一种高效的最大频繁 Embedded 子树挖掘算法*)

朱颖雯　吉根林
(南京师范大学计算机系　南京 210097)

摘　要　提出了一种高效的最大频繁 Embedded 子树挖掘算法——CMPETreeMiner。该算法采用先序遍历序列存储树，并将节点的范围属性加入该序列，采用伪投影技术对频繁子序列进行投影，并对投影序列中的每个节点编码。在挖掘带编码的频繁子序列过程中，对频繁子序列进行高效剪枝，得到最大频繁 Embedded 子树，无需生成所有频繁 Embedded 子树。实验结果表明，CMPETreeMiner 算法是高效可行的。

关键词　Embedded 子树，频繁子树，最大频繁子树，闭合频繁子树，数据挖掘

An Efficient Algorithm for Mining Maximal Frequent Embedded Subtrees

ZHU Ying-Wen　JI Gen-Lin
(Department of Computer, Nanjing Normal University, Nanjing 210097)

Abstract　In this paper a novel algorithm, named CMPETreeMiner, is presented to discover maximal frequent embedded subtrees from ordered labeled trees. The algorithm uses preorder sequence to store trees. The attribute of node's scope is added in the preorder sequence. Pseudo-projection technique is used during projection and each node is encoded. Mining the encoded frequent sub-sequence can be directly correspond to an embedded frequent subtree. Unlike prevoius works, several techniques are proposed to prune the frequent sub-sequence that do not correspond to closed or maximal frequent embedded subtree using code and scope information. Experiment results show that CMPETreeMiner is efficient and effective.

Keywords　Embedded subtree, Frequent subtree, Maximal frequent subtree, Closed frequent subtree, Data mining

1　引言

频繁子树挖掘是半结构化数据挖掘的一个重要研究内容，在生物信息学、Web 结构分析、模式识别、XML 数据挖掘等很多领域具有较高的应用价值。目前，国内外学者提出了若干频繁子树挖掘算法[1~10,12]，包括频繁 Induced 子树挖掘[1,3~5]、频繁 Embedded 子树挖掘[2,7~10]与最大频繁 Induced 子树挖掘[6,12]。PETreeMiner[10]是一个频繁 Embedded 子树挖掘算法，它利用序列前缀投影技术来挖掘频繁子树，在树的先序序列中加入节点的范围属性，并对频繁子序列投影，对投影序列中的节点编码，使得挖掘得到的频繁子序列直接对应成一棵频繁子树。它利用内存记录每个投影数据库节点的编码信息，导致内存消耗太大而影响算法效率。当树大小改变或支持度变小时，往往频繁子树的个数呈指数上升，有大量的频繁子树为其他频繁子树的子树，信息冗余使得用户无法找到自己需要的信息。为此，我们需要挖掘最大频繁子树。文[6]提出了 CMTreeMiner 算法挖掘最大频繁 Induced 子树，它没有采用生成所有频繁子树后再提取最大频繁子树的方法，而是在子树的枚举过程中采用剪枝技术直接得到最大频繁子树以及闭合频繁子树。但最大频繁 Embedded 子树挖掘算法尚未见文献报道，为此本文提出了一种高效的最大频繁 Embedded 子树挖掘算法——CMPETreeMiner，该算法采用先序遍历序列存储树，并将节点的范围属性加入该序列中，采用伪投影技术对频繁子序列进行投影，对投影序列中的每个节点编码。在挖掘带编码频繁子序列过程中使用剪枝技术尽早删除最终不能通过投影编码生成最大频繁 Embedded 子树的带编码频繁子序列，大大降低了搜索空间，节省了时间与空间的代价。实验表明，CMPETreeMiner 算法具有较高的效率。

2　相关概念

定义 1(Embedded 子树[2])　给定有序标号树 $T=(V,E,L,v_0)$，$T'=(V',E',L',v'_0)$。其中 V 与 V' 为节点集合，E 与 E' 为边的集合，L 与 L' 为标号集合，对 T 和 T' 中每一个节点 v 分别用 L 和 L' 中的元素标记为 $L(v)$ 与 $L'(v)$。v_0 与 v'_0 为树根。若满足以下条件，则称树 T' 是树 T 的 Embedded 子树。

① $V'\subseteq V, E'\subseteq E$。

② L' 保持 T' 中 V' 的标号。

③ 若节点 $v_1, v_2\in V, v_1, v_2\in V'$，且先序遍历 T 时 v_1 在 v_2 之前，则先序遍历 T' 时，v_1 也在 v_2 之前。

④ 若节点 $v_1, v_2\in V, v_1, v_2\in V'$，且在 T' 中 v_1 是 v_2 的双亲，则在 T 中 v_1 是 v_2 的双亲或祖先。

例如，图 1 中的四棵树是有序标号树，其中 $T2$、$T3$、$T4$ 都是 $T1$ 的 Embedded 子树。

若树 T' 是树 T 的 Embedded 子树，那么树 T 就是树 T' 的超树。本文以下讨论的子树均为 Embedded 子树。

*)江苏省自然科学基金(BK2005135)。**朱颖雯**　硕士研究生，主研方向为 XML 技术、数据挖掘；**吉根林**　博士，教授，博士生导师，主研方向为数据挖掘、机器学习、XML 技术。

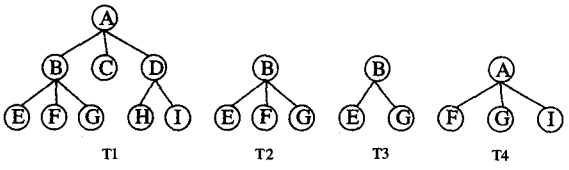


图 1　Embedded 子树示例

定义 2(频繁子树、闭合频繁子树(closed frequent subtree)、最大频繁子树(maximal frequent subtree))　给定有序标号树数据库 TDB 以及子树 T，T 的支持度定义为 $\text{Support}(T)=|P(T)/N|$，其中 $P(T)$ 是 TDB 中包含 T 的树的个数，即 T 的支持数，N 是 TDB 中树的个数。设 minsup 是用户指定的支持度阈值。当且仅当 $\text{Support}(T)\geqslant\text{minsup}$ 时 T 是频繁的。若 T 频繁且它的任何超树均不频繁则称 T 为最大频繁子树。若 T 频繁且它的任何超树的支持度均小于 T 的支持度，则称为闭合频繁子树。

闭合频繁子树的超树可能是频繁的，但最大频繁子树的超树一定是不频繁的。当闭合频繁子树 T 的支持度等于 minsup 时，T 就是一棵最大频繁子树。

性质 1　设有序标号树的数据库 TDB 中所有频繁子树的集合为 F，闭合频繁子树的集合为 C，最大频繁子树的集合为 M，则一定存在关系 $M\subseteq C\subseteq F$。从最大频繁子树集合 M 中可以得到 TDB 中所有的频繁子树。从闭合频繁子树集合 C 可以得到 TDB 中每棵频繁子树的支持度，$\text{Support}(t)=\max_{t'}\{\text{Support}(t')\}$，其中 $t'\in C$ 且 t' 是 t 的超树。

最大频繁 Embedded 子树挖掘要解决的问题就是给定有序标号树数据库 TDB 以及用户指定的最小支持度阈值 minsup，找到所有的最大频繁 Embedded 子树。

定义 3(前缀(prefix)、投影(projection)[11])　给定序列 $\alpha=e_1,e_2,\cdots,e_n$，序列 $\beta=e'_1,e'_2,\cdots,e'_m(1\leqslant m\leqslant n)$ 是 α 的前缀，当且仅当 $e'_i=e_i(1\leqslant i\leqslant m)$。给定序列 α,β，且 β 是 α 的子序列，序列 α' 是 α 在 β 下的投影，当且仅当①β 是 α' 的前缀；②不存在序列 α'' 满足 α'' 是 α' 的父亲序列，且 β 是 α'' 的前缀。令 $\alpha'=e_1,e_2,\cdots,e_n$ 是序列在前缀 $\beta=e_1,e_2,\cdots,e_m(1\leqslant m\leqslant n)$ 下的投影，那么令 $\gamma=e_{m+1},\cdots,e_n$，则称 γ 为投影 α' 关于前缀 β 的后缀。例如，序列〈ACDEFGCD〉在前缀〈ADF〉下的投影为〈ADFGCD〉，〈ADF〉是投影〈ADFGCD〉的前缀，〈GCD〉为后缀。

定义 4(节点的范围(scope)[2])　对树 T 进行先序遍历，其中节点 v 的先序序号为 q，v 的子孙中最后遍历的节点的先序序号为 r，称区间 $[q,r]$ 为节点 v 的范围。例如，图 2 中树的带有范围信息的先序序列为 $A[0,8]B[1,4]E[2,2]F[3,3]G[4,4]C[5,5]D[6,8]\ H[7,7]I[8,8]$。

性质 2　节点的范围信息反映了节点间的关系。若一个节点 $v[v_q,v_r]$ 是节点 $w[w_q,w_r]$ 的双亲或祖先，则 $v_q<w_q$ 且 $v_r\geqslant w_r$。

定义 5(树的节点编码(encode))　令树根的编码是 0，其他节点 v 的编码是：若一个节点是其双亲的第 1 个孩子，则该节点的编码是在其双亲编码后加 1；若是第 2 个孩子，则在其双亲后加 10；若是第 3 个孩子，则加 100；…以此类推。由于所有节点编码的第 1 位都是 0，因此除根以外其他节点编码的第 1 位 0 可以省略。一棵树的编码序列是在树的先序序列中加入每个节点的编码信息。例如，图 3 中树的编码序列为 A0B1E11F110G1100C10D100 H1001 I10010。

性质 3　节点的编码惟一确定了该节点在树中的位置，且只与先序序列中该节点之前的节点有关。例如，图 3 中节点 H 的编码只与节点 ABEFGCD 有关，与 I 无关。

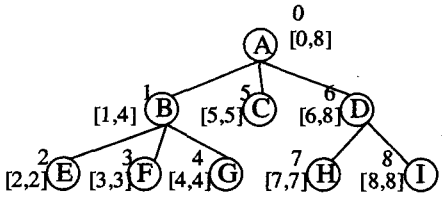


图 2　带范围信息的树

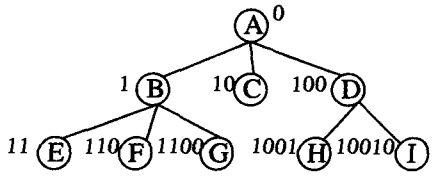


图 3　带编码信息的树

定义 6(枚举树[2])　枚举树是一个有向无环图，其中每个节点都是一棵有序标号树，并且对于每个节点，如果该节点对应的树 S 存在扩展树 T，则节点 S 存在一个指向节点 T 的边。其中，每个单节点树为空树 $\emptyset$ 的扩展树。图 5 给出了基于图 4 所示数据库的枚举树示例。

3　CMPETreeMiner 算法

3.1　算法思想

定义 7(Blanket[6])　树 t 的频繁直接超树的集合称为树 t 的 Blanket，记为 B_t。设 $t'\in B_t$，则 t' 是 t 的超树且只比 t 多一个节点，将该节点记为 $w=t'\backslash t$。根据 w 的位置，可以将 B_t 分为 B_{t_left} 与 B_{t_right}。若 w 是 t' 的最右节点(先序遍历 t' 得到的最后一个节点)，则 $t'\in B_{t\_right}$，否则 $t'\in B_{t\_left}$。如图 5 中树 $t_{35},t_{36},t_{37}\in B_{t22\_right}$，树 $t_{29},t_{32},t_{39}\in B_{t22\_left}$。

定理 1　若一棵频繁子树 t 是最大频繁子树，则 $B_t=\emptyset$；若一棵频繁子树是闭合频繁子树，则对于每一个 $t'\in B_t$，$\text{Support}(t')<\text{Support}(t)$。

定义 8(“存在-匹配”关系与“事务-匹配”关系[6])　若 $t'\in B_t$ 且对于有序标号树数据库中的每一棵包含子树 t 的树，t 的每一次存在伴随子树 t' 存在，则称子树 t' 与 t 满足“存在-匹配”(occurrence-match)关系。若 $t'\in B_t$ 且每一棵包含子树 t 的树中也包含 t'，则称子树 t' 与 t 满足“事务-匹配”(transaction-match)关系。例如，对于图 4 所示的数据库，图 5 中树 t_{23} 与 t_{37} 满足“存在-匹配”关系，t_{23} 与 t_{30} 满足“事务-匹配”但不满足“存在-匹配”关系，因为每棵包含树 t_{23} 的树中都包含 t_{30}(T1,T2,T3)，但 t_{23} 在树 T3 中存在 3 次而 t_{30} 只存在 1 次。

定理 2　对于枚举树中的一棵频繁子树 t，如果存在一棵树 $t'\in B_{t\_left}$ 且与 t 满足“存在-匹配”关系，则可以得到：(1)树 t 一定不是一棵闭合频繁子树(更不可能是最大频繁子树)；(2)对于任何一棵频繁子树 $t''\in B_{t\_right}$(即 t'' 是 t 的扩展)，一定至少存在一棵树 $t'''\in B_{t''\_left}$ 且 t''' 与 t'' 满足“存在-匹配”关系。

证明：由于 $t'\in B_{t\_left}$ 且 t' 与 t 满足“存在-匹配”关系，因此 $\text{Support}(t')=\text{Support}(t)$，从而结论(1)成立。由于 t 的每次存在总伴随 $t'\backslash t$ 节点存在且 t'' 是 t 的扩展，故 t'' 的每次存在也一定伴随 $t'\backslash t$ 节点存在且 $t'\backslash t$ 节点在 t'' 最右路径的左边，

通过在 t'' 上增加 $t'\backslash t$ 节点一定可以得到树 t''' 使得 $t''' \in B_{t''\_left}$ 且 t''' 与 t''' 满足“存在-匹配”关系。结论(2)成立。

从定理 2 得到，在枚举频繁子树的过程中，通过构造一棵频繁子树 t 的 B_{t_left} 可以对该子树进行剪枝。若 B_{t_left} 中存在频繁子树 t' 且与 t 满足“存在-匹配”关系，那么可以直接剪去子树 t(及其所有扩展树)。节省空间与时间的代价。如图 6 中树 t_2、t_3、t_4、t_5、t_6、t_{11}、t_{16}、t_{19}、t_{20}、t_{21}、t_{23}、t_{24}、t_{30}、t_{31}、t_{33}、t_{34}、t_{35}、t_{40}、t_{41} 均被剪去。

根据定义 8 可以将 B_t 分成三个不相交的子集：B_t^{OM}、B_t^{TM} 与 B_t^F，其中 B_t^{OM} 为 B_t 中与 t 满足“存在-匹配”关系的频繁子树，B_t^{TM} 为 B_t 中去除 B_t^{OM} 集合后与 t 满足“事务-匹配”关系的频繁子树，B_t^F 为剩余频繁子树集合。每个子集再根据它们在 B_{t_left} 与 B_{t_right} 上的投影又分为左右两部分(如 $B_t^{OM} = B_{t\_left}^{OM} \cup B_{t\_right}^{OM}$)。对树 t 剪枝的前提是 $B_{t\_left}^{OM} \neq \emptyset$。

CMPETreeMiner 算法将剪枝应用到了序列上，考虑在序列表示中直接对树进行剪枝。其首先在树的先序遍历序列中加入节点的范围属性，然后扫描数据库以查找所有的频繁节点 X(即长度为 1 的序列)，得到频繁 1-子树 t 及其前缀编码序列 $X0$。接着对每棵频繁子树 t 调用以下枚举过程，通过序列投影以及剪枝技术挖掘所有最大频繁子树。

(1)在包含频繁子树 t 的树中计算 $B_{t\_left}^{OM}$，若 $B_{t\_left}^{OM} \neq \emptyset$，则对树 t 剪枝后返回。

(2)若 $B_{t\_left}^{OM} = \emptyset$，则建立树 t 前缀编码序列的投影数据库，并对投影序列中的每个节点进行编码，得到带编码的投影数据库。

(3)在带编码的投影数据库中挖掘所有频繁节点 d(每个 d 由标号和编码两部分组成)，将每个频繁节点 d 与 t 的前缀编码序列合并得到新的前缀编码序列及其对应的新的频繁子树 t'，接着递归调用枚举过程进行挖掘。每棵频繁子树采用孩子-兄弟链表存储并记录各节点的编码信息。

(4)判断树 t 是否为闭合频繁子树或最大频繁子树。

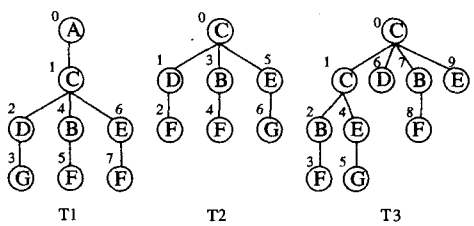
①若存在频繁节点 d，即 $B_{t\_right}^F \neq \emptyset$，则 t 一定不是最大频繁子树。若 $\exists t' \in B_{t\_right}^F \wedge \text{Support}(t') = \text{Support}(t)$，即 $B_{t\_right}^{OM} \cup B_{t\_right}^{TM} \neq \emptyset$，则 t 一定不是闭合频繁子树(更不可能是最大频繁子树)。不需要计算 $B_{t\_left}^{TM}$ 与 $B_{t\_left}^F$。

②若 $B_{t\_right}^{OM} \cup B_{t\_right}^{TM} = \emptyset \wedge B_{t\_left}^{OM} = \emptyset$ 时，计算 $B_{t\_left}^{TM}$。若 $B_{t\_left}^{TM} = \emptyset$，则 t 是闭合频繁子树，否则 t 即不是闭合频繁子树也不是最大频繁子树，不需要计算 $B_{t\_left}^F$。

③若 t 是闭合频繁子树，如果 $\text{Support}(t) = \text{minsup}$，则不需计算 $B_{t\_left}^F$，t 一定是最大频繁子树。否则，在 $B_{t\_right}^F = \emptyset$ 时，计算 $B_{t\_left}^F$，若 $B_{t\_left}^F = \emptyset$，则 t 即是闭合频繁子树又是最大频繁子树，否则 t 仅是闭合频繁子树。

3.2 投影数据库中节点编码的实现

本文借鉴文[11]中提出的伪投影技术，在投影数据库中仅存储头指针与尾指针，它们分别指向投影序列第 1 个节点与最后节点在树数据库序列中的位置。当对投影序列中的每个节点 v 编码时，考虑到 v 有可能是投影数据库中的频繁节点，当其与前缀编码序列合并生成新前缀编码序列时，一定是树 t' 的最右节点。根据性质 3，它的编码与前缀编码序列对应树 t 的编码有关，可用 t 对其编码。从树 t 的根节点开始扫描，若 v 是当前扫描节点的孩子，则编码 1，并继续扫描该节点的第 1 个孩子节点；若 v 不是当前扫描节点的子孙，则编码 0 并继续扫描该节点的右兄弟节点；直到 t 中最后一个节点。其中祖先-子孙关系的判断使用投影数据库中树的范围属性。为节省空间，使用位存储每个节点的编码。例如，图 4 中前缀编码序列 C0D1B10 对应的树 t 在 $T2$ 中出现了一次，即 $C[0,6]D[1,2]B[3,4]$，其投影序列为 $F[4,4]E[5,6]G[6,6]$。现对节点 $F[4,4]$ 编码过程如下：首先扫描 t 中根节点 C，根据性质 2，$F[4,4]$ 是节点 $C[0,6]$ 的子孙，所以编码加 1，并继续扫描 t 中 C 的第 1 个孩子节点 D，$F[4,4]$ 不是 D 编码加 0，继续扫描 t 中 D 的右兄弟[1,2]的子孙，所以节点 B，



A[0,7]C[1,7]D[2,3]G[3,3]B[4,5]F[5,5]E[6,7]F[7,7]
C[0,6]D[1,2]F[2,2]B[3,4]F[4,4]E[5,6]G[6,6]
C[0,9]C[1,5]B[2,3]F[3,3]E[4,5]G[5,5]D[6,6]B[7,8]F[8,8]E[9,9]

图 4 一个简单的有序标号树数据库

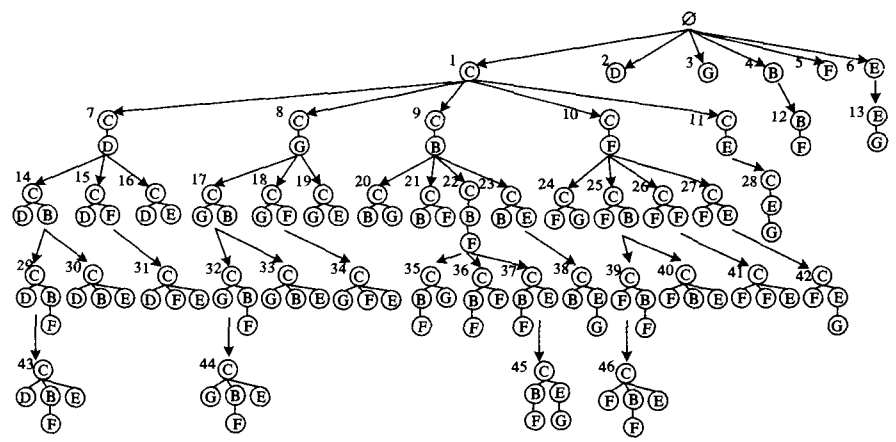


图 5 剪枝前的枚举树

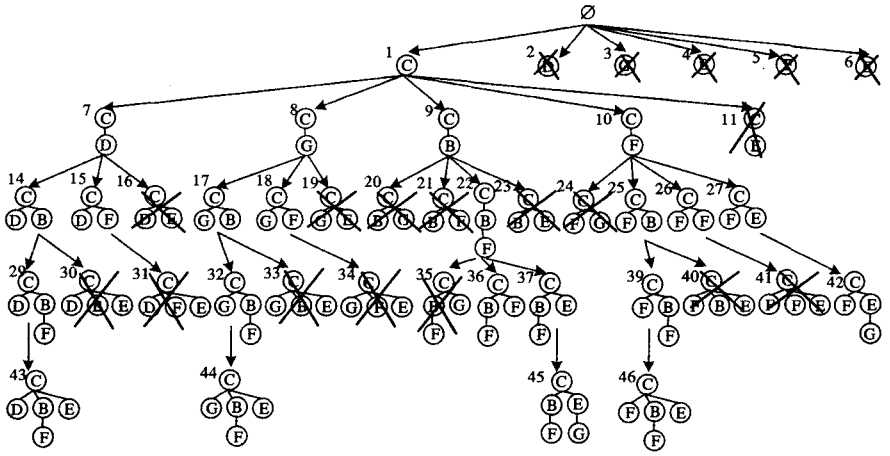


图 6　剪枝后的枚举树

$F[4,4]$是$B[3,4]$的子孙，编码加 1，扫描完毕，得到节点 $F[4,4]$的编码 101。

3.3　$B^{CM}_{t\_left}$、$B^{TM}_{t\_left}$、$B^{F}_{t_left}$的计算

假设树 t 在树数据库中对应的先序序列 $s=s_1s_2s_3\cdots s_n$，若 $B^{CM}_{t\_left}=\emptyset$，即存在树 $t'\in B^{CM}_{t\_left}$，则 t' 对应的先序序列 s'只比 s 多一个节点，记为 $s'\backslash s$。$s'\backslash s$ 节点要么在 s_1 之前；要么在 s_is_{i+1}之间($1\leqslant i\leqslant n-1$)。由于同一频繁序列可以对应许多不同的频繁树结构，故通过范围和编码信息来确定 $s'\backslash s$ 节点在序列 s'对应的树 t'中的位置。可能存在以下几种情况：

(1)若 $s'\backslash s$ 节点在 s_1 之前，如果 scope $[s'\backslash s]\geqslant$scope $[s_1]$，则 $s'\backslash s$ 节点是 s_1 节点的祖先节点。

(2)若 $s'\backslash s$ 节点在 s_is_{i+1}之间($1\leqslant i\leqslant n-1$)，如果 scope $[s'\backslash s]\geqslant$scope $[s_{i+1}]$，则 $s'\backslash s$ 节点是 s_{i+1}节点的双亲节点。否则如果 scope $[s_i]\geqslant$scope $[s'\backslash s]$，则 s_i 节点是 $s'\backslash s$ 节点的双亲节点。否则 $s'\backslash s$ 节点在 s_{i+1}节点的左边，具体又分两种情况：①若 s_{i+1}节点是 s_i 节点的双亲节点，则 $s'\backslash s$ 节点是 s_i 节点的左兄弟。②s_{i+1}节点不是 s_i 节点的双亲节点，则 $s'\backslash s$ 节点在 s_i 节点的左边，且与 s_i 节点存在一个共同的祖先，通过对 $s'\backslash s$ 进行编码可以惟一确定它在树 t'中的位置。

统计 s 在树数据库中出现的次数，对 s 在树数据库中的每次出现取具有相同位置信息的 s'的交集可以确定 $B^{CM}_{t\_left}$。例如，计算图 5 中树 t_{23}的 $B^{CM}_{t23\_left}$。t_{23}(s=CBE)在图 4 所示数据库中一共出现了 5 次，分别是 $T1$ 中出现 1 次($C[1,7]B[4,5]E[6,7]$)，$T2$ 中出现 1 次($C[0,6]B[3,4]E[5,6]$)，$T3$ 中出现 3 次($C[0,9]B[2,3]E[4,5]$、$C[0,9]B[2,3]E[9,9]$、$C[0,9]B[7,8]E[9,9]$)，首先，根据 s 的第一次出现得到一些 s'(用 $s'\backslash s$ 节点表示)：D(CB之间，C的孩子)、G(CB之间，C的孩子)、F(BE之间，B的孩子)。对于非频繁节点不考虑。然后，查看这些 $s'\backslash s$ 节点是否在 t_{23}的其他次出现中也存在。比如查看 s 在数据库的第二次出现中($T2$)CB间是否有节点 G 并且是 C 的孩子，发现不存在则删除该 $s'\backslash s$ 节点。最后只有节点 F(BE之间，B的孩子)满足条件。可以看到计算 $B^{CM}_{t_left}$本身代价并不是太高。因为对树数据库中 s 每次出现的判断都要删除一些在本次 s 出现中没有发生的 $s'\backslash s$ 节点，若所有 $s'\backslash s$ 节点都被删除，那就不再需要判断了。

$B^{TM}_{t_left}$的计算与 $B^{CM}_{t_left}$类似，t 可能在一棵树中出现多次，将同一棵树中由每一次 s 出现得到的所有 s'取并集后，再考虑取包含树 t 的树中具有相同位置信息的 s'的交集。$B^{F}_{t_left}$的计算比较复杂，需要记录并更新每个 $s'\backslash s$ 节点的支持度信息以判断该节点是否频繁。往往在判断过程中记录了许多节点，但最后只有很少的节点是频繁的。例如，计算图 5 中的树 t_{43}的 $B^{F}_{t\_left}$，它在 $T1$ 中出现 1 次可以得到 s'：G(DB之间，D的孩子)。在 $T2$ 中出现 1 次可以得到：F(DB之间，D的孩子)。在 $T3$ 中出现 1 次可以得到：C、B、F、E、G(CD之间，C的孩子)。最终没有一个 $s'\backslash s$ 节点是频繁的。而且在对每次出现判断的过程中不能删除节点，需要将所有信息留在内存中，直到计算完毕，故相当耗费内存，所以应尽量减少对 $B^{F}_{t_left}$的计算。

3.4　算法描述

CMPETreeMiner 算法描述见算法 1。原始数据库 DB 中每个元组由〈Tid，T〉组成，其中 T 是带有范围信息的先序遍历序列，Tid 是 T 在 DB 中的序号。

算法 1　CMPETreeMiner

输入：有序标号树集合 DB；支持度阈值 minsup。
输出：枚举子树集合 C；闭合频繁子树集合 CL；最大频繁子树集合 MX。
方法：

```
(1)PatternSequence=∅; /*前缀编码序列为空*/
(2)PatternTree=NULL; /*频繁子树采用孩子-兄弟链表表示*/
(3)C= CL= MX=∅;
(4)Compute support of all nodes in DB;
(5)for each 1-frequent node X {
(6)   PatternSequence=〈X, 0〉; /*前缀编码序列为 X0*/
(7)   PatternTree=Addnode(PatternTree, X, 0); /*频繁子树根节
      点为 X,其编码为 0*/
(8)   SDB=∅; /*当前前缀编码序列的投影数据库为空*/
(9)   for each 包含节点 X 的 record 〈Tid, T〉 in DB {
(10)     for T 中每一个标号为 X 且不在其他标号为 X 的节点范围
         内的节点 P {
(11)       P 范围内的节点构成序列 T'P; /*不包含 P 节点*/
(12)       SDB=SDB∪〈Tid, TP'〉; }}
(13)  CM-Explore(C, CL, MX, DB, minsup, PatternSequence,
      PatternTree); }
(14)return C,CL,MX;
```

其中，函数 CM-Explore(&C, &CL, &MX, SDB, minsup, PatternSequence, PatternTree)描述如下：

```
(1)C=CPatternSequence;
(2)E = TM =∅;
(3)compute B^CM_t_left from PatternSequence;
(4)if (B^CM_t_left≠∅) return; /*剪枝操作*/
(5)for each record 〈Tid, TP〉 in SDB {
(6)   for each node X in TP{
(7)     code=Encode(PatternTree, X ); }} /*对投影序列中每个
        节点 X 编码*/
(8)Compute support of all nodes in SDB;
(9)for each 1-frequent node d〈label, code〉{
(10)   E=E∪d;
(11)   newPatternSequence = PatternSequence +〈label,code〉; /*
       合并前缀编码序列*/
(12)   newPatternTree=Addnode(PatternTree,label,code);
(13)   newSDB=CreatePDB(SDB, d); /*根据节点 d 的位置得到新
       的投影数据库*/
(14)    if (Support(newPatternSequence) = Support(PatternSe-
        quence))
(15)     TM = TM∪newPatternSequence;
```

```
(16)  CM-Explore(C, CL, MX, SDB, minsup, newPatternSequence, newPatternTree);}
(17)  if (TM=∅) { /* B^CM_i_right ∧ B^TM_i_right = ∅ ∧ B^CM_i_left = ∅ */
(18)    compute B^TM_i_left from PatternSequence;
(19)    if (B^TM_i_left = ∅) {
(20)      CL=CL∪PatternSequence;
(21)      if (Support(PatternSequence)=minsup) MX=MX∪PatternSequence;
(22)      else if (E=∅){
(23)            compute B^F_i_left from PatternSequence;
(24)            if (B^F_i_left = ∅) MX=MX∪PatternSequence; }
(25)  } }
```

对于图 4 所示的数据库，可以得到 7 棵闭合频繁 Embedded 子树（t_{43}、t_8、t_{44}、t_{36}、t_{45}、t_{46}、t_{26}），其中有 5 棵是最大频繁 Embedded 子树（t_{43}、t_{44}、t_{36}、t_{45}、t_{46}）。

4 实验结果

为了验证 CMPETreeMiner 算法的有效性，我们利用 Visual C++ 6.0 编程实现了该算法。实验环境为 Intel Pentium IV 1.73GHz，内存 512MB，硬盘 60GB，操作系统 Windows XP。实验数据采用 Zaki 在文[2]中给出的数据生成器，该生成器模仿了网站访问树。生成器的主要控制参数包括：节点标号数 N，树的节点数目 M，树的最大深度 D，最大分支数目 F，数据库中树的总数 T。由于目前没有挖掘最大频繁 Embedded 子树的工作，但为了对 CMPETreeMiner 算法进行性能测试，我们利用 PETreeMiner 算法挖掘所有频繁 Embedded 子树然后再求其中最大频繁子树与之作比较实验。两者挖掘最大频繁子树的效率比较如图 7、8 所示。显然 CMPETreeMiner 算法的性能比 PETreeMiner 算法好得多。图 7 中，当支持度阈值在 0.1%时，由于 PETreeMiner 需要计算所有的频繁 Embedded 子树，必然导致时间的增长。此时频繁子树最大达到 17 个节点。但是 CMPETreeMiner 在枚举过程中，对最终不能生成最大频繁 Embedded 子树的频繁

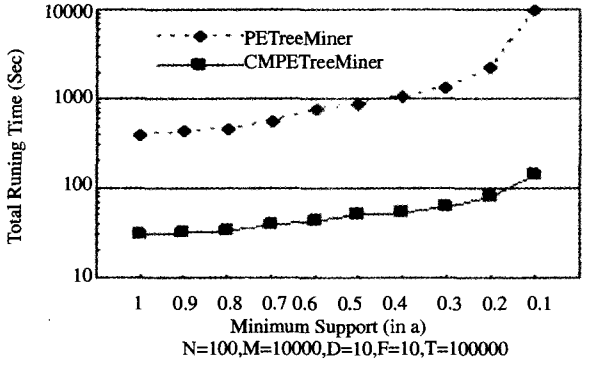


图 7 算法执行时间与支持度之间的关系

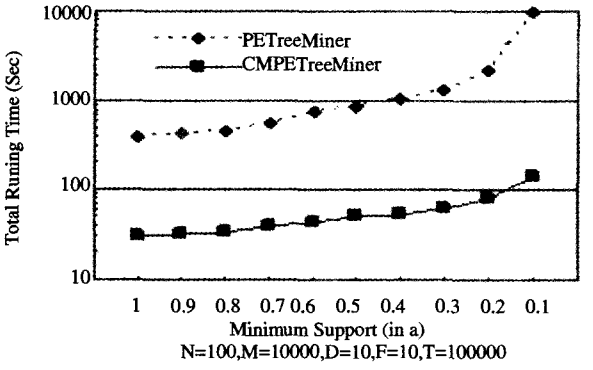


图 8 算法执行时间与数据集大小之间的关系

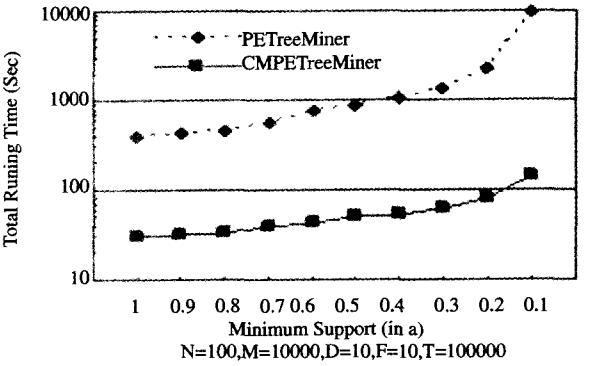


图 9 频繁模式的数量比较

子树进行了剪枝，大大减少了枚举树的个数，故效率优势更加明显。图 8 中当树的棵数由 100,000 变化到 900,000，算法运行时间和数据库中树的总数都构成一定的线性比例关系，CMPETreeMiner 相对于 PETreeMiner 优势较明显。图 9 中显示了当支持度阈值变化时，PETreeMiner 算法得到的频繁子树个数，CMPETreeMiner 枚举的频繁子树个数，闭合频繁子树个数，以及最大频繁子树个数。容易看出，CMPETreeMiner 算法由于采用了高效剪枝，枚举子树的个数已远远小于频繁子树的个数，而最大频繁子树的个数非常少。

结束语 最大频繁 Embedded 子树的挖掘，对于减少频繁子树的挖掘数量及减少挖掘冗余，有着重要的意义。本文提出的最大频繁 Embedded 子树挖掘算法将节点的范围属性加入树的先序遍历序列中，采用伪投影技术对频繁子序列进行投影，在投影过程中对序列中的每个节点编码，在挖掘带编码频繁子序列过程中使用剪枝技术，尽早删除最终不能成为最大频繁 Embedded 子树的序列，大大降低了搜索空间，节省了时间与空间的代价。我们下一步的工作是挖掘基于约束条件的频繁子树，使挖掘结果能够满足用户的兴趣。

参 考 文 献

1 Asai T, Abe K, Kawasoe S, et al. Efficient substructure discovery from large semi-structured data. In: Proceedings of the 2nd SIAM International Conf. on Data Mining, Hyatt Regency, Crystal City. Arlington, Virginia, USA, April 2002. 158～174

2 Zaki M J. Efficiently mining frequent trees in a forest. In: Proceedings of the 8th ACM SIGKDD Int'l Conf. on Knowledge Discovery and Data Mining. Edmonton, Alberta, Canada, July 2002

3 Asai T, Arimura H, Uno T, et al. Discovering frequent substructures in large unordered trees. In: Proceedings of the 6th International Conferences on Discovery Science, October 2003

4 Nijssen S, Kok J N. Efficient discovery of frequent unordered trees. In: First International Workshop on Mining Graphs, Trees and Sequences, 2003

5 Chi Y, Yang Y, Muntz R R. HybridTreeMiner: An efficient algorithm for mining frequent rooted trees and free trees using canonical forms. In: Proceedings of the 16th International Conference on Scientific and Statistical Database Management(SSDBM'04), June 2004. 11～20

6 Chi Y, Yang Y, Xia Y, et al. CMTreeMiner: Mining both closed and maximal frequent subtrees. In: Proceedings of the 8th Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD'04). Sydney, Australia, May 2004. 63～73

7 潘瑾，等. Chopper：一个高效的有序标号树频繁结构的挖掘算法. 见：第 20 届全国数据库年会(NDBC2003)，长沙，2003

8 朱永泰，等. ESPM—频繁子树挖掘算法. 计算机研究与发展，2004(10)：1720～1726

9 赵传申，孙志挥. 基于投影分支的快速频繁子树挖掘算法. 计算机研究与发展，2006，43(3)：456～462

10 陈子军，等. 基于投影编码的频繁子树挖掘算法. 见：第 23 届全国数据库年会(NDBC2006). 广州，2006

11 Jian Pei, et al. PrefixSpan: Mining sequential patterns by prefix-projected growth. In: Proc. of the 17th Int'l Conf. on Data Engineering, IEEE Computer Society, Heidelberg, Germany, 2001. 215～224

12 Xiao Y, et al. Efficient data mining for maximal frequent subtrees. In: Proceedings of the 3rd IEEE International Conf. on Data Mining (ICDM 2003), Melbourne, Florida, USA, 2003. 379～386