Frequent Subtree Mining Algorithm for Ribonucleic Acid Topological Pattern

Zhiqiang Li¹, Chengjie Xu², Chen Liu²*

1 School of Computer Science and Engineering, Yulin Normal University, Yulin 537000, China
2 School of Science, Hunan University of Technology, Zhuzhou 412007, China

Corresponding Author Email: liuchen6949@163.com

https://doi.org/10.18280/ria.330113

Received: 12 October 2018
Accepted: 28 January 2019

Keywords: ribonucleic acid, frequent subtree, topological pattern, frequent pattern mining

ABSTRACT

In recent years, with the demand for pattern information in the field of bioinformatics research, frequent subtree mining algorithms have become a research hotspot. In this paper, a fast and efficient mining algorithm based on frequent embedded subtree is proposed to solve the problem of large-scale of biological data and high sequence pattern similarity in the process of biological data mining. The algorithm uses a unique string coding method to represent the tree, and uses a scope-list for substring amplification and frequency testing. The pruning technique greatly compresses the search space and reduces the computational time. Compared with the classical pattern mining algorithm, the proposed algorithm improves the efficiency of mining greatly.

1. INTRODUCTION

The mining of frequent patterns in databases of transactions, sequences, trees, and graphs is a research hotspot in the field of data mining in recent years [1-2]. In essence, frequent pattern mining refers to the discovery of useful associations and correlations between patterns in a large number of databases. In recent years, with the increasing complexity of the discovered structures and the demand for pattern information in the field of bioinformatics research, frequent subtree mining algorithms have become a topic of increasing concern for researchers.

In the research of bioinformatics, the prediction of protein structure and the structural analysis and excavation of compounds are inseparable from the mining of graphs and trees. In fact, a large number of Ribonucleic Acid (RNA) structures have been collected, they are essentially tree structures. In order to obtain information on a newly generated RNA molecule, it can only be compared with known RNA structures to find whether there is a common topology pattern between them. These models have important reference value for determining the molecular function of new RNA.

The mining of frequent subtrees has been an active research area in recent years, and many corresponding algorithms have been proposed. Because of the influence of Apriori algorithm [3], current most mining subtree algorithms use a breadth-first search strategy. When performing frequent subtree mining, the common subtrees of all the trees in the dataset need to be listed. The following two methods are commonly used to enumerate all common subtrees [4]. One is the method of pattern growth, it is named as the depth-first strategy; the other is the method of layer-by-layer mining, it is named as the breadth-first strategy.

There are many frequent subtree mining algorithms. The TreeFinder algorithm [5] can be used to mine unordered embedded subtrees, but if different trees have the same tagging node or the support threshold is small, then the TreeFinder algorithm can’t fully excavate all the frequent subtrees. In addition, there are many other algorithms for mining unordered direct subtrees, such as Hybrid Tree Miner, uNot, uFreqt, Free Tree Miner, and Path Join [6-10]. However, the direct application of these algorithms to the mining of biological data has the following two limitations: Firstly, the efficiency of algorithm still needs to be improved. Currently, the most frequent subtrees pattern mining algorithms mainly adopt the Apriori algorithm and its improved algorithms. This type of algorithm uses a candidate set generation-screening method, it is an efficient algorithm for uncomplicated patterns. But the algorithm consumes a large amount of time to process a large-scale candidate set. At the same time, it requires multiple scans of the database, it will greatly affect the mining efficiency. Secondly, the data mining technology can’t effectively use the biological characteristics of DNA or protein sequence data. At present, most of the frequent pattern mining algorithms can only mine simpler frequent sequence patterns and frequent items, and they can’t be applied to more complex biological information data.

In order to deal with these problems, a frequent subtree mining algorithm based on biological data was proposed. The algorithm introduces a unique string encoding method representation tree, and uses the scope-list for substring amplification and frequent testing. The proposed algorithm greatly reduces the search space and speeds up the running time.

2. BASIC CONCEPTS AND PROBLEM DEFINITION

2.1 Frequent subtree mining

Definition 1 (rooted ordered mark tree) A rooted ordered mark tree is an acyclic connected graph, denoted as \( T(v_0, N, L, B) \), where \( N \) is a set of marked nodes in tree \( T \), \( v_0 \) is the root of the tree, \( L \) is a set of all node labels, \( B \) is a set of directed edges in tree \( T \).

Figure 1 shows examples of several marker trees. The
number of nodes in the tree $T$ is called the size of the tree, denoted as $|T|$; furthermore, for each node in the tree, a sequence number $i (i = 0 \ldots |T| - 1)$ will be allocated according to their position in the depth-first traversal. The node with the serial number $i$ in the tree is denoted as $n_i$; and the set of tokens (also called items) of the node is denoted as $L = \{0, 1, 2, 3, \ldots, m - 1\}$, where $m$ is the number of different markers in the tree. For any node $v_i \in N$, $L(v_i)$ represents the label of node $v_i$, and $n(v_i)$ represents the sequence number of node $v_i$. Each branch $b$ consists of an ordered pair of nodes, where node $v_x$ is the father of $v_y$. It should be noted that two different nodes in the tree may have the same tag. The labels of nodes $C, E,$ and $G$ in Figure 4.1 are all 2, and their serial numbers are 6, 5, and 4 respectively. Unless otherwise specified, the trees referred to herein refer to trees with rooted ordered marks.

Definition 2 (subtree) Given tree $T(v_0, N, L, B)$, if $T'(v_0', N', L', B')$ is an embedded subtree of $T$, only if (1) $N' \subseteq N$; (2) For any node $v_j \in N'$, $L(v_j) = L(v_i)$; (3) For any branch $b = (v_x, v_y) \in B'$, only if in tree $T$, $v_x$ is the father of $v_y$, marked as $T' \subseteq T$.

If it is not connected, it is called a subtree of $T$. For example, in Figure 1, $S_2$ is a subtree of $T$, and $S_2$ is a subtree of $T$.

Definition 3 (Frequent Subtree) Let $D = \{T_1, T_2, \ldots, T_n\}$ denote a database consisting of $n$ trees. $\text{minsup}$ is a given threshold, which satisfies $0 < \text{minsup} \leq 1$. If tree $s$ is a subtree belonging to $T_i$ in $D$, where $0 \leq p \leq n$, then $p$ is the support degree of tree $s$ in $D$, denoted as $\text{sup}(s) = p$. If $\text{sup}(s)/n \geq \text{minsup}$, then $s$ is a frequent subtree in $D$.

![Figure 1. Trees and their subtrees](image)

Definition 4 (Frequent Subtree Mining) Given a database $D$ composed of $n$ trees and a user-defined minimal support $\text{minsup}$, the frequent subtree mining problem is to find all frequent subtrees $s$ satisfying $\text{sup}(s)/n \geq \text{minsup}$ in $D$.

Definition 5 (node domain) If there are $g$ nodes in the tree $T$, starting from any node $v_i (0 \leq i \leq g - 1)$ in $T$, a sequence of vertices can be obtained by the root traversal of the subtree $T_i$ with the root. The last vertex in the sequence is the rightmost leaf vertex in the subtree. Let this rightmost leaf vertex be $v_i (0 \leq r \leq g - 1)$, then $s(v_i) = [n(v_i), n(v_r)]$ is the domain of the node $v_i$.

Table 1 lists the markers and node fields for the vertices of tree $T$ in Figure 1.

In Figure 1, the set of nodes of tree $T$ is $L = \{0, 1, 2, 3\}$, and the set of marked nodes is $N = \{A, B, C, D, E, F, G\}$. According to the depth-first traversal order of the tree, the sequence number of each node is shown in Table 2.

<table>
<thead>
<tr>
<th>Vertex</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number</td>
<td>0</td>
<td>1</td>
<td>6</td>
<td>2</td>
<td>5</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

From Figure 1, it can be seen that $L(A) = 0$, and the rightmost leaf node is $C$, then the root $A$ domain $s = [n(A), n(C)] = [0, 6]$.

Definition 6 (String Encoding Method) $C(T)$ represents the string encoding of the tree. The generation process is as follows: Firstly, $C(T)$ is initialized to be empty, and then its vertex sequence $(T)$ is obtained according to the depth-first traversal of $T$. During the traversal process, whenever the node $v_i$ is added to $C(T)$, the open form $L(v_i)$ of the node tag is added to $C(T)$: And when the node is rolled back to its parent, the marked closed form $L(v_i)$ is added to $C(T)$.

Since each vertex will be traversed twice, the code string that stores a tree occupies $2|T| + 1$. In addition, through this method, each tree $T$ will have a unique code string $C(T)$, and each code string $C$ also corresponds to a unique tree structure, forming a one-to-one correspondence between strings and trees, thus completely retaining the information of tree structure.

Definition 7 (substring) $X = x_1x_2 \ldots x_n$, $Y = y_1y_2 \ldots y_m$ represent two strings respectively, $n \geq m$. If for each $y_j$ in $Y$, $X$ has $x_j$ corresponding to it, such that $x_j = y_j$, and $j < l < \cdots < n$, then $X$ is a substring of $Y$.

From Definition 7, it is not difficult to find that only when $S$ is a subtree of $T$, $C(S)$ is a substring of $C(T)$. For example, in Figure 4.1, the string code of the tree $T$ is $C(T) = \{0 1 3 1 1 2 2 2 1 2 2 0\}$, and the code strings $C(S_1)$ and $C(S_2)$ of its two subtrees $S_1$ and $S_2$ are also corresponding $C(T)$ substrings. So to get all subtrees of tree $T$, all substrings of $C(T)$ can be generated and filtered.

### 2.2 Validity check of substrings

In the algorithm of this paper, all candidate subtree sets are firstly generated and their corresponding support is calculated. The candidate subtree set must be non-redundant, and each subtree is generated at most once. According to the anti-monotonicity of frequent subtrees (the frequency of a tree is always less than or equal to the frequency of its subtrees), a candidate set can be efficiently generated. Firstly, a candidate set is generated from the smallest sub-tree, and then the subtree is expanded by gradually adding neighboring nodes on the sub-tree. All infrequent subtrees are removed during the expansion process and only frequent subtrees are expanded. All the generated candidate subtrees will be represented by the string encoding method described above. In order to avoid repetitive generation of candidate subtrees, it is specified that
each subtree is only extended on the leaf node whose sequence number is the largest. Thus, the expansion of a candidate subtree \( T_i \) is essentially the extension of the rightmost end of the encoded string \( C(T_i) \) corresponding to the subtree. In the far right expansion process, a new string \( C' \) is formed by adding a character after the string \( C(T_i) \). However, the resulting new string \( C' \) does not necessarily represent a subtree, so it needs to be checked for legality.

Definition 8 A code string is legal if and only if it complies with the parentheses pairing rule, that is:

1. For each \( i \) in the string, a unique \( i \) must be paired with it.
2. If there is an \( i \) in the string before \( j \), the characters paired with them must satisfy: \( i \) is located after \( j \).

As shown in Figure 1, the string of the tree \( T \) is coded as \( C(T) = (01311223221220) \), and its two substrings are subtrees in which both \( (11221) \) and \( (011222220) \) represent \( T \), so both substrings are legal. Another example is the substring \( c \) of \( C(T) \), which is illegal.

Definition 9 A substring \( C \) is extensible if and only if \( C \) is a prefix of a valid substring \( C' \).

The tree \( T \) in Figure 2(a) is used as an example to illustrate the correctness of the above conditions. The corresponding code string of the tree is \( C(T) = (124452331) \). In the characteristic expansion process, we will encounter such a substring \( S_4 = (1244) \), which is a prefix of \( C(T) \) and therefore an extensible substring. In fact, \( S_4 \) represents the subtree of Figure 2(b), and its extension should be added to the node marked \( X \) in the tree.

![Figure 2. Tree T and its subtree extension](image)

If there is a substring \( S_2 = (12442) \), it is obviously that the subtree is the same as the substring \( S_2 \), but the position of the extension is not the same. The extension of \( S_2 \) should be added to the node labeled \( Y \) in Figure 2(c). In general, for an extensible substring, let its right-most unopened open-form character be \( \bar{i} \), then its corresponding subtree should extend the child node at the node corresponding to \( \bar{i} \).

3. FREQUENT SUBTREE MINING ALGORITHM

Based on the above analysis, a new algorithm is proposed for mining frequent subtrees for biological data. In the proposed algorithm, an initial scope-list table is created based on the input tree database, and a string encoding method corresponding to these trees is used to obtain a string set. In the process, a candidate set is generated from the smallest subtree (each vertex), and then the sub-tree is expanded on each sub-tree by gradually adding the right-most leaf node, so as to ensure that there is no redundant candidate sub-tree. During the entire subtree expansion process, each candidate tree can be generated to calculate the corresponding support degree to judge the frequency of the subtree. With the anti-monotonicity of frequent subtrees, only frequent subtree extensions are considered and all infrequent subtrees will be removed.

The extension of the algorithm to the candidate subtree \( T_i \) is achieved by performing the far right character expansion with the code string \( C(T_i) \) corresponding to the subtree. In the far right expansion of \( C(T_i) \), a new string \( C' \) can be formed simply by adding a character after the string \( C(T_i) \). Since the algorithm only extends the scalable sub-string corresponding to the sub-tree, the resulting new string \( C' \) does not necessarily represent a sub-tree, so its scalability must be verified after the new string \( C' \) is obtained. For all extensible substrings, their scope-list tables were created and then the scope-list tables were used to extend them. The process is repeated until all eligible substrings are generated, and finally the frequent subtrees that meet the requirements are obtained through frequent testing.

In the entire algorithm process, the search space is greatly compressed by the substring scalability check and the scope-list application, which improves the overall performance of the algorithm.

The overall framework of the algorithm is described as follows:

**Algorithm 1 Frequent Subtree Mining Algorithm**

**Input:** Tree database \( TDB \), Minimum support \( \text{min}_\text{sup} \);
**Output:** All embedded frequent subtrees;

**Begin**

Scan the database \( TDB \) once to generate the corresponding string encoding database \( SDB \);

Build the initial Scope-list;

For each code string \( Q = \{x_1', x_2', \ldots, x_n'\} \) in \( SDB \) do

For \( i = 1 \) To \( n \) Do

\( P_i = \{\}, i = 1, 2, \ldots, n; \)

For \( j = i \) To \( n \) Do

\( Y \) and \( x_j' \) are spelled into a string \( y' = \overline{Yx_j'}; \)

extensibility check for \( y' \); 

If \( y' \) is an extensibility substring Then

Construct a scope-list of \( y' \) using the scope-list of \( y \) and \( x_j'; \)

If \( x_j' \) is \( \bar{S} \) Then

Forming the corresponding subtree \( t' \) of \( y' \);

extensibility check for \( t' \);

If \( t' \) is frequent Then add \( y' \) to \( P_i \)

EndIf

EndIf

EndFor

EndFor

End

A Scope-list table is constructed for each newly generated
string \( y \) mentioned in the proposed algorithm. For the Scope-list table, it is defined as follows:

Definitions 10 Let \( y \) be a subtree \( T(y) \) with \( k \) nodes. The element in the Scope-list corresponding to the string is \([t, s]\).

Where \( t \) is the tree number of the parent tree where the subtree \( T(y) \) is located, and \( s \) is the range of the form \([l_1, u_1],[l_2, u_2], \ldots, [l_k, u_k]\), which indicates the domain of each node in the tree.

The data set \( D = \{T_0, T_1, T_2\} \) composed of three trees shown in Figure 3 is taken as an example to illustrate the generation process of the Scope-list table (hereinafter referred to as \( S - L \) table).

![Figure 3. Datasets made up of three trees](image)

Firstly, the algorithm builds frequent subtrees starting from the smallest subtree. The initial \( S - L \) table shown as Table 3 is established for each differently labeled vertex.

<table>
<thead>
<tr>
<th>( i )</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0([0,1])</td>
<td>1([1,2])</td>
<td>0([1,1])</td>
<td>1([2,2])</td>
<td></td>
</tr>
<tr>
<td>1([0,3])</td>
<td>2([1,1])</td>
<td>1([3,3])</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2([0,2])</td>
<td>2([2,2])</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The vertex labeled 3 is used as an example to illustrate the establishment of the initial \( S - L \) table. The vertices marked 3 appear at the node position 1 of the tree \( T_1 \), and the domain is \([1,1]\). Then, \( 0(1,1) \) is added to the \( 3 \)'s \( S - L \) table. The vertices marked with 3 also appear at the node position 3 of the tree \( T_1 \), and the domain is \([3,3]\). Then, \( 1(3,3) \) is added to the \( 3 \)'s \( S - L \) table. It also appears at node 2 of tree \( T_2 \), with the domain \([2,2]\). Then, \( 2(2,2) \) is added to the \( 3 \)'s \( S - L \) table. After searching all subtrees, all \( 3 \) items are available.

In order to determine the inclusion relationship between the domains of different nodes of the same subtree, the following definitions are given:

Definition 11 Let the domains of the vertices \( x \) and \( y \) of the same subtree be \( s_x = [l_x, u_x] \) and \( s_y = [l_y, u_y] \), respectively, only when \( l_x \leq l_y \) and \( u_y \geq u_x \).

When the new character string \( C_g \) is generated by adding the character \( g \) to a certain string \( C \) through the proposed algorithm, the corresponding \( S - L \) table is generated as follows:

1) Find the same pair of items in the \( S - L \) table for string \( C \) and character \( g \). Let a pair of items with the same \( t \) value be \((t, [l_{1t}, u_{1t}], [l_{2t}, u_{2t}], \ldots, [l_{kt}, u_{kt}])\) and \((t, [l_1, u_1])\).

2) \([l_{kt}, u_{kt}] \supseteq [l_{2t}, u_{2t}]\), then it generates item \((t, [l_{1t}, u_{1t}, l_{2t}, u_{2t}, \ldots, l_{kt}, u_{kt}, l_2, u_2])\) to join \( S - L \) table of \( C_g \).

The resulting rules are visible for each item \((t, [l_{1t}, u_{1t}, l_{2t}, u_{2t}, \ldots, l_{kt}, u_{kt}])\) in the \( S - L \) table for a string \( C \), \([l_{1t}, u_{1t}] \subseteq \{u_{1t}, u_{2t}, \ldots, l_{kt}, u_{kt}\} \). For example, in the database of Figure 3, the string \( C = [\{ \hat{1} \} \) and character \( g = \{ \hat{2} \) are set, and when generating the string \([\{ \hat{1}, \hat{2} \}] \), we firstly find out all pairs of items with the same \( t \) value in the \( S - L \) table of \([\{ \hat{1} \} \) and \([\{ \hat{2} \}] \). They are \((1(0,3)) \) and \((1(1,2)) \), \((2(0,2)) \) and \((2(2,1)) \). Since \([\{ \hat{1} \} \subseteq [0,3] \), the generated item \((1(0,3)) \) is added to the \( S - L \) table of the string \([\{ \hat{1} \}) \). By the same token, since \([\{ \hat{1} \} \subseteq [0,2] \), the generated term \((2(0,2)) \) is added to the \( S - L \) table of the string \([\{ \hat{1} \}) \).

For applying the proposed algorithm to the secondary structure pattern mining of RNA, the tree structure of the RNA molecule should first be obtained. For this purpose, a method of establishing a tree model for RNA secondary structure is given. In the tree, the sides are RNA strands of complementary base pairs, and the apexes are the 3' position and the 5' position of the double helix, the joint, the inner ring, the hairpin ring, the convex ring, and the like. From the 5' position to the 3' position of the RNA, the root is marked as 1, and the other vertices are sequentially labeled with an integer from 2 to n. The child nodes for each node will be sorted according to their tags. The tree model derived from an RNA secondary structure using this modeling approach should be unique because the 5' and 3' positions of the RNA molecule are fixed. An ordered, labeled tree model obtained using the above modeling method for an RNA secondary structure is shown in Figure 4.

![Figure 4. RNA structure and its tree model](image)

### 4. EXPERIMENTAL RESULTS AND ANALYSIS

#### 4.1 Algorithm performance comparison

In experiment, using the random program to generate the required test data, a parent tree with \( q \) markers and \( p \) nodes is generated. Then, the data set of the tree for field test is generated based on the required number of subtrees \( t \). The program's default parameters are set to \( p = 1000 \), \( q = 500 \), that is, the number of nodes in the tree is between \([0,1000]\), and the corresponding number of tags is between \([0,100]\).

The adaptability to data scale is often considered as one of the important indicators to measure data mining algorithms. Figure 5 shows the comparison of two typical sub-tree mining algorithms (PatternMatcher algorithm [11] and TreeMiner algorithm [12]), and the proposed algorithm's running time for different amounts of input data under certain support conditions. Take the test data scale from \( t = 100 \) to \( t = 10^6 \), and the support is kept at 0.5. The running time of the PM algorithm, TM algorithm, and the proposed algorithm (IRM) changes with the data volume.

It can be seen from Figure 5 that the running time of the PM algorithm, the TM algorithm, and the proposed algorithm increases as the amount of data increases. However, when the amount of data is greater than 100,000, the running time of the PM algorithm rises sharply. This is mainly due to the fact that the PM algorithm needs to repeatedly scan the database, and a
large candidate set is checked through pattern matching, which affects the efficiency of the algorithm. Therefore, the running time of the TM and the proposed algorithms mining in vertical mode is significantly less than the PM algorithm. At the same time, because of the superior pruning strategy in the proposed algorithm, it makes the time increase smaller than the TM algorithm. In summary, the proposed algorithm is significantly better than the PM and TM algorithms in a large-scale data environment.

4.2 Results of common topology pattern mining of RNA molecules

From the RNaseP [13] database, some RNA molecular structures were selected and a corresponding tree model was established on this basis. Using the algorithm IRTM, the common tree topology of 30 trees is mined.

Figure 8 shows the running time of the proposed algorithm as a function of support. It can be seen from the Figure 8 that as the degree of support decreases, the number of frequent patterns that are mined to meet the conditions is gradually increasing and the corresponding running time is also increased.
support degree is gradually increasing. However, when the support is close to 0, the significance of the frequent patterns being tapped is not significant. The growth rate of the number of models is relatively stable when the degree of support varies within the range of [0.3, 0.5]. It shows that the frequent patterns obtained by the proposed algorithm at this time are representative of the selected 30 RNA prokaryotic structures.

**Figure 9.** Number of patterns under different support degrees

Figure 10 shows a sub-tree mined at least 9 times by the proposed algorithm in 30 trees, which was not found by other algorithms. Therefore, compared with other direct subtree mining algorithms, the proposed algorithm can mine more “hidden” information for the mining of RNA sub-trees, and at the same time, it makes the mined results more biologically meaningful.

**Figure 10.** A subtree that is mined

5. CONCLUSION

In this paper, a fast and efficient frequent embedded subtree mining algorithm for biological data mining is proposed. The algorithm uses vertical mining, and uses a string encoding method to represent the tree. After the tree structure is represented by the string encoding method, the extension of the subtree is equivalent to the extension of the corresponding substring. The sub-string amplification and frequent test are performed by using the Scope-List. At the same time, the pruning technology greatly compresses the search space and speeds up the operation, which greatly improves the efficiency of the algorithm. Theoretical analysis and experiments show that compared with the classical embedded subtree mining algorithm, the proposed algorithm has higher performance and it is more efficient, faster and more stable. The efficiency of the proposed algorithm is improved by 17%.

ACKNOWLEDGMENTS

This work is supported by the National Natural Science Foundation of China (No.61841603), Guangxi Natural Science Foundation (No.2018JJA170050, No.2014GXNSFBA118268), Science and Technology Research Foundation of Guangxi Universities (201204LX339).

REFERENCES