PrefixTreeESpan 算法实验报告

算法选择： PrefixTreeESpan
挖掘子图： embedded substructure
代码实现： Python

数据集：

CSlog: 59691 trees
D10: 99999 trees
F5: 100000 trees
T1M: 1000000 trees

算法描述

主要思想：

1. 频繁子树的推导子树一定是频繁的
2. 频繁子树总是可以通过其前缀树增长得到
3. 利用深度优先搜索的思想通过递归迭代不断对前缀树进行增长，统计频繁的增长因子，从而得到更多的频繁子树

伪代码：

Algorithm PrefixTreeESpan
Input:A tree database D, minimum support threshold *min_sup*
Output: All frequent subtree patterns
Methods:
1) Scan D and find all frequent label *b*.
2) **For each** frequent label *b*
3) Output pattern tree <b –1>;
4) Find all **Occurrences** of b in Database D, and construct <b –1>-projected database through collecting all corresponding *Project- Instances* in *D*;
5) **call** *Fre*(<b –1>, 1 , ProDB(D,<b -1>) , *min_sup*).

Function Fre(S, *n* , ProDB(D,S), *min_sup*)
Parameters: *S*: a subtree pattern ; *n*: the length of S; *ProDB(D,S)* : the <S>-projected database; *min_sup*: the minimum support threshold.
Methods:
1) Scan *ProDB(D,S)* once to find all frequent ***GEs*** *b*.
2) **For each** GE *b*
3) extent *S* by *b* to form a subtree pattern S' , and output S' .
4) Find all **Occurrences** of *b* in ProDB(D,S), and construct < S' > -projected database through collecting all corresponding *Project- Instances* in *ProDB(D,S)* ;
5) **call** *Fre* (S' , *n+1* , *ProDB(D,* S'),*min_sup*).

Fig .5. Algorithm PrefixTreeESpan

定义了四个 class：Node、Tree、Project、PrefixTreeESpan，下面进行详细说明：

```
class Node:
```

```
    """Node in a tree
    Attributes:
        label: 节点的标签
        range_end：标识着以该节点为根节点的子树的范围，为该节点对应的-1 节点的下标。
```

```
class Tree:
    """class of a tree

    Attributes:
        nodes: 该树对应的所有节点的集合
    """
```

```
class Project:
    """projected database 类

    Attributes:
        tree_id: 该 projected database 存储的子树源自的原树的 ID
        start_index_list: 该 projected database 存储的每个子树对应的起始下标的集合
        end_index_list: 该 projected database 存储的每个子树对应的结束下标的集合
    """
```

```
class PrefixTreeESpan:
    """
    PrefixTreeESpan 算法的实现

    Attributes:
        in_path : 输入文件路径
        out_path : 输出文件的路径
        min_propotion : 算作频繁嵌入子树结构出现的树占树的总数量的比例

        tree_list : 所有输入的树的集合
        fre_pre_tree : 频繁嵌入子树（即结果）的集合
        min_support : min support
        length_one_patterns : 存储了第一步获取的长度为 1 的频繁子树的集合

        t_start : 程序开始的时间
        t_stop : 程序结束的时间

    methods:
    def read_tree(self):
    读入文件并存储树结构
```

```
    def get_fre(self, pre_tree, n, proj_db):
    根据第 n 级的频繁子树和和 projection database 来生成第 n+1 级的频繁子树，这是一个递归的过程
    def run(self)
    首先获取长度为 1 的频繁子树的集合，然后生成对应的 projection database，调用 get_pre,来生成不同级的频繁子树。

    def output_result(self):
    输出结果到文件

     """
```

参数说明：

下面表格中的 min_propotion 表示应当算作频繁嵌入子树结构出现的树占树的总数量的比例。比如说，对于 T1M 数据集来说，如果 min_propotion = 0.1，那么 min_support = 1000000*0.1 = 100000

运行时间

如下表（单位：s）：

数据集 \ min_propotion	0.1	0.05	0.02
CSlog (59691 trees)	4.8444	5.7097	16.1403
D10 (99999 trees)	3.1476	3.9258	9.1301
F5 (100000 trees)	4.5799	6.8828	9.6244
T1M (1000000 trees)	31.8882	34.4471	60.5043

实验结果

对于不同 min_propotion 不同数据集挖掘出的频繁子树的个数如下表（单位：个数）：

数据集 \ min_propotion	0.1	0.05	0.02

CSlog (59691 trees)	2	6	73
D10 (99999 trees)	6	11	104
F5 (100000 trees)	16	33	87
T1M (1000000 trees)	6	8	38