**一种找寻最大Size pattern 的算法**

这种算法是基于FP-tree的，每次只能找到某个特定值的max size pattern, i.e. max length(row)\*length(column). 若对于max size的行列集合有不同种取法，只保存一种。

**Preliminaries：**

在矩阵中，用特定值(particular value)的指示函数提取特征。即

,

对于结果矩阵：

其有两个distinct value: 0.3 and 0.5. Their respective *indicator matrix* are displayed as below:

For particular value = 0.3:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **a** | **b** | **c** | **d** |
| **1** | 1 | 1 | 0 | 1 |
| **2** | 1 | 1 | 0 | 0 |
| **3** | 1 | 1 | 1 | 1 |
| **4** | 0 | 0 | 0 | 0 |
| **5** | 1 | 0 | 1 | 1 |
| **6** | 0 | 0 | 0 | 0 |

For particular value = 0.5:

|  |  |  |  |
| --- | --- | --- | --- |
| 0 | 0 | 1 | 0 |
| 0 | 0 | 1 | 1 |
| 0 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 |
| 0 | 1 | 0 | 0 |
| 1 | 1 | 1 | 1 |

After obtaining the *indicator matrix* for each value. The problem can be viewed as a frequent itemset searching problem. Regard each column as a commodity and each row a transaction, number 1 stands for the presence of commodity in transaction .

So the problem of searching for max size pattern of certain value in the original matrix transform to the problem, where the maximal size of certain closed itemset multiply the size of corresponding transaction are wanted.

Since we just want the maximal pattern, it is not necessary to use the closed itemset searching algorithm from which all closed itemsets are to be found. With a heuristic thought. We propose a faster and effective algorithm.

**Algorithm description:**

For each particular value. We build the FP-tree (Han et al) of all transactions. 不同于标准的FP-tree的地方在于不用记录各个值的指针，只需要按照排序过后的L1项构建单纯的树形结构即可。



|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **a** | **b** | **c** | **d** | **e** |
| **1** | 1 | 1 | 1 | 0 | 0 |
| **2** | 0 | 1 | 1 | 1 | 0 |
| **3** | 1 | 0 | 1 | 1 | 1 |
| **4** | 1 | 0 | 0 | 1 | 1 |
| **5** | 1 | 1 | 1 | 0 | 0 |
| **6** | 1 | 1 | 1 | 1 | 0 |
| **7** | 1 | 0 | 0 | 0 | 0 |
| **8** | 1 | 1 | 1 | 0 | 0 |
| **9** | 1 | 1 | 0 | 1 | 0 |
| **10** | 0 | 1 | 1 | 0 | 1 |
| **11** | 1 | 1 | 0 | 0 | 0 |
| **12** | 1 | 0 | 0 | 1 | 1 |
| **13** | 0 | 1 | 1 | 0 | 0 |
| **14** | 1 | 0 | 0 | 0 | 0 |

|  |  |
| --- | --- |
| **Item** | **Count** |
| **a** | 11 |
| **b** | 9 |
| **c** | 8 |
| **d** | 6 |
| **e** | 4 |

生成FP-tree后，最大size pattern的itemset分布一定是FP-tree的某个分支的连续的子串。且该连续子串的最下面一个节点的count\*子串节点个数是全局最大的。如图中以a开头的分支中，pattern size最大的子串为”a-b-c”，size=3\*4 = 12。

然而全局的max size不一定在该分支上。如本例中“b-c”的size为2\*7=14.此时就应当对FP-tree做一些处理，即嫁接步骤。其思想为考虑过a分支后，将节点a去除，a分支的子分支树将嫁接到null节点上。也就是去除包含a的transactions重新加入FP-tree，形成新的FP-tree树。此时null结点下b节点成为count最大的根节点。于是以b节点为根节点的分支将再次进行以上pattern searching的步骤。重复这个size计算与分支嫁接的过程直到null节点没有任何子结点。输出最大的maxsize与对应的行列集合。



由于在矩阵的pattern中，对pattern的行、列数目有所要求，我们不希望要太过“细长”或者太过“窄宽”的pattern，故对搜寻pattern的行列集合分别有一个数目下限：minrow和mincol。引入这两个量后，FP-tree上的size计算过程可以相应的简化。每次计算不用从根节点开始，而是从第level = mincol的层次节点开始。并且若该节点的count小于minrow时，就不必计算了，也不用继续向下递归（下面的节点count数单调不增），直接返回即可。

算法

1. 初始化全局变量maxsize = 0。设置minrow, mincol。初始化全局集合colset = {}，rowset = {}。设置根节点为从null结点向下的根节点中count最大的节点。
2. 对于当前节点，首先判断其count是否小于minrow，是的话直接放弃计算该节点以及其子分支；否则再看其节点level是否大于等于mincol，若小于的话跳到4步。
3. 用其节点的计数乘以层数可得. 判断 若是则有，且记录相应colset,rowset。
4. 若该节点无子结点则返回。否则对所有子结点，分别递归调用步骤3、4，即size的计算步骤，根据size与maxsize的关系，保持或更新maxsize以及对应的colset，rowset。
5. 至此得到null节点下一个分支遍历计算完成后的全局变量maxsize, colset, rowset。开始嫁接：将刚计算完的分支根节点减去，将其子分支嫁接到null节点上。取null节点下count最大的节点作为根节点。转到步骤2。

例如对于上例取mincol = 3, minrow=2，则深度遍历FP-tree中以a为根节点的分支只有两个节点进行了size计算与比较。嫁接后以b为根节点的分支中也仅有一个节点满足要求，进行了size计算与比较。最终的maxsize为12，colset为{a,b,c}, rowset为{1,5,6,8}。



实验补充：这种方法在小数据集上表现不错，但如果是600\*600都接近同一个值（树的深度太深）那么将会内存溢出！！(经过改进后不存在内存溢出问题了。)