

NB-MAFIA: 基于 N-List 的最长频繁项集挖掘算法

沈戈晖¹ 刘沛东¹ 邓志鸿^{2,3,†}

1. 北京大学信息科学技术学院计算机科学技术系, 北京 100871; 2. 北京大学信息科学技术学院智能科学系, 北京 100871;
3. 北京大学机器感知与智能教育部重点实验室, 北京 100871; † 通信作者, E-mail: zhdeng@cis.pku.edu.cn

摘要 本文在深度优先搜索的框架上, 引入基于项集前缀树节点链表的项集表示方法 N-List, 提出一个高效的最长频繁项集挖掘算法 NB-MAFIA。N-List 的高压缩率和高效的求交集方法可以实现项集支持度的快速计算, 同时采用对搜索空间的剪枝策略和超集检测策略来提高算法效率。在多个真实和仿真数据集上, 通过实验评估了 NB-MAFIA 和两个经典算法。实验结果表明 NB-MAFIA 在多数情况下优于其他算法, 尤其在真实和稠密数据集上优势更为明显。

关键词 数据挖掘; 频繁项集挖掘; 最长项集; N-List; 算法

中图分类号 TP302

NB-MAFIA: An N-List Based Maximal Frequent Itemset Algorithm

SHEN Gehui¹, LIU Peidong¹, DENG Zhihong^{2,3,†}

1. Department of Computer Science and Technology, School of Electronics Engineering and Computer Science, Peking University, Beijing 100871; 2. Department of Machine Intelligence, School of Electronics Engineering and Computer Science, Peking University, Beijing 100871; 3. Key Laboratory Machine Perception (MOE), Peking University, Beijing 100871; † Corresponding author: E-mail: zhdeng@cis.pku.edu.cn

Abstract The authors propose an efficient algorithm, NB-MAFIA, for mining maximal frequent itemset using N-List, which uses node list of prefix tree to represent itemsets. By using N-List, itemsets' support can be efficiently computed because of the high compactness of N-List and the efficiency of the method to intersect two N-Lists. Meanwhile, the authors employ some search space pruning strategies and superset checking strategy to improve NB-MAFIA. To evaluate NB-MAFIA, the authors compare proposed algorithm with two state-of-the-art algorithms on a variety of real and synthesis datasets. Experimental results show that NB-MAFIA is efficient and outperform the baseline algorithms in most case. Especially, NB-MAFIA is more efficient on dense datasets.

Key words data mining; frequent itemset mining; maximal itemset; N-List; algorithm

近些年来, 由于应用的多样性和广泛性, 频繁模式挖掘受到研究者极大关注。自Agrawal等^[1-2]提出频繁模式挖掘以来, 它便成为数据挖掘领域中的重要问题。频繁模式挖掘不仅在关联规则挖掘中处于核心地位, 而且还在相关性分析、因果分析、序列模式挖掘、情节挖掘、多维模式挖掘等任务上有重要作用^[1]。

给定一个事务数据库DB, 假设DB包含的所有项的集合为 $I = \{i_1, i_2, \dots, i_n\}$ 。DB中有若干事务记

录, 记为 $\{T_1, T_2, \dots, T_{|DB|}\}$, 每个事务 T 都是 I 的一个子集, 即 $T \subseteq I$ 。对于任意 $X \subseteq I$, 如果事务 T 满足 $X \subseteq T$, 称 T 包含 X 。定义 $i(X) = \{Y \in D | X \subseteq Y\}$ 。 I 的所有子集构成的集合(即 I 的幂集)构成一个格, 称为项集格。我们把包含 X 的事务数称为 X 的计数支持度, 记为 $X.count$, 即 $|i(X)|$ 。包含 X 的事务数在数据库中所占比例称为 X 的支持度, 记为 $X.support$ 。给定一个最小支持度 $\minSup \in [0, 1]$, 称一个项集 X 是频繁的当且仅当 $X.support \geq \minSup$, 即

$X.count \geq \minSup \times |DB|$ 。定义所有频繁项集的集合为 FI (frequent itemsets), 长度为 k 的频繁项集称为频繁 k -项集。

当一个事务数据库非常稠密, 并且最小支持度设定非常低时, 频繁项集的数量会非常巨大。我们知道, 对于一个长度为 l 的频繁项集, 它的 $2^l - 1$ 个非空子集都是频繁的。因此频繁项集数量会随 l 呈指数型增长。在这种情况下, 挖掘出所有频繁项集是不现实的。一种可行方案是, 只挖掘所有频繁项集的一个子集, 通过这个子集可以方便地还原所有频繁项集。因此, 最长频繁项集(maximal frequent itemsets, MFI) 和闭频繁项集(closed frequent itemsets, CFI)的概念被提出。如果一个频繁项集 X 满足它的真超集都是不频繁的, 则称 X 是最长频繁项集。如果一个频繁项集 X 满足它的真超集的支持度都不等于其自身的支持度, 则称 X 是闭频繁项集。根据定义, 显然有 $MFI \subseteq CFI \subseteq FI$ 。在实际应用中, MFI 的数量要比 CFI 少几个数量级, 而 CFI 的数量又比 FI 少几个数量级。如果我们只挖掘 MFI, 再根据它还原 FI, 而每个项集的支持度也可以通过扫描数据库得到, 比起挖掘所有的频繁项集, 可以大大减少挖掘过程的时间开销和内存开销。此外, 有的应用只需要 MFI 的信息就足够^[3]。

自最长频繁模式被提出以来, 已有多种算法先后被提出并改进, 以期达到对最长频繁模式的高效挖掘。其中, Burdick 等^[4]以项的全序搜索树为搜索空间, 在深度优先搜索的基础上, 提出几个有效的剪枝策略: PEP, FHUT, MFIHUT 以及 Dynamic Reordering, 并同时使用高效的 MFI 超集检测策略 LMFI^[4-6], 得到一个整合算法: MAFIA^[4-5]。由于 MAFIA 的剪枝策略有效地减少了搜索空间, LMFI 策略也减少了检测当前项集是否是 MFI 的开销。因此, 相比已有的算法, 对于多数数据集, 特别是稠密数据集, MAFIA 在运行时间上有很大的优势。但是, MAFIA 在计算项集的支持度时, 使用位向量来表示项集, 使得每次计算支持度时都要花费一个 $|DB|$ 位的位运算。当数据库包含很多事务记录时, 位运算所需计算代价就很大。因此, 基于位向量的表示方法不适用于挖掘大规模数据集。

基于 Deng 等^[7]提出的新的项集表示结构 N-List, 本文提出一个新的最长模式挖掘算法 NB-MAFIA (N-List Based MAFIA)。我们采用基于全序搜索树和深度优先搜索的基本算法框架, 利用 N-

List 数据结构来表示项集, 并快速计算项集支持度, 同时适当地结合 MAFIA 中的剪枝策略和超集检测策略, 得到一个更有效的最长频繁项集挖掘算法。与位向量相比, 用 N-List 表示项集有较高的压缩率。当数据集比较稠密时, N-List 的长度会随着项集长度增加而显著减少, 进而对两个 N-List 的求交集运算会很快, 因此我们的算法特别适用于稠密的数据集。实验结果表明, NB-MAFIA 算法在绝大多数情况下都优于 MAFIA 算法和 FP-growth* 算法, 在稠密的数据集上, 相比已有的算法, NB-MAFIA 算法的时间效率基本上是最好的。

1 相关工作

迄今为止, 有很多研究致力于如何高效地挖掘所有频繁项集^[7-22]和最长频繁项集^[5,23-29]。Apriori 方法使用单纯的广度优先策略来遍历搜索树, 为了得到支持度信息, 它需要显式地生成候选项集并计数^[2]。MaxMiner 在使用广度优先搜索遍历的同时, 能够对可以裁剪的结点进行预测^[23]。

当频繁项集的长度较长(超过 15 或 20 个项)时, FI 和 FCI 的规模变得非常庞大, 传统方法因为要对过多的项集进行计数而变得不易实现。对于频繁 k -项集, 基于 Apriori 的算法需要挖掘出 2^k 个子集, 当项集比较长时, 算法的可扩展性就很差。另外, 由于未能及时获得较长项集的信息, 基于广度优先搜索的算法在进行有效的预测和剪枝时效果不佳。文献[23-24]分析了深度优先搜索策略的优越性。

数据库的表示形式直接影响项集生成和计数过程的效率, 从而成为影响挖掘效率的重要因素。生成项集 $Z=(X \cup Y)$ 的过程即为计算 $t(Z)=t(X) \cap t(Y)$ 的过程, 而计数则需计算 Z 的支持度。通常, 数据库由很多记录组成, 每一个记录代表一个事务。我们也可以使用垂直的结构来表示数据库和项集。对于项集 X , 用 $t(X)$ 中所有的事务编号(transaction identifiers, TIDs)来表示^[10,13-14,17]。MAFIA 正是基于这种表示方法, 对项集支持度计数过程进行优化。

Shenoy 等^[17]通过 VIPER 算法证实了基于垂直表示的方法有时可以比水平表示的最优算法表现得更加出色。但是, VIPER 算法生成了所有频繁项集, 因此不适用于模式较长的最长频繁模式。Holsheimer 等^[30]和 Savasere 等^[16]各自提出基于垂直表示的挖掘所有频繁项集的方法。Dunkel 等^[13]

分析数据库的不同表示方法对算法性能的影响。Ganti 等^[14]展示了使用垂直表示方法的优越性。

Burdick 等^[4]提出 MAFIA 算法来挖掘最长频繁模式, 使用链表结构来组织频繁项集。每个项集 I 对应一个位向量, 位向量的长度是事务数据库中的事务数。如果 I 在某个事务中出现, 则位向量中对应的位置为 1, 否则置 0。由于数据库的所有信息都保存在位向量中, 在生成候选项集并计算其支持度时只需要使用与按位运算即可。同时, MAFIA 针对搜索空间的剪枝提出了一系列有效的策略。Burdick 等^[5]对 MAFIA 进行改进和完善, 加入超集检测技术^[6], 并且针对位向量提出更有效的压缩技术。

FP-growth 方法^[8]在挖掘频繁项集时不需要生成候选项集, 而是使用一个压缩程度很高的数据结构 FP-tree 来存储数据库, 并使用分治策略进行挖掘。实验证明, 该算法显著地提高了挖掘效率。Grahne 等^[28]基于 FP-tree 数据结构, 提出挖掘最长频繁模式的 FPmax 算法。FP-growth* 算法^[29]则用一种特殊的数据结构 FP-array 来优化 FPmax 算法在稀疏数据上的效率, 同时使用 FP-tree 的一个变种 MFI-tree, 提高了检测一个频繁项集是否为最长频繁项集的效率。

Deng 等^[21]提出一种新的项集前缀树 PPC-tree。基于 PPC-tree, Deng 等^[7]进一步提出新的项集表示结构 N-List 及相应的频繁项集挖掘算法 PrePost。PPC-tree 有与 FP-tree 相似的结构和相同的压缩率。由于 PPC-tree 在前缀树的每个节点记录了前、后序遍历编号, 因此可以用对应的前缀树节点链表来表示项集, 这个链表称为 N-List^[7]。PPC-tree 记录了数据库的所有信息, 在对数据库进行两次扫描, 构建好 PPC-tree 后, 即可从内存中删除整个数据库。与 FP-growth 算法相比, PrePost 在递归寻找所有频繁项集过程中不需要反复构建局部前缀树, 也不需要反复扫描 PPC-tree, 甚至可以在构建好所有频繁项的 N-List 后将 PPC-tree 删除。计算项集支持度的过程等价于将两个 N-List 做交运算, 该过程可以由一个复杂度为 $O(m+n)$ 的算法来实现, 其中 m 和 n 分别为两个 N-List 的长度。同时, 相较于类似 Tidset 的垂直表示方式, N-List 由于有较高的压缩率, 其长度一般远小于事务数, 因此用 N-List 来表示项集, 算法效率会有很大的提高。

2 NB-MAFIA: 基于 N-List 的最长频繁模式挖掘算法

2.1 最长频繁模式挖掘的基本框架

对于一个事务数据库 DB, 假设它包含的所有项的集合用 I 表示, $I = \{i_1, i_2, \dots, i_m\}$ 。假设 I 中的元素有一个全序关系 \leq_L (比如字典序或项的支持度升序等), 根据这个顺序, 如果项 i 出现在项 j 前面, 记为 $i \leq_L j$ 。对于一个由 $I = \{a, b, c, d\}$ 组成的事务数据库, 规定全序关系 \leq_L 为字典序 (即 $a \leq_L b \leq_L c \leq_L d$)。图 1 展示这个数据库中所有项集根据这个序关系生成的一个搜索树。除树的最上层的结点对应空集外, 树中其他结点都与数据库中一个非空项集相对应, 第 k 层 (假定树的最上层为第 0 层) 包含所有 k -项集。我们利用一定的顺序来遍历这棵树, 从而找到所有最长频繁项集。为了满足 N-List 的性质, 本文规定全序关系 \leq_L 代表将项按支持度升序排列。

定义 1^[5] 对于搜索树中的结点 C , 将它对应的项集称为这个结点的 head 集合, 记做 $C.head$ 。

定义 2^[5] 对于搜索树中的结点 C , 将所有从这个结点可扩展出的项称为该结点的 tail 集合, 记做 $C.tail$ 。

显然, $C.head$ 与 $C.tail$ 之间有如下的关系: $C.tail = \{i \in I | \forall j \in C.head, j \leq_L i\}$ 。

定义 3^[5] 对于搜索树中的结点 C , 定义其 head 集合与 tail 集合的并集称为 HUT (head union tail) 集合, 记做 $C.HUT$, $C.HUT = C.head \cup C.tail$ 。

定义 4^[5] 对于搜索树中的结点 C , 将 $C.tail$ 中的每个项 i 称为 C 的一个 1-扩展项。

定义 5^[5] 对于搜索树中的结点 C , 称与 C 有相同父结点的结点为 C 的兄弟结点。所有在搜索树中位于 C 右边的兄弟结点的集合记做 $C.sibling+$ 。

在图 1 中, 对于 $\{a, c\}$ 代表的结点 P , 由上述定义可知, $P.head = \{a, c\}$, $P.tail = \{d\}$, $P.HUT = \{a, c, d\}$, $P.sibling+ = \{\{a, d\}\}$ 。其中 d 是结点 P 的 1-扩展项。

结合文献[1]提出的 Apriori 性质, 可以使用深度优先搜索 (Depth-First Search, DFS) 策略来遍历搜索树, 得到一个最基本的最长模式挖掘算法: DFS 算法, 其伪代码如算法 1 所示。在每个结点 C , 对于 $C.tail$ 中的每个 1-扩展项 i , 检测项集 $\{C.head\} \cup \{i\}$ 的支持度, 如果其支持度大于 \minSup , 则向下递归进入相应结点。当 C 的所有由 1-扩展项扩展得到

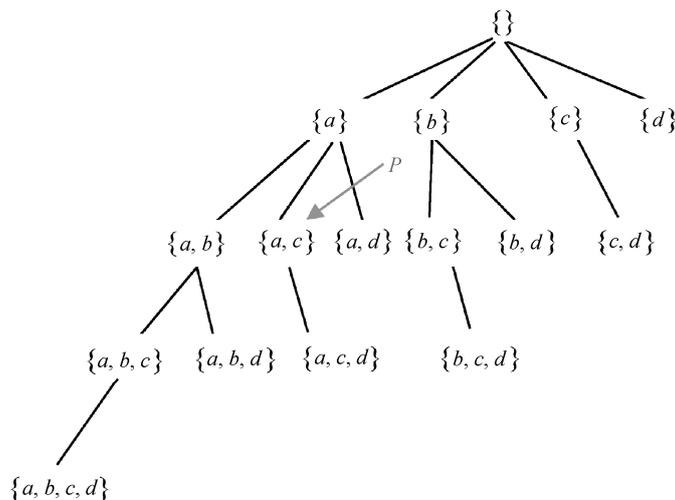


图 1 包含 4 个项的字典序搜索树
Fig. 1 Lexicographic subset tree for four items

的超集均不频繁时, C 在频繁模式搜索树中是一个叶结点。此时检测当前的最长频繁模式集合 MFI 中是否有 $C.head$ 的超集, 如果没有, 则 C 是一个最长模式, 将它加入到 MFI 中。

算法 1 (DFS 算法)

输入: 一个交易数据库 DB 和最小支持度 minSup

输出: DB 的所有最长频繁模式集合 MFI

MFI ← ∅

Root.head ← ∅

Root.tail ← DB 频繁项的集合

Call DFS(Root)

DFS(Current node C)

for $C.tail$ 中的所有项 i do

$C_extension.head$ ← $C.head \cup \{i\}$

$C_extension.tail$ ← $\{j \in C.tail \mid i \leq j\}$

计算 $C_extension.support$

If ($C_extension.support \geq minSup$)

 DFS($C_extension$)

 endif

endfor

if (C 是叶结点并且 MFI 中不存在 $C.head$ 的超集) do

MFI ← MFI \cup $C.head$

endif

2.2 PPC-tree 和 N-List 的基本性质及在 NB-MAFIA 中的应用

以下给出相关的概念和性质。关于 N-List 更

详细的介绍和以下性质的证明可以参考文献[7]。某些性质描述与原文相比略有改动, 但其本质相同, 证明过程类似。

定义 6 (PPC-tree) PPC-tree 是一棵前缀树, 它有如下两个特点: PPC-tree 有一个空的根节点, 利用输入数据生成的其他前缀树都是该根节点的子孙; PPC-tree 的每一个节点除记录项的名称与支持度外, 还记录在整棵树的前序、后续遍历中该节点的顺序编号。根节点的前序编号总是最小的, 后续编号总是最大的。

对表 1 给出的事务数据库, 当最小支持度 minSup=0.4 时, 构建的 PPC-tree 如图 2 所示。

性质 1 给出一个 PPC-tree 中两点 N_1 和 N_2 的前序编号和后序编号, 记做 $N_1.pre-order$, $N_1.post-order$, $N_2.pre-order$, $N_2.post-order$ 。 N_1 是 N_2 的祖先当且仅当 $N_1.pre-order < N_2.pre-order$ 且 $N_1.post-order > N_2.post-order$ 。

表 1 一个事务数据库
Table 1 A transaction database

ID	Items	Ordered frequent items
1	a, c, g, f	c, f, a
2	e, a, c, b	b, c, e, a
3	e, c, b, i	b, c, e
4	b, f, h	b, f
5	b, f, e, c, d	b, c, e, f

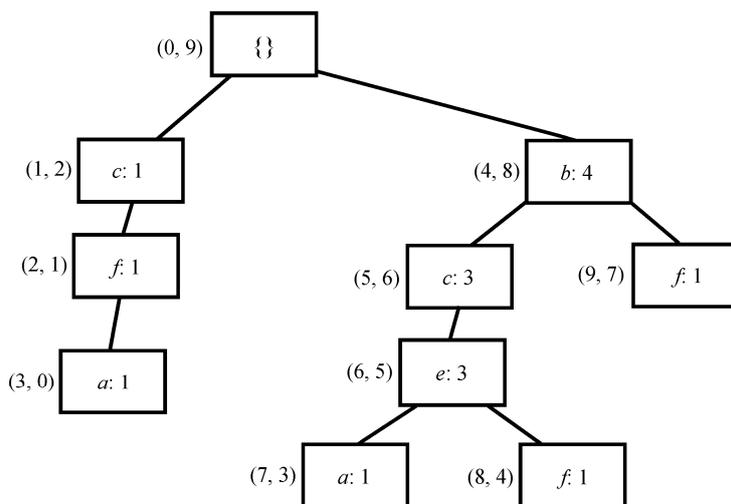


图 2 minSup=0.4 时, 表 1 所示的数据库构建的 PPC-tree
Fig. 2 PPC-tree resulting from Table 1 (minSup=0.4)

定义 7 (N-List) N-List 是一个记录节点信息的列表, 在 N-List 中, 用 $\langle(\text{pre-order}, \text{post-order}): \text{support}\rangle$ 表示 PPC-tree 的一个节点, N-List 就是将某些特定的节点按前序编号升序排列得到的列表。

定义 8 (1-项集的 N-List) 1-项集的 N-List 是将所有代表某特定项 i 的节点信息按序排列得到的列表, $\{i\}$ 的 N-List 简称为 i 的 N-List。

定义 9 (2-项集的 N-List) 假设一个 2-项集为 $\{i, j\}$, 且 i 的支持度大于 j (即 i 在 PPC-tree 中位置更高), 将 i 的 N-List 记做 $\{\langle(x_{1i}, y_{1i}):z_{1i}\rangle, \langle(x_{2i}, y_{2i}):z_{2i}\rangle, \dots\}$, j 的 N-List 记做 $\{\langle(x_{1j}, y_{1j}):z_{1j}\rangle, \langle(x_{2j}, y_{2j}):z_{2j}\rangle, \dots\}$, 则根据以下规则生成 $\{i, j\}$ 的 N-List。

1) 如果 i 的某个节点 $\langle(x_{mi}, y_{mi}):z_{mi}\rangle$ 是 j 的某个节点 $\langle(x_{nj}, y_{nj}):z_{nj}\rangle$ 的祖先, 那么将 $\langle(x_{mi}, y_{mi}):z_{mj}\rangle$ 加入 $\{i, j\}$ 的 N-List。

2) 检查 $\{i, j\}$ 的 N-List, 将所有前序编号相同的节点信息通过将支持度相加的方式合并。

定义 10 (k 项集的 N-List) k -项集的 N-List 需要通过两个 $k-1$ 项集的 N-List 合并得到。设 k 项集为 $i_1 i_2 \dots i_k i_{k+1} i_{k+2} (i_1 \leq_L i_2 \leq_L \dots \leq_L i_k \leq_L i_{k+1} \leq_L i_{k+2})$ 。两个 $k-1$ 项集分别为 $I_1 = i_1 i_2 \dots i_k i_{k+2}$, $I_2 = i_1 i_2 \dots i_k i_{k+1}$ 。将 I_1 的 N-List 记做 $\{\langle(x_{1i}, y_{1i}):z_{1i}\rangle, \langle(x_{2i}, y_{2i}):z_{2i}\rangle, \dots\}$, I_2 的 N-List 记做 $\{\langle(x_{1j}, y_{1j}):z_{1j}\rangle, \langle(x_{2j}, y_{2j}):z_{2j}\rangle, \dots\}$, 根据定义 9 中规则 2, 类似地生成该 k 项集的 N-List。

为了更好理解上述概念, 以图 2 为例, 展示一些项集的 N-List。根据定义 8, $\{c\}$ 的 N-List 为 $\langle(1,$

$2):1\rangle, \langle(5, 6):3\rangle$, $\{e\}$ 的 N-List 为 $\langle(6, 5):3\rangle$, $\{f\}$ 的 N-List 为 $\langle(2, 1):1\rangle, \langle(8, 4):1\rangle, \langle(9, 7):1\rangle$ 。根据定义 9, $\{cf\}$ 的 N-List 为 $\langle(1, 2):1\rangle, \langle(5, 6):1\rangle$, $\{ef\}$ 的 N-List 为 $\langle(6, 5):1\rangle$ 。类似地, $\{cef\}$ 的 N-List 为 $\langle(5, 6):1\rangle$ 。

性质 2 两个长度分别为 m 和 n 的 N-List 求交集的时间复杂度是 $O(m+n)$ 。

性质 3 某 k 项集的支持度可以通过将该 k 项集的 N-List 的所有节点的支持度相加得到。

至此, 我们得到快速计算项集支持度的方法。当数据集较稠密时, PPC-tree 对数据集的压缩率较高, 从而项集的 N-List 长度远小于事务的数量。同时, N-List 的长度和合并时间会随着项集长度的增加而减少, 因此用 N-List 表示项集的方法在计算项集支持度时十分高效。这也是 N-List 相较于 Tidset 的优势所在。

2.3 NB-MAFIA 算法

2.3.1 基本算法

对于 DFS 算法, 其主要运行时间都用在计算 1-扩展项集的支持度上。根据前面介绍的 N-List 的性质, 对于两个项集 $i_1 i_2 \dots i_n i_u$ 和 $i_1 i_2 \dots i_n i_v$, 其中 $i_1 \leq_L i_2 \leq_L \dots \leq_L i_n \leq_L i_u \leq_L i_v$, 假设它们的 N-List 分别为 NL_1 和 NL_2 , 那么我们可对 NL_1 和 NL_2 做求交集操作, 得到 $i_1 i_2 \dots i_n i_u i_v$ 的 N-List, 进而立即得到 $i_1 i_2 \dots i_n i_u i_v$ 的支持度。因此, 对于当前搜索结点 C , $C.\text{head} = \{i_1 i_2 \dots i_n\}$, $C.\text{tail} = \{i_k i_{k+1} \dots i_m\} (i_1 \leq_L i_2 \leq_L \dots \leq_L i_n \leq_L i_k \leq_L i_{k+1} \leq_L \dots \leq_L i_m)$, 为了得到 C 的所有 1-扩展项集

的支持度,我们只需要记录项集 $i_1i_2\dots i_{n-1}i_n$ 和 $i_1i_2\dots i_{n-1}i_k, i_1i_2\dots i_{n-1}i_{k+1}, \dots, i_1i_2\dots i_{n-1}i_m$ 的 N-List, $i_1i_2\dots i_n$ 即为结点 C 所对应的项集,后面项集组成的集合则为 $C.sibling^+$ 。基于这种思想,我们可以将 N-List 与深度优先搜索策略相结合,得到基于 N-List 的 DFS 算法,其伪代码如算法 2 所示。

算法 2 (基于 N-List 的 DFS 算法)

输入: 一个交易数据库 DB 和最小支持度 minSup

输出: DB 的所有最长频繁模式集合 MFI

MFI $\leftarrow\emptyset$

Root.head $\leftarrow\emptyset$

Root.tail \leftarrow DB 频繁项的集合(按照项的支持度升序排列)

运行 PPC-tree construction 算法,得到所有频繁 1-项集的 N-List,记为 NL1

Call DFS_based_NList(Root, NULL, NULL)

DFS_based_NList(Current node C , C 's NList NL_cur, $C.sibling^+$'s NLists[] NLS)

$k\leftarrow C.head$ 中按项的支持度升序排列排在最后的项

for $C.tail$ 中的所有项 i do

if (NL_cur==NULL)

NL_child[i] = NL1[i]

else

$C_tmp\leftarrow C.head\cup\{i\}-\{k\}$

NL_tmp \leftarrow NLS 中记录 C_tmp 的元素

NL_child[i]=NL_intersection(NL_cur,NL_tmp)

endif

endfor

for $C.tail$ 中的所有项 i do

$C_extension.head\leftarrow C.head\cup\{i\}$

$C_extension.tail\leftarrow\{j\in C.tail\mid i\leq j\}$

$C_tmp\leftarrow C.head\cup\{i\}-\{k\}$

NLS_child \leftarrow NL_child 中所有在 NL_child[i]后面的元素

If (NL_child[i].support \geq minSup)

DFS_based_NList($C_extension$, NL_child[i], NLS_child)

endif

endfor

if (C 是叶结点并且 MFI 中不存在 $C.head$ 的超集) do

MFI \leftarrow MFI $\cup C.head$

endif

2.3.2 优化策略

MAFIA 使用的第一个重要剪枝策略是父结点等价剪枝(parent equivalence pruning, PEP)^[5]。PEP 剪枝主要基于项集搜索树的以下性质。

性质 4 对于搜索树中的一个结点 C , y 为

$C.tail$ 中的一个元素,如果包含 $X=C.head$ 的事务和包含 $Y=C.head\cup\{y\}$ 的事务完全相同,即 $t(X)=t(Y)$,那么对于 $C.tail$ 的任意子集 S ,项集 $I_1=X\cup S$ 和 $I_2=Y\cup S$ 的支持度相同。

证明 显然 $t(I_2)\subseteq t(I_1)$ 。如果存在一个事务 T 包含 I_1 但不包含 I_2 ,因为 I_2 只比 I_1 多一个项 y ,因此 $y\notin T$ 。但由于 $t(X)=t(Y)$,且 T 包含 X ,故 T 包含 Y ,因此 $y\in T$,这就发生矛盾。因此有 $t(I_2)=t(I_1)$,即 I_1 与 I_2 的支持度相同,进而 $I_1=X\cup S$ 一定不是最长频繁项集。

因此,在对以 C 为根结点的子树进行搜索时,可以直接将 y 从 $C.tail$ 中删除,并放入 $C.head$,不会影响搜索出的最长模式。

我们用到的另一个剪枝策略是 HUTMFI 剪枝^[3]。假设当前搜索结点为 C ,如果当前的 MFI 中已经存在 $C.HUT$ 的超集,那么以 C 为根结点的子树中必然没有最长频项集,因此可以立即结束进一步的递归搜索。

另外,将任何一个项集加入 MFI 时,必须检测它的超集是否已经在 MFI 中存在,这个检测操作比较频繁。我们使用 MAFIA 所使用的策略^[5]来优化超集检测过程,有效地改进算法效率。对于一个给定的结点 C ,MFI 中只有一部分是 $C.head$ 的超集,称之为 C 的局部最长频繁项集(local MFI, LMFI)。对于任意的 $y\in C.tail$,我们会从 y 得到 C 的 1-扩展项集并向下进行递归到结点 C_1 。可以看到,当前 C_1 的 LMFI 显然是 C 的 LMFI 的子集,并且 C_1 的 LMFI 中的项集都要包含 y 。因此,可以对 C 的 LMFI 进行一个排序,将其中包含 y 的项集放在一起,这些项集构成 C_1 的 LMFI。当以 C 为根结点的子树搜索完成时,由于在这个子树上找到的最长频繁项集都是 C 的超集,因此要把新加入 MFI 的项集加入到 C 的 LMFI 中。在判断是否要将一个项集加入 MFI 时,只需要判断当前结点的 LMFI 是否为空即可。

算法 3 展示采用上述优化方法的伪代码(为了清晰起见,本节算法中没有加入与 N-List 相关的部分)。

算法 3 (基于优化策略的算法)

输入: 一个交易数据库 DB 和最小支持度 minSup

输出: DB 的所有最长频繁模式集合 MFI

MFI $\leftarrow\emptyset$

```

Root.head←∅
Root.tail←DB 频繁项的集合
Call DFS(Root)

DFS(Current node C)
//HUTMFI
If (MFI 中存在 C.HUT 的超集)
    return
endif
for C.tail 中的所有项 i do
    C_extension.head←C.head ∪ {i}
    C_extension.tail← {j ∈ C.tail | i ≤L j}
    计算 C_extension.support
    //PEP
    if(C_extension.support == C.support)
        将 i 从 C.tail 删掉并加入 C.head
    else if (C_extension.support ≥ minSup)
        对 MFI 中在 C.LMFIleft 和 C.LMFIright 之间的
        项集进行排序, 包含 i 的放在右边, 不包含 i 的放
        在左边。假设最靠左边包含 i 的项集标号为 l。
        C_extension.LMFIleft←l
        C_extension.LMFIright←C.LMFIright
        DFS(C_extension)
        调整 C.LMFIright 使得所有新加入 MFI 的项
        集都在 C 的 LMFI 中
    endif
endifor
if (C 的 LMFI 为空集) do
    MFI←MFI ∪ C.head
endif

```

2.3.3 NB-MAFIA 算法伪码

最后, 我们将算法 2 与算法 3 相结合, 形成 NB-MAFIA 算法。算法 4 给出 NB-MAFIA 算法的伪代码。需要注意的是, 算法 2 在第一个 for 循环中计算所有 1-扩展项集的支持度, 我们在这个循环中同时执行 PEP 剪枝的步骤。

算法 4 (NB-MAFIA 算法)

输入: 一个交易数据库 DB 和最小支持度 minSup

输出: DB 的所有最长频繁模式集合 MFI

```

MFI←∅
Root.head←∅
Root.tail←DB 频繁项的集合(按照项的支持度升序排
列)
运行 PPC-tree construction 算法, 得到所有频繁 1-项
集的 N-List, 记为 NL1

```

```

Call NBMAFIA (Root, NULL, NULL)
NBMAFIA (Current node C, C'sNList NL_cur,
C.sibling+s'NLists[] NLS)
//HUTMFI
If (MFI 中存在 C.HUT 的超集)
    return
endif
k←C.head 中按项的支持度升序排列排在最后的项
for C.tail 中的所有项 i do
    if (NL_cur == NULL)
        NL_child[i] = NL1[i]
    else
        C_tmp←C.head ∪ {i} - {k}
        NL_tmp←NLS 中记录 C_tmp 的元素
        NL_child[i]=NL_intersection(NL_cur, NL_tmp)
    endif
    //PEP
    If (NL_child[i].support == C.support)
        将 i 从 C.tail 删掉并加入 C.head
    else if (NL_child[i].support < minSup)
        将 i 从 C.tail 删掉
    endifor
for C.tail 中的所有项 i do
    C_extension.head←C.head ∪ {i}
    C_extension.tail← {j ∈ C.tail | i ≤L j}
    C_tmp←C.head ∪ {i} - {k}
    NLS_child←NL_child 中所有在 NL_child[i]后
    面的元素
    对 MFI 中在 C.LMFIleft 和 C.LMFIright 之间的
    项集进行排序, 包含 i 的放在右边, 不包含 i 的放在
    左边。假设最靠左边包含 i 的项集标号为 l
    C_extension.LMFIleft←l
    C_extension.LMFIright←C.LMFIright
    NBMAFIA(C_extension, NL_child[i], NLS_child)
    调整 C.LMFIright 使得所有新加入 MFI 的项
    集都在 C 的 LMFI 中
    endif
endifor
if (C 的 LMFI 为空集) do
    MFI←MFI ∪ C.head
endif

```

3 实验

通过比较 NB-MAFIA 与其他两个著名算法 MAFIA^[5]和 FP-growth^{*[29]}在不同数据集上挖掘最长频繁模式所用时间, 来验证其有效性。在每组实验中, 3 个算法运行所得到的最长频繁模式完全相同, 保证了 NB-MAFIA 算法的正确性和完整性。我们

根据 2003 年频繁项集挖掘评测比赛官方网站(<http://fimi.ua.ac.be/experiments/fimi03/maximal>)上提供的信息, 在每个数据集上选取合适的最小支持度用于测试。同时, 为了保证试验结果的公平, 我们在同一台机器上运行 3 个算法, 并比较结果, 避免由其他客观因素带来的性能差异。

3.1 实验设置

我们使用 6 个真实数据集和两个仿真数据集来完成实验, 这些数据集都是在已有的关于频繁模式挖掘的论文中经常提及和使用的。6 个真实数据集为 Mushroom, Accidents, Pumsb, Retail, Kosarak 和 Connect。两个仿真数据集分别是 T10I4D100K 和 T40I10D100K。这些数据集都由 2003 年频繁模式挖掘评测比赛官方网站(<http://fimi.ua.ac.be/data/>)提供。T10I4D100K 和 T40I10D100K 由数据生成器 IBM Almaden 生成。生成 T10I4D100K 的标准为: 平均事务长度(average transactions length, ATL)为 10, 潜在的最长频繁项集的平均长度为 4, 事务数和项的数量分别为 100000 和 1000。T40I10D100K 的标准与之类似。

考虑到不同算法在不同特征的数据集上的表现差异较大, 我们同时选择稠密的数据集和稀疏的数据集进行实验。在真实数据集中, Accidents, Pumsb 和 Connect 都是十分稠密的数据集, 即使在较高的最小支持度下, 它们也包含大量的频繁项集。Mushroom 虽然也比较稠密, 但是其整体规模较小(只包含一百多个项和几千个事务)。Kosarak 的大小仅次于 Accidents, 但比 Accidents 要稀疏得多。Retail 是十分稀疏的数据集。两个仿真数据集 T10I4D100K 和 T40I10D100K 也是十分稀疏的。表 2 为实验使用数据集的基本特征。

表 2 一些数据集的基本特征
Table 2 An overview of datasets in experiments

数据集	#item	#trans	ATL
Accidents	468	340183	33.8
Mushroom	119	8214	23.0
Connect	129	675557	43.0
Pumsb	7117	49046	74.0
Kosarak	36842	990007	7.1
Retail	16470	88162	10.3
T10I4D100K	870	100000	10.1
T40I10D100K	942	100000	39.6

实验的所有程序用 C/C++ 编写。其中 MAFIA 的代码下载自 <http://himalaya-tools.sourceforge.net/Mafia/>, FP-growth* 的代码下载自 <http://fimi.cs.helsinki.fi/src/>。所有的实验在一台 CPU 配置为 Intel i5-3230 2.6 GHz, 内存 2 G 的 PC 上进行。实验的操作系统为 32bit Ubuntu 12.04。

3.2 运行时间的比较和分析

图 3 展示了 3 个算法在 8 个数据集上, 基于不同的最小支持度设定的运行时间。由于 3 个算法在读入阶段都需要建立一定的数据结构来保证算法正确、有效地运行, 因此这一部分所用的时间是不可忽视的。由于最长频繁模式的数目有限, 加入输出时间对于总体运行时间不会有较大影响, 因此, 上述运行时间描述的都是程序运行的总时间。

图 3(a)为 3 个算法运行在 Accidents 数据集上的情况。在所有最小支持度下, NB-MAFIA 的时间效率在 3 个算法中都是最好的。随着最小支持度的下降, MAFIA 算法的运行时间增长非常快。当最小支持度高于 0.3 时, MAFIA 的效率优于 FP-growth*, 而在更低的最小支持度下, FP-growth* 则比 MAFIA 所用时间少。图 3(b)是关于 Mushroom 数据集的比较。显然, NB-MAFIA 仍然表现最为出色, 尽管在最小支持度低到 4% 后与 MAFIA 的运行时间基本上相同。FP-growth* 则是三者中所用时间最多的。图 3(c)展示了 3 个算法运行在 Connect 数据集上的情况。NB-MAFIA 依然具有最快的速度, 并且所用时间不到其他两个算法的一半。MAFIA 和 FP-growth* 的时间效率十分接近。图 3(d)是针对 Pumsb 数据集的测试结果展示。对于这个数据集, 在所有最小支持度下, 3 个算法的表现十分接近, 没有一个算法明显优于其他二者。我们的 NB-MAFIA 算法的表现也总不是最差的。图 3(e)是 3 个算法在 Kosarak 数据集上的情况。可以看到, NB-MAFIA 的运行时间在所有最小支持度下都比 MAFIA 少。只有在最小支持度为 0.25% 时, NB-MAFIA 的效率略逊色于 FP-growth* 算法。图 3(f)是 3 个算法在 Retail 数据集上的运行情况。在这个数据集上, FP-growth* 的效果最稳定, 也最优秀。FP-growth* 比 NB-MAFIA 快 3~5 倍。MAFIA 的表现极差, 当最小支持度较低时, 它的效率甚至比 FP-growth* 要低一个数量级以上。相似的结果在图 3(g)关于 T10I4D100K 数据集的结果中也可以看到。图 3(h)是 T40I10D100K 的结果, 只有在这一数据集上, NB-

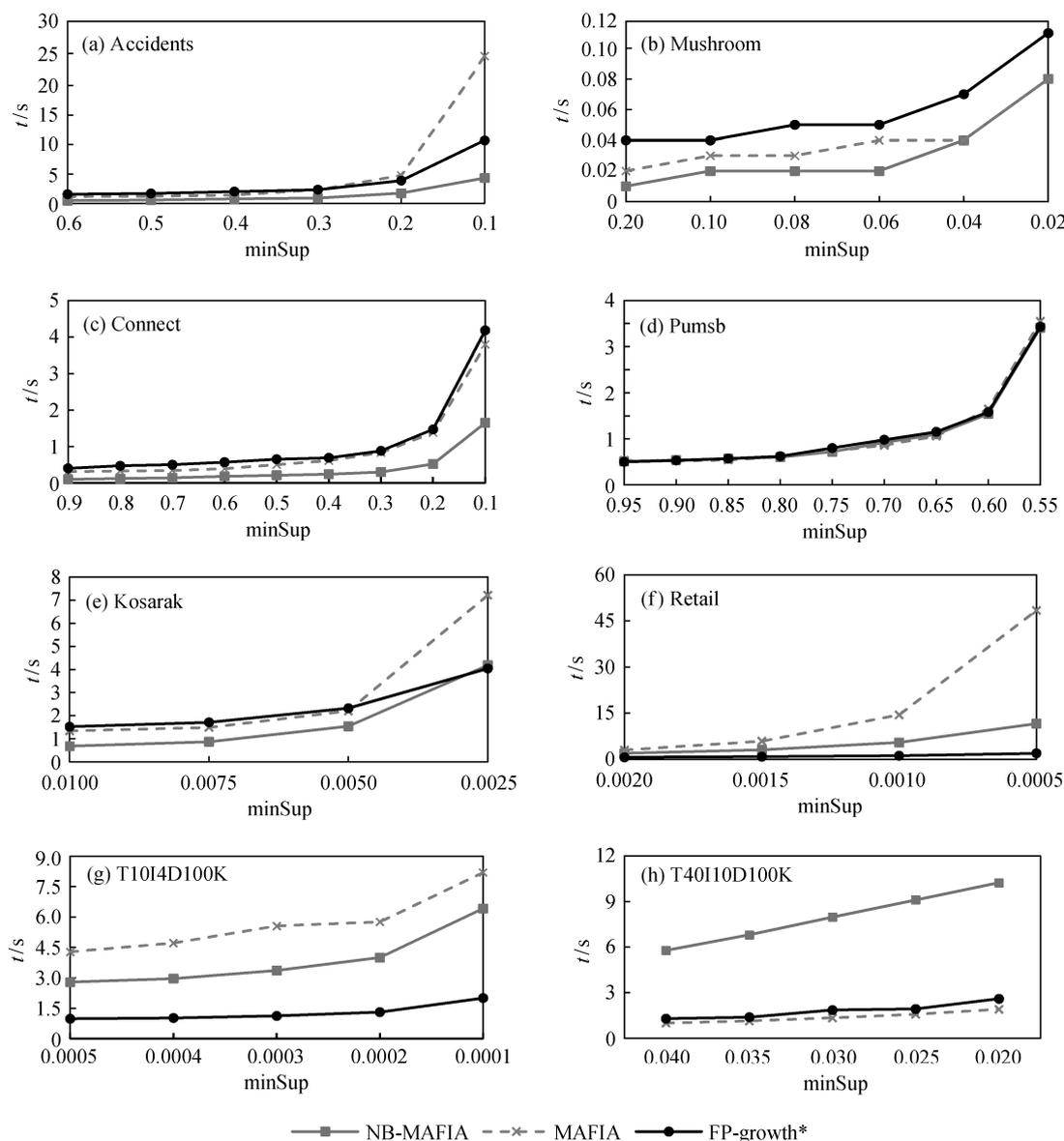


图3 3种算法在8个数据集上的运行时间

Fig. 3 Runtime of three algorithms for eight datasets

MAFIA 的效率比其他两个算法都差。

通过上述实验结果可以看到, 在稠密的数据集 Accident, Mushroom, Connect 和 Kosarak 上, 我们的 NB-MAFIA 算法都明显优于其他两个算法, 而对于稠密的 Pumsb 数据集, NB-MAFIA 也不比其他两个算法差。这主要是因为, 在稠密的数据集上, PPC-tree 对于数据集有较高的压缩率^[7], 我们可以用 N-List 来高效地表示项集、计算项集支持度。而对于稀疏的数据集, 虽然 NB-MAFIA 的表现不如 FP-growth*, 但在多数情况下依然要优于

MAFIA 算法。由于 NB-MAFIA 是在 MAFIA 搜索框架的基础上改进得到的, 因此, 我们通过实验证明了 NB-MAFIA 对 MAFIA 的改进确实有效。NB-MAFIA 表现较差的两个数据集 T10I4D100K 和 T40I10D100K 都是人造数据集, 并不能很好地表现实际应用中数据的特点。另外, 从实验结果来看, 对于挖掘所有频繁模式任务, PrePost 算法在 Retail 和 T10I4D100K 数据集上的表现与 FP-growth 算法和 FP-growth* 算法相似, 甚至有比二者差的情况^[7]。这从侧面印证了对于最长模式挖掘任务, NB-

MAFIA 和 FP-growth*算法在两个数据集上效率的相对关系。由于两个算法所用数据结构的差异,我们很难定量分析出导致两者效率差别的原因。

4 总结

本文提出一个整合算法 NB-MAFIA,可以高效地完成最长频繁模式挖掘任务。NB-MAFIA 是基于 MAFIA 算法的深度优先搜索策略,并用 N-List 数据结构来表示项集,同时适当地结合 MAFIA 中用到的剪枝策略和超集检测策略来提高搜索效率。由于 N-List 所具有的性质,我们可以通过高效地对其进行求交集操作来获得项集的支持度。我们在多个真实和仿真数据集上进行测试,验证了 NB-MAFIA 优越的性能。实验证明,在稠密的数据集上,NB-MAFIA 都有出色的表现,在 5 个数据集中的 4 个数据集上都比 MAFIA 和 FP-growth*更加高效。在稀疏的数据集上,整体上看,NB-MAFIA 也比 MAFIA 有效。在真实的数据集上,NB-MAFIA 都比 MAFIA 性能好。这说明我们的算法是对 MAFIA 的一个有效改进。我们未来的研究方向,可以针对稀疏数据集对 NB-MAFIA 算法进行进一步改进;也可以以 N-List 结构为基础,改进闭模式挖掘、高效用模式挖掘等任务的效率;还可以 Deng 等^[22]提出的相较于 N-List 更简省有效的数据结构 Nodeset 为基础,进一步提高 NB-MAFIA 的效率和可扩展性。

参考文献

- [1] Agrawal R, Imielinski T, Swami A. Mining association rules between sets of items in large databases // Buneman P, Jajodia S. SIGMOD93: Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data. Washington, DC: ACM Press, 1993: 207–216
- [2] Agrawal R, Srikant R. Fast algorithm for mining association rules // Bocca B J, Jarke M, Zaniolo C. VLDB94: The 20th International Conference on Very Large Data Bases. Santiago de Chile: Morgan Kaufmann, 1994: 487–499
- [3] Rigoutsos I, Floratos A. Combinatorial pattern discovery in biological sequences: the teiresias algorithm. *Bioinformatics*, 1998, 14(1): 55–67
- [4] Burdick D, Calimlim M, Gehrke J. MAFIA: a maximal frequent itemset algorithm for transactional databases // Georgakopoulos D, Buchmann A. ICDE2001: Proceedings of the 17th International Conference on Data Engineering. Heidelberg: IEEE Computer Society, 2011: 443–452
- [5] Burdick D, Calimlim M, Flannick J, et al. Mafia: a maximal frequent itemset algorithm. *IEEE TKDE J*, 2005, 17(11): 1490–1504
- [6] Gouda K, Zaki M J. Efficiently mining maximal frequent itemsets // Cercone N, Lin T Y, Wu X D. ICDM2001: Proceedings of the 2001 IEEE International Conference on Data Mining. San Jose: IEEE Computer Society, 2001: 163–170
- [7] Deng Z H, Wang Z H, Jiang J J. A new algorithm for fast mining frequent itemsets using N-lists. *Science China Information Sciences*, 2012, 55(9): 2008–2030
- [8] Han J W, Pei J, Yin Y W. Mining frequent patterns without candidate generation // Chen W D, Naughton J F, Bernstein, P A. Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data. Dallas: ACM Press, 2000: 1–12
- [9] Pei J, Han J W, Lu H J, et al. H-Mine: hyper-structure mining of frequent patterns in large databases // Cercone N, Lin Y T, Wu X D. ICDM2001: Proceedings of the 2001 IEEE International Conference on Data Mining. San Jose: IEEE Computer Society, 2001: 441–448
- [10] Agrawal R, Mannila H, Srikant R, et al. Fast discovery of association rules // Fayyad U M, Piatetsky-Shapiro G, Smyth P, et al. Advances in knowledge discovery and data mining. AAAI/MIT Press, 1996: 307–328
- [11] Aggarwal C C, Yu P S. Mining large itemsets for association rules. *IEEE Data Eng Bull*, 1998, 12(1): 23–31
- [12] Aggarwal C C, Yu P S. Online generation of association rules // Urban D S, Bertino E. ICDE1998: Proceedings of the 2001 IEEE International Conference on Data Mining. Orlando: IEEE Computer Society, 1998: 402–411
- [13] Dunkel B, Soparkar N. Data organization and access for efficient data mining // Kitsuregawa M, Papazoglou M P. ICDE1999: Proceedings of the 15th International Conference on Data Engineering. Sydney: IEEE Computer Society, 1999: 522–529
- [14] Ganti V, Gehrke J, Ramakrishnan R. Demon: mining and monitoring evolving data. *IEEE TKDE J*, 2001,

- 13(1): 50–63
- [15] Park J S, Chen M S, Yu P S. An effective hash-based algorithm for mining association rules // Carey M J, Schneider D A. KDD1995: International Conference on Management of Data. San Jose: ACM Press, 1995: 175–186
- [16] Savasere A, Omiecinski E, Navathe S B. An efficient algorithm for mining association rules in large databases // Dayal U, Gray P M D, Nishio S. VLDB94: Proceedings of 21th International Conference on Very Large Data Bases. Zurich: Morgan Kaufmann, 1995: 432–444
- [17] Shenoy P, Haritsa J, Sudarshan S, et al. Turbo-charging vertical mining of large database // Chen W D, Naughton J F, Bernstein, P A. SIGMOD00: Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data. Dallas: ACM Press, 2000: 22–33
- [18] Toivonen H. Sampling large databases for association rules // Vijayarman T M, Buchmann A P, Mohan C, et al. VLDB96: Proceedings of 22th International Conference on Very Large Data Bases. Mumbai: Morgan Kaufmann, 1996: 134–145
- [19] Yip C L, Loo K K, Kao B, et al. Lgen — a lattice-based candidate set generation algorithm for I/O efficient association rule mining // Zhong N, Zhou L Z. PAKDD99: Methodologies for Knowledge Discovery and Data Mining, Third Pacific-Asia Conference. Beijing, 1999: 54–63
- [20] Zaki M J, Gouda K. Fast vertical mining using diffsets // Getoor L, Senator T E, Domingos P, et al. KDD2003: The 9th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. Washington DC: ACM Press, 2003: 326–335
- [21] Deng Z H, Wang Z H. A new fast vertical method for mining frequent itemsets. International Journal of Computational Intelligence Systems, 2010, 3(6): 733–744
- [22] Deng Z H, Lv S L. Fast mining frequent itemsets using nodesets. Expert Systems with Applications, 2014, 41(10): 4505–4512
- [23] Bayardo R J. Efficiently mining long patterns from databases // Haas M L, Tiwary A. SIGMOD1998: Proceedings ACM SIGMOD International Conference on Management of Data. Washington: ACM Press, 1998: 85–93
- [24] Agarwal R C, Aggarwal C C, Prasad V V V. A tree projection algorithm for generation of frequent item sets. J Parallel and Distributed Computing, 2001, 61(3): 350–371
- [25] Gunopulos D, Mannila H, Saluja S. Discovering all most specific sentences by randomized algorithms // Kolaitis G P, Afrati N F. ICDT97: Proceedings of 6th International Conference Database Theory. Greece, 1997: 215–229
- [26] Lin D I, Kedem Z M. Pincer search: a new algorithm for discovering the maximum frequent set // Schek H, Saltor F, Ramos I, et al. EDBT98: Proceedings of 6th International Conference on Extending Database Technology. Valencia, 1998: 105–119
- [27] Zaki M J. Scalable algorithms for association mining. IEEE TKDE J, 2000, 12(3): 372–390
- [28] Grahne G, Zhu J F. High performance mining of maximal frequent itemsets // HPDM2003: Proceedings of the 6th International Workshop High Performance Data Mining. San Francisco, 2003: 135–143
- [29] Grahne G, Zhu J F. Fast algorithms for frequent itemset mining using FP-Trees. IEEE TKDE J, 2005, 17(10): 1347–1362
- [30] Holsheimer M, Kersten M L, Mannila H, et al. A perspective on databases and data mining // Fayyad U M, Uthurusamy R. KDD95: Proc Proceedings of the First International Conference on Knowledge Discovery and Data Mining. Montreal: AAAI Press, 1995: 150–155