频繁子树挖掘

PrefixTreeESpan Algorithm

2015-10-25

姜佳君
1501111332

频繁模式挖掘作业报告

姓名： 姜佳君 **学号：** 1501111332 **日期：** 2015.10.25

一、 课题介绍

频繁模式挖掘是数据挖掘中一个重要课题，它从海量数据中找到频繁出现的模式，例如频繁集，频繁序列等。然而随着数据库技术的广泛应用，出现了越来越多的非关系型的复杂数据，例如树结构和图结构的数据。它们有着广泛的应用，例如 XML 文档，化学结构式，社会网络数据等等。如何从这些海量的复杂数据结构中，找到有用的信息成为目前数据挖掘领域的一个新的课题。作为频繁模式在树和图数据库中的继承，频繁子树和子图模式挖掘成为一个新的热点研究课题。

二、 算法介绍

2.1 背景知识

基于模式增长的频繁子树挖掘算法 PrefixTreeESpan(Prefix-Tree-projected Embedded-Subtree pattern)考虑的是有序的标签树和嵌入子树，其定义如下：

定义 2.1： 有序标签树树 T 可以表示为 T = （V，v_0，E，Σ，L），其中(1)V 是节点的集合；(2) v_0是根节点；(3)E 是边的集合；(4)Σ是标签集合；(5)L 是一个从 V 到Σ的函数，它为每个节点分配标签。因为树 T 是有序的，对于树 T 的非叶子节点而言，其儿子节点之间的顺序是固定的。

有两种关于子树的定义，即嵌入子树（embedded subtree）和推导子树(induced subtree)。

定义2.2： 嵌入子树对于树T = （V，v_0，E，Σ，L）和另外一个树T' =(V', v_0', E', Σ', L')，称树T'是树T 的嵌入子树，则它必须满足如下条件：存在一个内射函数 $f:V' \to V$，其中：(1)对于T'中的任意节点v_1，它的标签与节点 $f(v_1)$标签一致，这里 $f(v_1)$是v_1在树T中的对应节点；(2)对于树T'中的任意边(v_1,v_2)，这里v_1是v_2的父亲，则在树T中节点 $f(v_1)$ 是 $f(v_2)$的祖先节点（含父亲节点）；(3)对于树T中的兄弟节点v_1和v_2，如果v_1在v_2的左边，则在树T中，$f(v_1)$ 在 $f(v_2)$

的左边。

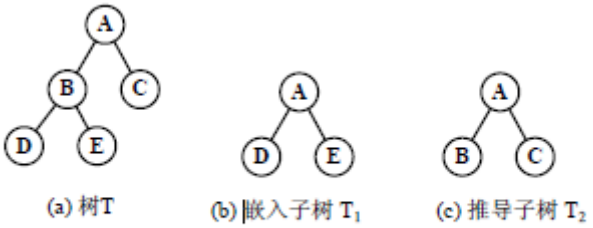


图 2.1 推导子树与嵌入子树

定义2.3： 推导子树对于树T = （V，v_0，E，Σ，L）和另外一个树T' =（V'，v_0'，E'，Σ'，L'），称树T'是树T 的嵌入子树，则满足如下条件：存在一个内射函数 $f:V' \to V$，其中：1） 对于T'中的任意节点v_1，它的标签与节点 $f(v_1)$标签一致，这里 $f(v_1)$是v_1在树T中的对应节点；2)对于树T'中的任意边(v_1,v_2)，这里v_1是v_2的父亲，则在树T中节点 $f(v_1)$也是 $f(v_2)$的父亲节点；3)对于树T中的兄弟节点v_1和v_2，如果v_1在v_2的左边，则在树T中， $f(v_1)$在 $f(v_2)$的左边。

通过定义2.2和2.3，可知推导子树是嵌入子树的一个特例。如图2.1， T_1是T的嵌入子树，但不是推导子树。因为节点A和D在T_1中是父子关系，但是在T中却是祖先和子孙的关系，不是直接的父子关系。

定义2.4： Pre-Order-String 给定一棵树T：1)如果T只有一个节点r，则T的Pre-Order-String表示为 $S(T)=l(r)-1$，其中$l(r)$表示为r的节点标签，‘-1’表示为结束符；2)如果T中的节点数目超过1，假定T的根节点为r，r有k个儿子r_i (i = 1, ..., k)，每个以r_i为根的子树表示为T_{r_i} 。则T的Pre-Order-String表示为 $S(T)=l(r)S(T_{r_1})...S(T_{r_k})-1$ 。

图2.2给出了Pre-Order-String的一个实例。

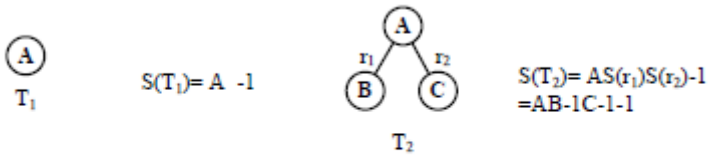


图 2.2 树的Pre-Order-String表示

定义2.5： 频繁嵌入子树挖掘（问题定义） 给定一个树数据库 $D=\{T_i \mid i=1..r\}$（其中 T_i 是一棵数据树），和最小支持计数min_sup。对于一棵模式树P，它的支持计数被表示为 $Sup(P)=|\{T_i \mid P \in T_i\}|$。频繁嵌入子树挖掘算法要求准确地找到所有的模式子树 P_j，其中 $\sup(P_j) \geq \min\_sup$。

在PrefixTreeESpan算法中，每一个频繁子树总是由其前缀树(Prefix-Tree)增长得到。前缀树定义如下:

定义2.6： 前缀树假定有m个节点的树T和有n个节点的树 $S(n \leq m)$。当从根节点开始，前序扫描树T直到它的第n个节点，则整个的前序扫描到的部分形成树M。如果树S同构于树M，则树S是树T的前缀树。

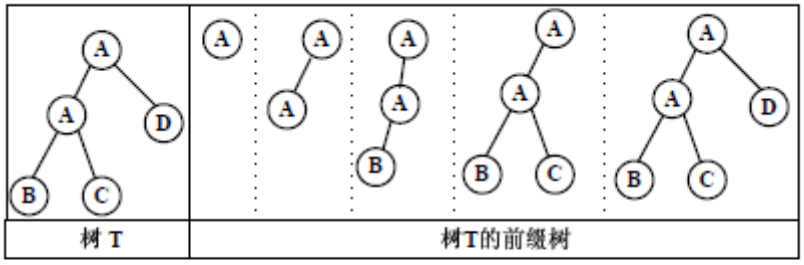


图 2.3 前缀树

图2.3中给出了前缀树的例子。在图2.3中，左边的树T有5颗前缀树，它们的节点数目分别为1，2，3，4，5。

定义2.7： 出现给定一棵数据树T和一棵模式树M，如果T中有m个子树同构于M，则这些子树被称为M在T中的出现，被表示为 $O_i(M,T), i=1,\ldots,m$。

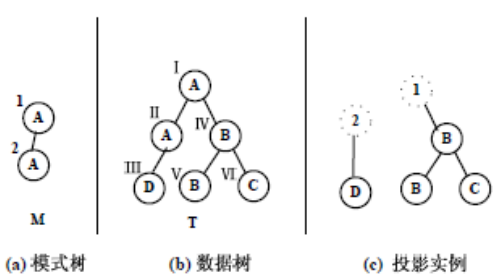


图 2.4 出现和投影实例

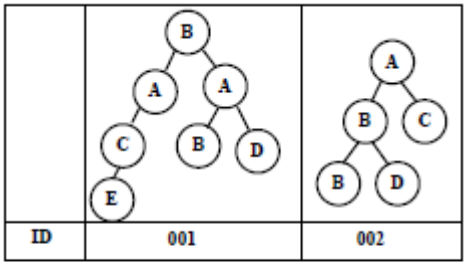


图 2.5 例2.1

定义2.8： 投影实例给定一棵数据树T和一棵模式树M，对于M在T中的一个出现$O_i(M,T)$，树T中所有在这个存在以后的出现的节点（按前序扫描的顺序）将成为一个相对于出现$O_i(M,T)$的投影实例。

图2.4显示了“出现”和“投影实例”的例子。在图2.4中，模式树M在数据树T中的出现为‘(I, II)’。其对应的投影由图2.4c显示。

2.2 PrefixTreeESpan 算法

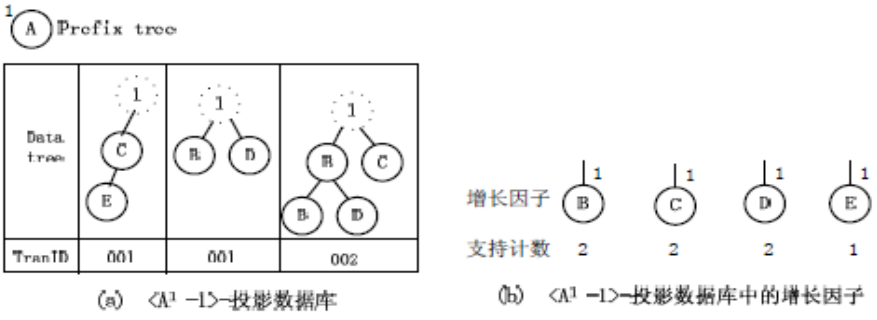
首先通过例2.1来描述PrefixTreeESpan算法的运行过程。

例2.1： 设有树数据库D，如图2.5所示，最小支持计数min_sup = 2。节点的标签集合为{A,B,C,D,E}。

采用PrefixTreeESpan算法，挖掘过程如下:

长度为1的频繁子树	A	B	C	D
计数	2	2	2	2

图 2.6 长度为1的频繁子树



(a) <A¹ -1>-投影数据库 (b) <A¹ -1>-投影数据库中的增长因子

图 2.7 $< A^1, -1 >$投影数据库和其增长因子

步骤1： 找到长度为1的所有频繁嵌入子树。扫描数据库D来找到所有的频繁节点标签。每个频繁节点标签就是一棵长度为1的频繁子树。例2.1 中可以找到< A^1,-1 >，< B^1,-1 >，< C^1,-1 >和< D^1,-1 > (这是树的Pre-Order-String表

示方法，见定义2.4。)。显然这些树需要被输出，如图2.6所示。注意到，< A^1, -1 >中A的右上角的‘1’表示节点‘A’在树的前序扫描时候的位置。

步骤2：划分搜索空间。所有的频繁子树集合可以根据前缀树进行划分。在上面的例2.1中，可以划分为：1）带有前缀< A^1,-1 >的频繁子树；2）带有前缀< B^1,-1 >的频繁子树；3）带有前缀< C^1,-1 >的频繁子树；和4）带有前缀< D^1, -1 >的频繁子树。

步骤3：在每个划分空间中找到频繁子树。根据前缀，建立不同的投影数据库。然后在每个投影数据库中，迭代地找到频繁子树。

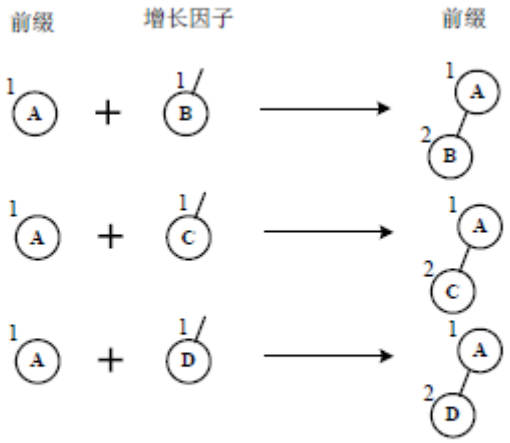


图 2.8 通过增长因子进行增长

建立投影数据库的方法. 设要建立< A^1,-1 >投影数据库，则需扫描整个数据库D找到所有节点‘A’的存在。对于某个树中的‘A’的存在，所有在这个存在以后的出现的节点（按前序扫描的顺序）将成为一个投影实例。显然一个投影实例只是对应一个存在。数据库中的所有< A^1,-1 > 的投影实例，组合成为< A^1, -1 >的投影数据库，如图2.7a所示。在图2.7a的数据库D中，有‘A’的三个存在。对于第一个存在，节点‘C’和‘E’在其后面，因而它们形成第一个投影实例。虽然其它节点，如‘A’，‘B’和‘D’，也在这个出现后面，因为它们不能直接或者间接的连接到这个出现上，所以在形成第一个投影实例时，它们不予考虑。

定义2.9： 增长因子(Growth Element) 假设有节点数目为m个的树T，和有m + 1个节点的树T'，其中T是T'的前缀树。节点v是存在于T中，而不存在于T'中的节点，被表示为(T'\T)。这个节点被称为从T到T'的增长因子。事实上，节点v是T'中第m + 1个节点（按照前序访问）。注意到，增长因子不仅仅表示节点v的标签，而且表示其连接到前缀树上的位置，所以它通常用一个二元组来表示，即(L, n)，这里L是标签，n表示这个增长因子是被连接到前缀树的第n个节点上。

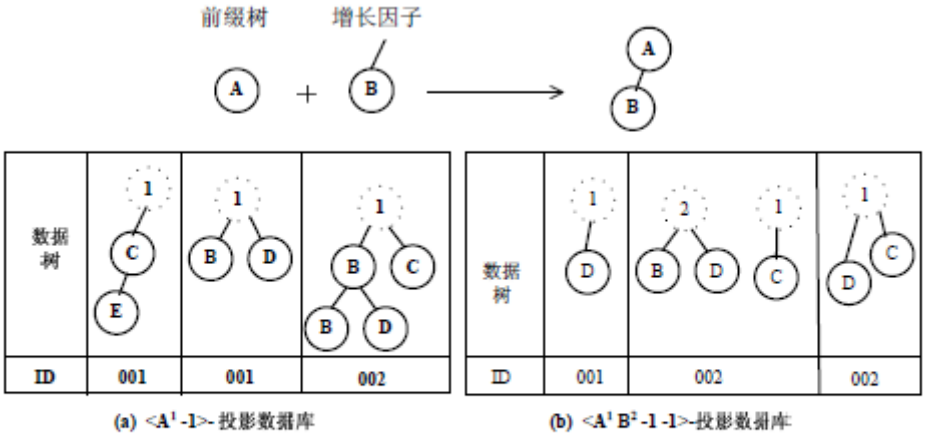


图 2.9 从$< A^1, -1 >$投影数据库形成$< A^1, B^2, -1, -1 >$投影数据库

扫描图2.7a中的< A^1,-1 >投影数据库，能找到所有的增长因子{(C, 1), (E, 1), (B, 1), (D, 1)} ，如图2.7b中所示。其中增长因子(E, 1) 不是频繁的。每个增长因子将引起一个从前缀< A^1,-1 >的增长，如图2.8所示。因此以< A^1,-1 >为前缀的频繁子树，又可以划分为3个子集：1) 带有前缀< A^1,B^2,-1,-1 >的频繁子树；2）带有前缀< A^1,C^2,-1,-1 >的频繁子树；3）带有前缀< A^1,D^2,-1,-1 >的频繁子树。对于这样的每个子集，又可以通过建立其相应的投影数据库获得。例如为了建立< A^1,B^2,-1,-1 >投影数据库，首先扫描< A^1,-1 >投影数据库，找到所有增长因子(B, 1)的存在。按照前序访问，每个存在后面节点将形成一个投影实例。显然一个存在对应一个投影实例。这些投影实例的集合将形成< A^1,B^2,-1,-1 >投影数据库，如图2.9b所示。< A^1,B^2,-1,-1 >投影数据库中的增长因子为{(D, 1), (B, 2), (D, 2), (C, 1)}。需要注意的是，增长因子(D, 2)是不同于增长因子(D, 1)的，因为增长因子(D, 2)是连接到前缀树< A^1,B^2,-1,-1 >的第2个节点上，而增长因子(D, 1)是连接到前缀树< A^1,B^2,-1,-1 >的第1个节点上，图2.10显示了两者的不同。事实上，只有增长因子(D, 1)是频繁的。迭代地，所有这些以< A^1,B^2,-1,-1 >为前缀的频繁子树，被划分为一个集合，即以< A^1,B^2,-1,D^3,-1,-1 >为前缀的频繁子树。后面的挖掘过程是类似的，这里不再详细讨论。

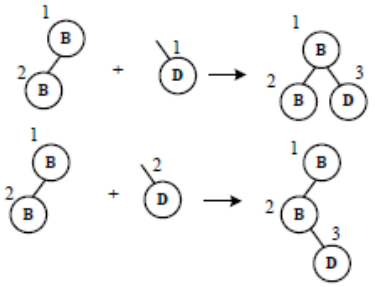


图 2.10 增长因子$(D,1)$和$(D,2)$的不同

算法2.1给出了PrefixTreeESpan算法的描述。在算法PrefixTreeESpan中，首先扫描整个数据库D来找到所有的频繁的标签(算法2.1的第1行)；对于每个频繁标签b，输出< b,-1 >（第3行）；然后找到< b,-1 >的所有出现，其对应的所有投影实例组合形成< b,-1 >投影数据库（第4行），最后迭代地调用函数Fre（第5行）。在函数Fre中，首先扫描投影数据库ProDB(D, S)得到所有的频繁增长因子（Fre函数第1行）。对于每个增长因子b，利用这个增长因子b来增长其前缀，得到一个频繁子树（第3行）；然后找到b的所有存在，其对应的投影实例组合形成投影数据库（第4行）；最后迭代调用方法Fre（第5行）。假定有n个频繁树节点。在最差情况下，算法2.1的复杂度为$O(n^n)$。然而大部分的搜索空间在算法2.1中被提前剪枝，因而算法的运行效率比较高。

Algorithm 2.1 PrefixTreeESpan 算法

Require: **输入**: 树数据库D，最小支持计数min_sup。
输出: 所有频繁子树集合RS。
1: 扫描树数据库D找到频繁的节点标签b。
2: **for** 每个频繁节点标签b **do**
3: 将频繁子树$< b-1 >$插入结果集合。
4: 在数据库D找到频繁节点标签b的所有出现，并找到所有的投影实例。所有的投影实例形成投影数据库$ProDB(D, < b, -1 >)$。
5: 调用函数$Fre(< b, -1 >, 1, ProDB(D, < b, -1 >), min_sup)$。

$Fre(S, n, ProDB(D, S), min_sup)$
1: 扫描投影数据库$ProDB(D, s)$找到所有的频繁增长因子b
2: **for** 每个增长因子b **do**
3: 通过增长因子b增长模式S来得到频繁子树模式S'。将S'插入到频繁子树集合RS。
4: 在投影数据库$ProDB(D, S)$中找到b的所有出现和投影实例。所有的投影实例的集合形成投影数据库$ProDB(D, S)$。
5: 调用函数$Fre(S', n+1, ProDB(D, S'), min_sup)$。

三、 具体实现

根据上面的 PrefixTreeESpan 算法的介绍，这里对算法的实现不再做赘余介绍，主要介绍在实现过程中的方法和思路以及在考虑的一些细节问题。

3.1 实验环境

操作系统：Windows8.1 64bit
继承开发环境：Eclipse Mars
编程语言：Java

3.2 数据结构设计

由上面的算法介绍很容易得出，我们在实现的过程中需要保存树结构的相关信息，同时在对树进行循环扩展的过程中，需要使用扩展节点信息。因此，这里我主要设计了两个类用于存储相关信息。

1. 树节点信息

```java
/**
 * Tree node structure
 * store some info of tree node
 * @author Jiajun
 *
 */
public class TreeNode {
    //node value
    private int value;
    //self position in the tree
    private int selfPos;
    //parent position in the tree
    private int parentPos;
    //the minus before the node
    private int preMinusCount;

    public TreeNode(){
    }
    /**
     *
     * @param value
     * @param selfPos
     * @param parentPos
     * @param minusNum
     */
    public TreeNode(int value, int selfPos, int parentPos, int minusNum){
    }
    /**
     * get node value
     * @return
     */
    public int getValue() {
    }
    /**
     * set node value
     * @param value
     */
    public void setValue(int value) {
    }
    /**
     * get node self position
     * @return
     */
    public int getSelfPos() {
    }
    /**
     * set node self position
     * @param selfPos
     */
    public void setSelfPos(int selfPos) {
    }
    /**
     * get parent's position
     * @return
     */
    public int getParentPos() {
    }
    /**
```

```
     * set parent's position
     * @param parentPos
     */
    public void setParentPos(int parentPos) {
    }
    /**
     * get the number of previous minus
     * @return
     */
    public int getPreMinusCount() {
    }
    /**
     * set the number of previous minus
     * @param preMinusCount
     */
    public void setPreMinusCount(int preMinusCount) {
    }
}
```

2. 树结构信息

```
/**
 * tree struct class : include tree data and some functions  used while matching
subTree
 * @author Administrator
 *
 */
public class TreeStruct {
    //store tree in the String form
    private String treeSequence = null;
    //tree date in node form, each node is an Integer form (the backward path '-1'
included)
    private ArrayList<TreeNode> treeList = new ArrayList<>();
    //store the sequence that have been matched currently
    private String matchedSequence = null;
    /**
     *
     * @param treeSequence
     */
    public TreeStruct(String treeSequence){
    }
    /**
     * get the total nodes' number
     * @return
     */
    public int getTreeLength(){
    }
    /**
     * get the node of index
     * @param index
     * @return
     */
    public TreeNode getTreeNode(int index){
    }
    /**
     * get all tree nodes
     * @return
     */
    public List<TreeNode> getTreeNodeList(){
```

```
        }

        @Override
        public String toString() {
        }
    }
```

除了以上的两种结构类，还有一种特殊的结构类，用于在进行树搜索的过程中记录搜索过程信息（包括当前的子树序列、包含有当前子树序列的所有树，以及该子树在每棵父亲树中对应的位置信息，如图 3.1 所示）。

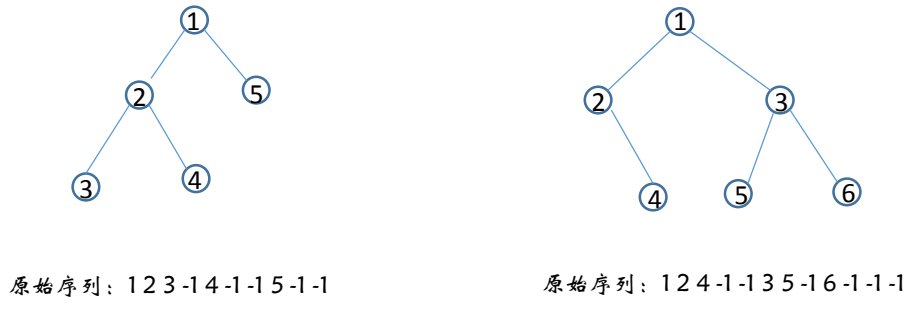


图 3.1 子树序列为：2 4 -1 -1 时所保存的产生树信息

由于结构和树的结构比较类似，这里不再进行介绍。详见源代码文件。

3.3 实现说明

接下来着重对公共子树的搜索(扩展)过程中涉及到的算法进行详细的介绍。PrefixTreeESpan 算法主要由两部分操作组成：1.寻找可扩展节点；2.对子树进行扩展。分别对应程序中的函数：

```
1.public List<GrowthElement> getGrowthElement()
2.public SubTreeStruct expandSubtree(GrowthElement ge)
```

（1）在初始状态，子树的序列为空，此时其父亲树是所有给定树的全体集合。然后在其父亲树列表中通过getGrowthELement获取所有可以扩展的节点信息，即所有的不同“label”信息，作为其起始的可扩展节点集合。然后通过函数expandSubtree函数对原始的父亲树集合分别使用生成的可扩展节点进行扩展，能够进行扩展的父亲树，将其添加到下一代子树的父亲树列表中，并标注其对应的“label”位置信息。

（2）对新生成的子树（包含了其对应的父亲树集合）应用getGrowthElement获取其父亲树列表中可以所谓当前子树的可扩展“label”信息，当前的候选“label”需要满足其在父亲树的父节点的位置信息已经包含在其对应父亲树的位置记录列表中。

（3）使用上面获取到的可扩展节点信息，对当前的子树进行扩展。如果当前子树的全部候选扩展节点都扩展失败，则将当前的子树信息输出，否则将新生成的子树信息作为下一次的扩展子树进行扩展（级重复步骤（2））。

（4）重复步骤（2）-（3），直到所有子树都不能再扩展，计算完成。

四、实验

本实验在所给数据集上进行实验，实验结果（频繁子树个数/时间消耗[单位秒]）如下：

支持度 / 数据集	5%	10%	15%	20%	30%
F5	20/1.064	9/0.939	6/0.847	2/0.707	2/0.715
D10	9/1.605	4/1.366	4/1.340	2/1.122	2/1.134
CSlog	10/139.167	4/138.552	2/138.425	1/132.845	1/133.484

五、分析总结

由上面的实验数据可以看出，在相同的实验数据上程序运行时间随着挖掘频繁模式支持度的升高，时间效率整体呈上升趋势。此外，我们很容易看出在不同的数据集上时间复杂度差异很大，这与该算法中的获取扩展子树的过程息息相关。测试数据 F5 和 D10 的数据量为 100000，而 CSlog 的数据量只有 60000.但是在运行的过程中起消耗的时间明显较大，原因在于 F5 和 D10 测试数据中的树结构比较简单，树的节点比较少，因此在匹配的时候复杂度较低。相反，CSlog 数据中的每棵树节点较多，导致在进行扩展的过程中进行节点匹配花费时间比较长，性能降低。从整体上看，实验效果还是比较理想。能够比较高效地对频繁子树进行求解。

六、说明

- 本实验只考虑了连续的频繁模式（如频繁模式 1 2 -1 5 -1 -1，图 3-1 中左边的满足，右边的不满足，即只计算直接后继）。
- 在本实现中，尽力屏蔽掉重复的频繁模式，但是依然有部分重复。（如 1 2 -1 -1 包含子树 1 -1，正常应该输出 1 2 -1 -1，忽略 1 -1，但是在实际的实现中只屏蔽掉了一部分，有待完善）