items of other items in sequential patterns. An efficient algorithm PFI-PrefixSpan (Prefix-Frequent-Items PrefixSpan) based on PFI-Graph is proposed in this paper. It avoids redundant data scanning, and thus can effectively speed up the discovery process of new patterns. Extensive experimental results on some synthetic and real sequence datasets show that the proposed novel structure is substantially more efficient than PrefixSpan with physical-projection and pseudo-projection, especially for dense and highly similar sequence databases.

Index Terms—sequential pattern mining; dense database; highly similar sequence; long sequence; prefix frequent items

I. INTRODUCTION

Sequential pattern mining discovers frequent subsequences as patterns in a sequence database, and the subsequences whose occurrence frequency in the set of sequences is no less than minimum support threshold (called min_sup). It is an important data mining problem with broad applications [1-4], including the analysis of customer purchase patterns or Web access patterns\cite{5,6}, the analysis of sequencing or time related processes such as scientific experiments, natural disasters, and disease treatments \cite{7,8}, the analysis of DNA sequences \cite{9-12} and so on.

Many previous studies have contributed to the efficient mining of long sequence. Algorithms SPAM \cite{13} and LAPIN \cite{14} with sequence-extended sequence and itemset-extended sequence, FP-growth \cite{15} with FP-tree and PrefixSpan \cite{1} with projection-based are efficient for mining long sequence. Some studies contributed to mining highly similar sequence, such as SeqBDD \cite{16} with binary decision diagram.

In this paper, we present an efficient method of dense and highly similar sequential pattern mining called PFI-PrefixSpan (Prefix Frequent Items based PrefixSpan). It is based on the Prefix-Frequent-Items Graph (PFI-Graph) which is used to assist in early pruning and avoid duplicated projections. FPI-Graph is a directed acyclic graph and presents the prefix frequent items of other items in sequential patterns. This novel algorithm can reduce the scale of projected databases and the time of building projected databases through adding the pruning steps and reducing the scanning of certain specific sequential patterns production.

The rest of this paper is organized as follows: Section 2 reviews PrefixSpan algorithm. Section 3 discusses the novel structure: PFI-Graph and the algorithm PFI-PrefixSpan. Section 4 shows the experimental results of sequential pattern mining. Finally, the conclusion is provided in Section 5.

II. PREFIXSPAN ALGORITHM

The key advantage of PrefixSpan, an algorithm that examines the prefix subsequences and projects only their corresponding suffix subsequences into projected databases, is that it does not generate any candidates and only counts the frequency of local items. It utilizes a divide-and-conquer framework by creating subsets of sequential patterns that can be further divided when necessary \cite{19}.

<table>
<thead>
<tr>
<th>Sequence id</th>
<th>Sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>(1)(2)(3)</td>
</tr>
<tr>
<td>20</td>
<td>(1)(4)(3)(2)(3)</td>
</tr>
<tr>
<td>30</td>
<td>(5)(1)(2)(3)(2)</td>
</tr>
<tr>
<td>40</td>
<td>(5)(1)(3)(2)(3)</td>
</tr>
</tbody>
</table>

Table I. A SEQUENCE DATABASE

Project supported by Fundamental Research Funds for the Central Universities (No. 2013YJS032), Scientific Research Funds for the Ningxia Universities (No. NGY2013094), and Beijing Natural Science Foundation (No. 4142042).

© 2014 ACADEMY PUBLISHER
doi:10.4304/jsw.9.8.2080-2086
The major consuming of PrefixSpan is database projection, and the technique to reduce the size of projected databases is pseudo projection [1]. The idea is outlined as follows: instead of registering the index of the starting position of the projected suffix sequence in the sequence. Pseudo projection reduces the consuming of projection substantially when the projected database can fit in main memory.


We also find some replicated projected database, as shown in Table 2 and Figure 1. For example, the sequential patterns which begin with prefix (1) and the second item is (3) are: (1)(3), (1)(3)(2), (1)(3)(3), (1)(3)(4). Because the sequential patterns begin with prefix (3) are: (3), (3)(2), (3)(3), (3)(1). Therefore, the proceedings to find patterns (1)(3), (1)(3)(2), (1)(3)(3), (1)(3)(4) are duplicated work to find patterns (3), (3)(2), (3)(3), (3)(1). We call the item (1) is the Prefix Frequent Items (PFI) of item (3), denoted as $PF(3)=\{1\}$. We also find that the sequential patterns beginning with prefix (5) and the second item (3) are duplicated of the patterns beginning with (5) and the second item (1). For example, (5)(3), (5)(3)(2), and (5)(1)(3), (5)(1)(3)(2). Because finding all the local Prefix Frequent Items of all items consume much time, we just consider the PFI of length-1 patterns.

**Definition 1 (PFI).** Given a length-1 pattern $\alpha$, $\beta_1$ is a frequent item that appears in the prefix of $\alpha$ in some sequences. The $\text{counts}(\beta_1)$ is the number of sequences in which $\beta_1$ appears before $\alpha$. If $\text{counts}(\beta_1)=\text{support}(\alpha)$, then $\beta_1$ is one element of Prefix Frequent Items of $\alpha$, denoted as $PF(\alpha)=\{\beta_1, ..., \beta_i, ..., \beta_k\}$.
Definition 2 (CountsPFI). The number of elements in PFI($\alpha$) is called CountsPFI($\alpha$). The sum of all CountsPFI($\alpha$) is denoted as CountsPFI.

The major cost of PrefixSpan is the construction of projected databases. We give two ways to improve: (1) before the projected databases are constructed, adding the pruning step. Do not scan projected database when the projection sequence number is less than min_sup; (2) do not generate and scan the projected databases to some specific sequential patterns. For example, given a pattern $\alpha$, if PFI($\alpha$)=$\beta_1$, ..., $\beta_i$, is not null, then do not generate and scan the projected databases when the prefixes belong to prefix set ($\beta_i$)($\alpha$), ..., ($\beta_i$)($\alpha$). Generate the sequential patterns with regards to prefix ($\alpha$), and at the same time generate the patterns with regards to prefix set ($\beta_i$)($\alpha$), ..., ($\beta_i$)($\alpha$). For example, we know that item (1) belong to PFI(3), then we do not build projected database with regards to prefix (1)(3). We generate the patterns with regards to prefix (1)(3) until patterns with regards to prefix (3) generated, by adding item (1) to the beginning of patterns (3)(...) to generate (1)(3)(...). Therefore, our works aim to find out the PFI($\alpha$) of all 1-length pattern $\alpha$.

We can see that, large CountsPFI means more duplicated works in generating patterns. Therefore, we can reduce runtime and memory usage by avoiding these works.

Definition 3 (FromPattern). If $\beta$ belongs to Prefix Frequent Items of $\alpha$, then $\beta$ is a FromPattern of $\alpha$, that is $\beta$$\in$FromPattern($\alpha$). If $\gamma$ belongs to Prefix Frequent Items of $\beta$, then ($\gamma$)$\beta$ is a FromPattern of $\alpha$, that is ($\gamma$)$\beta$$\in$FromPattern($\alpha$). And so on, until no new Prefix Frequent Items are found. The FromPattern of $\alpha$ is denoted as FromPattern($\alpha$)=$\{\beta\}, \{\gamma\}$$\beta$, ...).

Definition 4 (ToPattern). If $\alpha$ belongs to Prefix Frequent Items of $\beta$, then $\beta$ is the ToPattern of $\alpha$, so $\beta$$\in$ToPattern($\alpha$), denoted as ToPattern($\alpha$)=$\{\beta\}, ...\}$.

PFI-Graph is a directed acyclic graph, and it records CountsPFI as weight of each node $\alpha$. Node in the graph is an item in sequential pattern which has FromPattern or ToPattern, as shown in Figure 2. The PFI-Graph of dataset new_orleans is shown in Figure 2, and the root of the graph is $\Phi$. The weight of a node (denoted as $\alpha$) is the CountsPFI($\alpha$), while its children nodes are items in ToPattern($\alpha$), and its parent nodes are items in FromPattern($\alpha$).

Take node (63) in Figure 2 as an example. The PFI of item (63) are (157) and (158), that is PFI(63)=$\{(157), (158)\}$. The weight of CountsPFI(63) of node (63) is 3. Here,3 is understood as 2+1, while 2 refers to the item number in PFI(63) and 1 refers to the item number in PFI(158). Meanwhile, (158) is the parent of (63). There are 3 links pointing to node (63) in the graph, and they are 157$\rightarrow$63, 158$\rightarrow$63 and 157$\rightarrow$158$\rightarrow$63. So the FromPattern(63)=$\{(157), (158), (157)(158)\}$ and ToPattern(63)=null. Node (56) has FromPattern and ToPattern, the FromPattern(56)=$\{(78), (208)\}$ and ToPattern(56)=$\{(170)\}$. The (170) is the most complex node in Figure 2, and PFI(170)=$\{(78), (56), (208)\}$. There are 7 links pointing to node (170) in the graph: 7 is equal to 3+3+1, the first value 3 refers to the item number in PFI(170), the second value 3 refers to the item number in PFI(56) and value 1 refers to the item number in PFI(208). Seven links pointing to (170) are: 157$\rightarrow$170, 56$\rightarrow$170, 208$\rightarrow$170, 157$\rightarrow$56$\rightarrow$170, 157$\rightarrow$208$\rightarrow$170, 208$\rightarrow$56$\rightarrow$70, and 157$\rightarrow$208$\rightarrow$56$\rightarrow$170. It is clear that the FromPattern(170)=$\{(157), (56), (208), (157)(208), (208)(56), (157)(208)(56)\}$ and ToPattern(170)=null. Therefore, when we scan the projected databases of prefix (170) and generate some sequential patterns (170)($\alpha$), we can generate some more patterns, such as:

\[
\begin{align*}
(157)(170)(&\alpha), (56)(170)(&\alpha), (208)(170)(&\alpha), \\
(157)(56)(170)(&\alpha), (157)(208)(170)(&\alpha), \\
\end{align*}
\]

Referring to the Figure 2, we can build a FromPattern tree on data new_orleans, as shown in Figure 3. The root node is $\Phi$. Recursively, if ($\alpha$) is a node in the tree, then its children are all nodes ($\alpha'$) such that ($\alpha'$)=$\beta$($\alpha$) and $\beta$$\in$FromPattern($\alpha$).

Based on the concepts of PFI, FromPattern and ToPattern, algorithm Prefix Frequent Items Prefix-Span (PFI-PrefixSpan) can be described as follows. There are two works to do in the novel algorithm, the first one is to build the PFI-Graph, in order to find all the
FromPattern(α) and ToPattern(α) (α ∈ PFI-Graph). The second one is to generate sequential patterns.

**Algorithm PFI-PrefixSpan** (PrefixSpan based on prefix frequent items)

**Input:** A sequence database S, and the minimum support threshold min_sup.

**Output:** The complete set of sequential patterns

**Method 1:** Call PFI-PrefixSpan (S).

The parameters S is the sequence database S.

**Steps:**
1. Scan S once, find all length-1 patterns α₁, …, α_N.
2. Scan S again, find all PFIs (i=1,…,N) and build PFI-Graph G.
3. For each αᵢ
   (a) Refer to G, find out ToPattern(αᵢ) (i=1,…,N) and FromPattern(αᵢ) (i=1,…,N).
   (b) Call PrefixSpan(αᵢ, l, S|αᵢ, ToPattern(αᵢ), FromPattern(αᵢ)).

**Method 2:** Call PrefixSpan(αᵢ, l, S|αᵢ, ToPattern, FromPattern).

The parameters are (αᵢ) α is a sequential pattern; (b) l is the length of αᵢ; and (c) S|αᵢ is the αᵢ-projected database; (d) when l=1, ToPattern is the ToPattern(αᵢ), otherwise ToPattern is null. (e) FromPattern is the FromPattern(αᵢ).

**Steps:**
1. Scan S|αᵢ once, find each frequent item, b.
2. For each frequent item b,
   (a) append b to αᵢ to from a sequential pattern αᵢ’, and output αᵢ’;
   (b) If FromPattern(αᵢ)≠null.
      For each βⱼ ∈ FromPattern(αᵢ), append βⱼ to αᵢ’ to sequential patterns αᵢ’’, and output αᵢ’’;
   (c) If b ∈ ToPattern(αᵢ), then stop, go to next b.
3. For each αᵢ’, construct αᵢ’-projected database S|αᵢ’, and call PrefixSpan(αᵢ’, l+1, S|αᵢ’, null, FromPattern).

There are two problems in the algorithm PFI-PrefixSpan. The first one is if βⱼ ∈ FromPattern(αᵢ) and γⱼ ∈ FromPattern(βⱼ), then it is needed to add all γⱼ to αᵢ’’. It is a chain that γⱼ→βⱼ→αᵢ, and so on, until there is no new FromPattern in the chain. We can get the chain from the PFI-Graph G. The second one is the new algorithm reduce the runtime and memory usage of constructing projected database than PrefixSpan, but adding the additional consuming of building and searching PFI-Graph.

**IV. PERFORMANCE EVALUATION**

In this chapter, we provide two experimental results. The first one is using some synthetic datasets to compare the performance of PrefixSpan with physical projection, PrefixSpan with pseudo projection and PFI-PrefixSpan. The purpose is to verify the performance of the second and third algorithms are better than the first one. But the PFI number in these synthetic datasets is small. Then we use three real data sets which have dense and highly similar sequences to compare pseudo projection PrefixSpan with PFI-PrefixSpan, whereas these data sets have big PFI number.

### 4.1 Test Environment and Data Sets

The experiments are performed on a 2.1 GHZ CUP with 2GB memory, and running on Win7. All the algorithms were coded in Java language. In our performance study, we use two kinds of data sets: synthetic data sets and real data sets. The synthetic data sets are generated by a data generator similar in spirit to the IBM data generator designed for testing sequential pattern mining algorithms. The convention for the data sets is as follows: C1N0.1T8S8 means that the data set contains 1000 sequences and the number of different items is 100. The average number of items in a transaction is 8 and the average number of transactions in sequence is 8.

Two real data sets new_york and new_orleans are from UCI[19], and the number of sequence and the number of average transactions in sequence are shown in Table 3. These files are the data underlying the Entree system.

<table>
<thead>
<tr>
<th>DATA SETS</th>
<th>#sequence</th>
<th>#transactions/sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>new_york</td>
<td>1200</td>
<td>8</td>
</tr>
<tr>
<td>new_orleans</td>
<td>327</td>
<td>11</td>
</tr>
</tbody>
</table>

### 4.2 Experimental Results

Firstly, we conduct our experiments of three algorithms: (1) PrefixSpan with physical projection (abbreviated as Phy-PrefixSpan), (2) PrefixSpan with pseudo projection (abbreviated as Pseudo-PrefixSpan) and (3) PrefixSpan with Prefix-Frequent-Items (PFI-Prefix-Span) on two synthetic datasets.

The first test is on data set C1N0.1T8S8. The average transactions number in sequence is 8, and the different items is 100. The actual transactions number in sequence is 7, so the distinct item recurrence rate or density [14] m=average sequence length/different items =8/100=0.08 (or 0.07).

The memory usages of the three algorithms are shown in the Figure 4. The support thresholds are from 0.063 to 0.075, and the CountsPFI are 2 and 3. It makes clear distinction among three algorithms, and the memory usage of Pseudo-PrefixSpan is about 38% lower than Phy-PrefixSpan. Memory usage of PFI-PrefixSpan is significantly lower than the former two algorithms when min_sup is lower than 0.07. Figure 5 shows the processing time of the three algorithms at different support thresholds on data set C1N0.1T8S8. But the runtime of Pseudo-PrefixSpan and PFI-PrefixSpan is higher than Phy-PrefixSpan. The reason is that the time consumed for navigating from the pseudo location to physical location and build PFI-Graph is more than the time saved by using PFI-Graph. Figure 6 shows the distributions of frequent sequences, the length of frequent patterns is from 1 to 6 and length 3 and 4 patterns are the highest.
The second test is on the data set C1N0.1T8S10, which contains 1000 sequences and the different items is 100 too. The average number of items in a transaction is 8 and the average number of transactions in a sequence is 10. The actual transactions number in sequence is 9, so the distinct item recurrence rate is 0.1 (or 0.09). It is denser than C1N0.1T8S8.

Figure 7 shows the runtime of the three algorithms at support thresholds from 0.1 to 0.12, and the CountsPFI is 4. On average, the runtime of Pseudo-PrefixSpan is about 16% lower than Phy-PrefixSpan, and PFI-PrefixSpan is about 20% lower than the Phy-PrefixSpan. The memory usage is shown in Figure 8, and both of PFI-PrefixSpan and Pseudo-PrefixSpan are about 12% lower than Phy-PrefixSpan. Figure 9 shows the distributions of frequent sequences. The length of frequent patterns is from 1 to 7, and length 4 pattern is the highest.

It can be seen from the experimental results that, the memory usage of Pseudo-PrefixSpan and PFI-PrefixSpan are significantly lower than Phy-PrefixSpan. But the performance of Pseudo-Prefix-Span and PFI-PrefixSpan is not very different. The reason is the small CountsPFI of synthetic data sets. We also tried other data sets, such as CXN0.03S8T20, CXN0.05S8T20, CXN0.03S8T30 and so on, but the CountsPFI is still less than 8, even when the sequence number meet the minimum support as low as 10.

<table>
<thead>
<tr>
<th>min_sup</th>
<th>#patterns</th>
<th>#CountsPFI</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.13</td>
<td>246</td>
<td>7</td>
</tr>
<tr>
<td>0.12</td>
<td>279</td>
<td>7</td>
</tr>
<tr>
<td>0.11</td>
<td>366</td>
<td>9</td>
</tr>
<tr>
<td>0.1</td>
<td>452</td>
<td>11</td>
</tr>
<tr>
<td>0.09</td>
<td>536</td>
<td>11</td>
</tr>
<tr>
<td>0.08</td>
<td>677</td>
<td>11</td>
</tr>
<tr>
<td>0.07</td>
<td>949</td>
<td>14</td>
</tr>
</tbody>
</table>

Secondly, we make experiments of Pseudo-PrefixSpan and PFI-PrefixSpan on three real sequence databases, in order to verify that PrefixSpan with PFI-Graph performs better than PrefixSpan. Both of these algorithms use pseudo projection. The character of these data sets is highly similar items, whereas the CountsPFI is big.

The first test real data set is new_orleans, the number of sequence is 327 and the average number of transactions in a sequence is 11. There is one item of a transaction. As it is shown in Table 4, the CountsPFI is from 7 to 14 and ascending when the min_sup decreasing. Therefore, more duplicated work can be reduced with small min_sup.

Figures 10 to 12 show the runtime, memory usage and frequent sequence length of PrefixSpan and PFI-PrefixSpan on data set new_orleans. Figure 10 shows the runtime of two algorithms at different support thresholds, and the runtime of PFI-PrefixSpan is 13.1% lower than PrefixSpan. We can see that the distance between the two broken lines is almost steadily, therefore the advantages of the new algorithm is stable. The memory usage of two algorithms on new_orleans is shown in Figure 11. It is clear that the memory usage of PFI-PrefixSpan is reduced by 5.5% than PrefixSpan on average. When minimum support is 0.1, PFI-PrefixSpan reduces memory consumption by 10% than PrefixSpan. Figure 12 shows the distribution of frequent sequences of data set new_orleans, from which we can see that the length-3 pattern is the largest number of frequent pattern, and the 3,
is about 27.3% (that is 3/11) of average sequence length 11.

From the Figures 10 to 12, we can assume that runtime and memory usage of PFI-PrefixSpan algorithm is lower than PrefixSpan when the minimum support is low. But if the minimum support is too low, the memory usage of PFI-PrefixSpan is bigger than PrefixSpan. For example, suppose the \( \text{min\_sup} \) is 0.02, the number of sequences is 7, and the \( \text{CountsPFI} \) grows up to 34. Runtime of PFI-PrefixSpan is about 13% lower than the PrefixSpan. But the memory usage of the new algorithm is 6% higher than the old one. The reason is that the memory consumed to store and search PFI-Graph is increasing too much.

<table>
<thead>
<tr>
<th>( \text{min_sup} )</th>
<th>#patterns</th>
<th>#CountsPFI</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.06</td>
<td>138</td>
<td>10</td>
</tr>
<tr>
<td>0.055</td>
<td>154</td>
<td>10</td>
</tr>
<tr>
<td>0.05</td>
<td>197</td>
<td>10</td>
</tr>
<tr>
<td>0.045</td>
<td>236</td>
<td>10</td>
</tr>
<tr>
<td>0.04</td>
<td>283</td>
<td>10</td>
</tr>
<tr>
<td>0.035</td>
<td>351</td>
<td>10</td>
</tr>
<tr>
<td>0.03</td>
<td>495</td>
<td>11</td>
</tr>
</tbody>
</table>

TABLE V. SOME INFORMATION OF NEW_YORK

The second test is performed on the data set new_york. Sequence number of new_york is 1200, and average number of transactions in sequence is 8. Some information is shown in Table 5, the \( \text{min\_sup} \) is from 0.03 to 0.06 and the \( \text{CountsPFI} \) values are 10 and 11. When \( \text{min\_sup} \) is 0.11, the \( \text{CountsPFI} \) is 9. Therefore, the \( \text{CountsPFI} \) is around 10 when \( \text{min\_sup} \) less than 0.1.

The runtime of PFI_PrefixSpan and PrefixSpan on data set new_york is shown in Figure 13. From this figure, we can see the average runtime of novel algorithm is about 7% lower than PrefixSpan. The distance between the two broken lines is almost steady. Figure 14 shows that when the \( \text{min\_sup} \) are 0.04 and 0.035, the memory usage of PFI-PrefixSpan are 12% and 11.2% lower than PrefixSpan. On average, the memory usage of novel algorithm is 7% lower than PrefixSpan. Figure 15 shows the distribution of frequent sequences of data set new_york. We can see that the length 2 and 3 patterns are the largest number in all frequent patterns, and the length 2 is about 25% (that is 2/8) of average sequence length 8.

From the experiment results on synthetic data sets and real datasets, we conclude that:

1. PrefixSpan with pseudo projection algorithm uses lower memory usage than PrefixSpan with physical projection, especially in large sequence databases.
2. Compared with Pseudo-PrefixSpan and Phy-PrefixSpan, novel algorithm PFI-PrefixSpan based on PFI-Graph can reduce the memory usage and the runtime.
3. PFI-PrefixSpan applies to highly similar and dense database.
4. It is difficult to build PFI-Graph in large sequence database, because the PFI is small and it only exists in very low minimum support. For example, when the \( \text{min\_sup} = 0.0005 \), the \( \text{CountsPFI} \) of data sets C10N0.1T2.5S10, C10N0.1T8S8, C10N0.1T8S10, C10N0.05T8S8, C100N0.1T2.5S10, C100N0.1T8S8 is 0. When the \( \text{min\_sup} = 0.0001 \) and the number of sequence is 10, the \( \text{CountsPFI} \) of C100N0.1T2.5S10 and C100N0.1T8S8 is only 3.
In order to improve the efficiency of sequential pattern mining on dense and highly similar sequence database, a novel algorithm PFI-PrefixSpan is provided in this paper. It is based on Prefix Frequent Items Graph (PFI-Graph) and it works quite well in the experiments. The memory usage and runtime of novel algorithm are about 25% and 20% less than PrefixSpan with physical projection (Physical- PrefixSpan) on average, respectively. It reduces about 10% runtime and 6% memory usage than PrefixSpan with pseudo projection (Pseudo-PrefixSpan) on given real datasets, respectively.

The novel algorithm is based on the Prefix Frequent Items (PFI) and PFI-Graph, and it checks weather the efforts for building a projected database is duplicated ones to others. So, the main idea of the novel algorithm is to avoid the duplicated projected database.

Some experiments in this paper verify that this algorithm is suitable for highly similar and dense sequence databases. But it is difficult to build PFI-Graph in large and sparse database.

REFERENCES