ISSN 1000-9825, CODEN RUXUEW
Journal of Software, Vol.19, No.2, February 2008, pp.379−388
DOI: 10.3724/SP.J.1001.2008.00379

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel/Fax: +86-10-62562563

Cherry:一种无须子集检查的闭合频繁集挖掘算法*

陶利民+, 黄林鹏

(上海交通大学 计算机科学与工程系,上海 200032)

Cherry: An Algorithm for Mining Frequent Closed Itemsets without Subset Checking

TAO Li-Min+, HUANG Lin-Peng

(Department of Computer Science and Engineering, Shanghai Jiaotong University, Shanghai 200032, China)

+ Corresponding author: Phn: +86-13918512358, Fax: +86-21-58451044, E-mail: charleswilliams@163.com, http://www.sjtu.edu.cn

Tao LM, Huang LP. Cherry: An algorithm for mining frequent closed itemsets without subset checking. *Journal of Software*, 2008,19(2):379−388. http://www.jos.org.cn/1000-9825/19/379.htm

Abstract: Through the theoretical analysis and research works on some famous mining algorithms, a new mining algorithm named Cherry is proposed in this paper. It bases on FP-tree technology and adopts a novel Cherry-Items-detecting technology. This novel technology can find those prefixes which result to the unclosed or redundant frequent itemsets without maintaining the frequent closed itemsets mined so far in the main memory. In the performance test, the Cherry algorithm is compared with other state of the art algorithms, such as FP-CLOSE, LCMv2 and DCI-CLOSE, in many synthetic and real data sets. The experimental results demonstrate that the Cherry algorithm outperforms them in low support.
Key words: association rule; frequent closed itemset

摘 要: 通过对一些著名的闭合频繁集挖掘算法(如 CLOSET+, FP-CLOSE,DCI-CLOSED 和 LCMv2 等)的研究并结合挖掘理论分析,提出了一种新的挖掘算法 Cherry,它基于 FP-tree 结构,并采用了新颖的 Cherry Item 检测技术,无须在内存中保留闭合频繁集而直接检测出会导致重复的频繁项前缀,从而极大地提高了挖掘效率.性能实验的比较和测试表明,该 Cherry 算法在低支持度的测试中要优于目前的一些主流挖掘算法,如 LCMv2,DCI-CLOSE 和 FP-CLOSE 等.
关键词: 关联规则;闭合频繁集
中图法分类号: TP311 **文献标识码**: A

频繁集的挖掘在数据挖掘领域有着广泛的应用.虽然频繁集的挖掘在近些年来被深入研究,也提出了一些比最初的 Apriori[1]性能更好的算法,但是,一旦支持度降低,频繁集的数量就会急剧增加,这些算法的性能也急剧下降.随后,一些学者提出了闭合频繁集和最大频繁集的概念.目前,在数据挖掘领域内,对于不同的应用,已经形成了频繁集、闭合频繁集和最大频繁集这 3 种各有侧重的挖掘方法.本文主要对闭合频繁集的挖掘理论进行研究,对基于 FP-tree 技术的挖掘算法提出了一项 Cherry Item 检测技术,用来取代传统的子集检查技术,并可以解决在挖掘过程中必须在内存中保留所有已挖掘到的闭合频繁集的问题,从而有效地提高了算法的性能.

* Supported by the National Natural Science Foundation of China under Grant No.60673116 (国家自然科学基金)
Received 2006-02-21; Accepted 2006-10-10

1 问题介绍

在数据挖掘领域内,闭合频繁集是指那些不存在与其支持度相同但却是其超集的频繁集.闭合频繁集与一个事务数据库中所有的频繁集在语义上是完全等价的,但在数量上却大量减少.闭合频繁集的概念采用文献[2]中的描述,用以下两个函数 f 和 g 来定义:

$$f(T)=\{i\in I|\forall t\in T,i\in t\},$$

$$g(A)=\{t\in D|\forall i\in A,i\in t\},$$

其中,$I=\{a_1,\ldots,a_M\}$,I 是一个有限的项的集合,D 是一个事物数据库,包含了有限的事物,D 中的每个事物 $t\in D$ 是一个没有重复项的集合,$t=\{i_1,i_2,\ldots,i_T\}$,$\forall i_j\in t\Rightarrow i_j\in I$.$A$ 是一个项集,且 $A\subseteq I$.

定义 1. 一个项集 A 是闭合的,当且仅当 $c(A)=f(g(A))=f\circ g(A)=A$,其中,复合函数 c 可以称为闭包函数或闭包操作.

定义 2. 如果一个项集 A 是闭合的,且$|g(A)|=\sup(A)\geq min_sup$,则称该项集为闭合频繁集,其中,$\sup(A)$表示在事物数据库中包含该项集 A 的事物的数量,min_sup 为给定的最小支持度.

本文第 1 节、第 2 节对闭合频繁集的挖掘问题和目前一些主流挖掘算法的思路进行介绍.第 3 节提出 Cherry Item 检测技术以及对前缀项集进行闭包计算技术.第 4 节对 Cherry 算法进行详细介绍.最后两节是性能测试实验结果分析和结论.在本文的性能测试实验中,我们主要与国外的一些针对闭合频繁集挖掘的算法进行比较和测试,这是因为目前国内提出的大多是针对最大频繁集的挖掘算法,如文献[3,4]等,且挖掘到的最大频繁集在数量上要远小于闭合频繁集,因此,这两类算法不能进行性能分析和比较.

2 相关研究

A-CLOSE[3]是第一种对闭合频繁集进行挖掘的算法.CLOSET+算法是 Wang 等人在文献[5]中提出的一个著名的基于 FP-tree 技术的闭合频繁集挖掘算法,在 ICDM(2003)会议组织的 FIMI 测试中获得第一的 FP-CLOSE 就是 CLOSET+算法的变种.之后,在 ICDM(2004)会议组织的 FIMI 测试中,两种无须在内存中保留闭合频繁集的算法被提出来,其中之一就是在这次测试中获得第一的 LCMv2,另一种是 DCI-CLOSE 算法.它们无论是在内存的使用量上还是在挖掘的效率上,都超过了 CLOSET+和 FP-CLOSE,代表了目前世界上性能最好的闭合频繁集挖掘算法.下面就这些算法的挖掘思路分别予以简单介绍.

FP-CLOSE 和 CLOSET+算法的理论基础都是采用了一种“频繁模式增长(FP-Growth)”的策略来进行搜索,在实现过程中,采用了 FP-tree 结构在内存中存储全局的事物数据库.即在 FP-tree 中,每个节点代表一个频繁项和它在某些事物中的共享支持度.从叶节点通过父指针可以回退到根节点的这样一个链表称为一条路径.这条路径所包含的节点即代表了包含这些项的事物的集合.但是 CLOSET+和 FP-CLOSE 有一个共同的弱点,就是在挖掘过程中必须在内存中保留所有已经挖掘到的闭合频繁集,虽然在支持度较高时,闭合频繁集的数量不多,即使把它们全部保留在内存中,这两种算法也能得到较令人满意的挖掘性能,但是随着支持度的降低,闭合频繁集的数量急剧增加,这两种算法的性能也逐渐降低.例如,对于常被用作测试算法性能的 chess 事物数据库,它只有 3 196 个事物,但当支持度降低到 1 500 时,也即支持度在 47%时,闭合频繁集的数量将达到 549 872 个之多.此时,由于这些闭合频繁集占用了大量的内存,导致操作系统不得不进行大量的内存页面调度,从而极大地影响了算法的性能.

下面,我们用一个例子来说明 FP-CLOSE 和 CLOSET+算法在挖掘过程中生成重复闭合频繁集,并用子集检查技术进行修正的过程.我们以表 1 所示的全局事物数据库为例,并以最小支持度为 1 进行闭合频繁集的搜索.算法首先对表 1 所示的全局事物数据库进行两次遍历:第 1 次,计算各个项的支持度,删去小于最小支持度的项,并进行排序,得到频繁项的支持度,见表 2;第 2 次,把每个事物所包含的项按表 2 的顺序排序后,插入到全局 FP-tree 中,最后得到如图 1(a)所示的全局 FP-tree.在该图中,实线指针为树的节点指针,指向某个节点的父节点或子节点;虚线指针为连接指针,指向该节点在 FP-tree 树中的其他节点.随后,算法依照表 2 中的顺序从支持度最

小的项开始挖掘闭合频繁集.算法首先构建以项 *A* 为前缀的子 FP-tree,如图 1(b)所示.对该树进行分析,得知其是一棵单链树,由此,可以得到一个包含前缀项集 *A* 的闭合频繁集 *ACF*:1,并把其保留在内存中,供子集检查时使用.用同样的方法对表 2 中的项 *C* 进行挖掘,可以得到如图 1(c)所示的以项 *C* 为前缀项的子 FP-tree,并得到 *CF*:1 这样的一个频繁集,但是为了保证结果的正确性,算法必须进行子集检查.通过检查我们发现,*CF*:1 与已经得到的闭合频繁集 *ACF*:1 支持度相同,但却是后者的子集,所以,*CF*:1 必定是一个不闭合的频繁集,也即重复发现的频繁集.因此,*CF*:1 必须被删除.

Table 1 A transaction database (TDB)

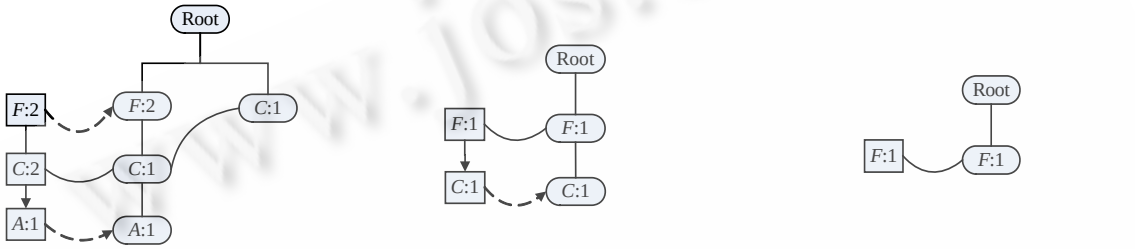
表 1 一个事物数据库(TDB)

Tid	Items	Frequent items
100	*A,C,F*	*F,C,A*
200	*F*	*F*
300	*C*	*C*

Table 2 Global frequent items sequence

表 2 全局频繁项序列

Sequence	Item	Support
1	*F*	2
2	*C*	2
3	*A*	1



(a) Global FP-tree (b) Projected (Sub) FP-tree with prefix item *A* (c) Projected (Sub) FP-tree with prefix item *C*

(a) 全局 FP-tree (b) 以项 *A* 为前缀的投影(子)FP-tree (c) 以项 *C* 为前缀的投影(子)FP-tree

Fig.1 FP-Trees in FP-CLOSE algorithm and CLOSET+ algorithm

图 1 FP-CLOSE 和 CLOSET+算法中的 FP-tree

DCI-CLOSE 算法的作者则采用了一种完全不同的方法进行闭合频繁集的挖掘.他把只包含 1 个项的频繁集称为构造因子(generator),并不断地在最初的构造因子中添加新项,对得到的新前缀进行构造因子检测,从而决定是否继续挖掘.而对构造因子的检测是采用作者提出的一种序保持(order preserving)的操作,该操作无须在内存中保留已有的闭合频繁集.同时,它在实现的过程中采用了向量和位图技术来进行存储和比较,因此它的挖掘效率相对较高.

而 LCMv2 采用了一项称为 PPC(prefix-preserving closure extension)的技术.它也是从最短的频繁集开始挖掘,对这些频繁集称为 ppc,并在其上定义一个闭包扩展操作(closure extension)和闭包末位操作(closure tail).通过闭包扩展操作,使得一个 ppc 扩展成为一个闭合频繁集,同时在其上添加大于该闭合频繁集闭包末位操作的项,并检测是否形成一个新的 ppc,从而继续挖掘.这种算法在实现过程中只采用数组,同时,它所采用的理论相对比较先进,因而挖掘的效率较高.

通过上述算法的挖掘思路我们可以看出,CLOSET+及其变种 FP-CLOSE 是采用挖掘到频繁集后用子集检查技术进行闭包性检测,从而得到正确的结果;而 DCI-CLOSE 和 LCMv2 则是在生成频繁集之前就检测某个前缀项集是否会导致重复,从而只生成不重复的闭合频繁集.显然,后者的效率要高于前者.因此,我们提出的 Cherry 算法正是针对 CLOSET+和 FP-CLOSE 的弱点进行了改进,提出在挖掘过程中对前缀项集进行 Cherry Item 的检测来代替子集检查技术,从而无须在内存中保留所有已经挖掘到的闭合频繁集,也能直接得到正确的结果.

3 Cherry Item检测技术和前缀项集闭包计算技术

为了解决 CLOSET+及其变种 FP-CLOSE 算法中搜索到重复闭合频繁集的问题,我们首先要在理论上分析

前缀项集和闭合频繁集之间的关系,于是引入如下定理:

定理 1. 给定一个前缀项集 P 和一个闭合频繁集 C,则有 $P\subseteq C\Leftrightarrow g(P)\supseteq g(C)$.

证明:首先证明 $P\subseteq C\Rightarrow g(P)\supseteq g(C)$,因为 $P\subseteq C\Rightarrow C=P\cup(C-P)$,则 $g(C)=g(P\cup(C-P))=g(P)\cap g(C-P)$,所以 $g(P)\supseteq g(C)$.

其次证明 $P\subseteq C\Leftarrow g(P)\supseteq g(C)$,因为 $g(P)\supseteq g(C)$,所以 $g(C)=g(P)\cap g(C)=g(P\cup C)$,即 $C=P\cup C\Rightarrow P\subseteq C$. □

定理 1 告诉我们,如果在包含某个前缀项集的事物集范围内进行搜索,就能搜索到所有包含该前缀项集的闭合频繁集.一旦完成了这样的一次搜索,我们就可以确信,所有包含该前缀的闭合频繁集都被发现了.显然,这些闭合频繁集与前缀项集之间必定有某种联系,因此,我们对已经被当作前缀搜索过的项分成如下两种类型:

定义 3. 给定一个项$\{i\}$,如果任取一个闭合频繁集 C,只要 $g(g\{i\})\supseteq g(C)$,C 必定是已经被挖掘到的闭合频繁集,则称该项 i 为全局已搜索项.

定义 4. 给定一个前缀项集 P 和一个项 i,$i\notin P$.如果任取一个闭合频繁集 C,只要 $g(P\cup\{i\})\supseteq g(C)$,$C$ 必定是已经被挖掘到的闭合频繁集,则称该项 i 为前缀项集 P 下的局部已搜索项.

至此,我们可以给出 Cherry Item 项和 Cherry Item 项集的定义:

定义 5. 在给定的前缀项集 P 下,所有全局已搜索项和在前缀 P 下的局部已搜索项的集合称为 Cherry Item 项集,其中任意一个项称为 Cherry Item 项.

最后,我们给出可以替代子集检查技术的定理,并用以避免搜索到那些已经被发现的闭合频繁集.

定理 2. 给定一个前缀项集 P 和 Cherry Item 项集 H,项 i 是在前缀 P 下局部支持度大于等于最小支持度的项,即$|g(P\cup\{i\})|\geq$min_sup.如果存在一个项 j 且 $j\in H$,使得$\forall t\in g(P\cup\{i\})$均有 $j\in t$,则可以无须以 $P\cup\{i\}$为前缀项集进行搜索,因为这种搜索必定导致发现重复的闭合频繁集.

证明:根据定理 1,任何闭合频繁集 C 只要包含 $P\cup\{i\}$,就必定有 $g(C)\subseteq g(P\cup\{i\})$.因为 j 是 Cherry Item,所以 j 有可能是全局已搜索项或在前缀项集 P 下的局部已搜索项.为此,我们首先假设 j 是全局已搜索项,那么根据条件$\forall t\in g(P\cup\{i\})$均有 $j\in t$,则 $g(P\cup\{i\})\subseteq g(\{j\})$,所以 $g(C)\subseteq g(\{j\})$,根据定义 3,C 必定是已经挖掘到的闭合频繁集;其次,我们假设 j 是在前缀项集 P 下的局部已搜索项,那么根据条件:$\forall t\in g(P\cup\{i\})$均有 $j\in t$,则 $g(P\cup\{i\})\subseteq g(P\cup\{j\})$,所以 $g(C)\subseteq g(P\cup\{j\})$,根据定义 4,$C$ 必定也是已经挖掘到的闭合频繁集. □

有了该定理,我们在挖掘闭合频繁集时,可以直接检测出会导致生成重复闭合频繁集的前缀项集.例如,当前的前缀项集为 P,当前某个未处理的项为 i,则在构成一个新前缀项集 $P\cup\{i\}$以后,利用定理 2 检测所有包含该项集的事物中是否都包含某一个 Cherry Item 项.如果没有,就能保证任何包含 $P\cup\{i\}$的闭合频繁集都不是重复的.因此,这种基于对 Cherry Item 项的检测可以无须保留已挖掘到的闭合频繁集而达到与子集检查同等的效果,所以极大地提高了搜索的效率.

接下来的问题就是如何采用一种合理的形式把 Cherry Item 项保留在内存中.为此,我们对 FP-tree 作了一些改动:如果当前 FP-tree 上的某个项被添加到前缀项集中,并以此来搜索过闭合频繁集,则可以把所有表示该项的节点均插入到其父节点的 Cherry Item 项链表内.但同时,根据定理 2 的规定,这些 Cherry Item 项必须在包含其父节点的事物集中全部出现才能被插入.例如,上例中的项 A 被当作前缀项集进行搜索后,我们可以把全局 FP-tree 中节点 A:1 作为 Cherry Item 项插入到其父节点的 Cherry Item 项链表内,是因为其父节点的支持度与 Cherry Item 项的支持度相等.这些被插入到 Cherry Item 项链表内的节点在图 2(a)中就像是挂在树上的小樱桃,这也是我们把它命名为 Cherry Item 的一个原因.

此外,在闭合频繁集的挖掘过程中我们还提出了一项对前缀项集进行了闭包计算的技术,使得每个前缀项集都是一个闭合频繁集.比较而言,CLOSET+和 FP-CLOSE 只能以频繁集为前缀.为此,我们介绍如下定理:

定理 3. 给定一个前缀项集 P 和项 i,且 $i\notin P$.如果 sup(P)=sup($P\cup\{i\}$),则任取一个闭合频繁集 C,只要 $P\subset C\Rightarrow i\in C$.

证明:首先,因为 sup(P)=sup($P\cup\{i\}$),所以$\forall t\in g(P)$均有 $i\in t$;其次,因为 $P\subset C$,根据定理 1,必有 $g(C)\subseteq g(P)$.因此,可以得出$\forall t\in g(C)$均有 $i\in t$,即 $i\in f(g(C))$,也即 $i\in C$. □

有了定理 3 之后,当我们得到一个前缀项集 P 时,可以在未处理的项中检测一下是否有这样的项 i,使得 $\sup(P)=\sup(P\cup\{i\})$,则根据定理 3,凡是包含 P 的闭合频繁集都会包含项 i,所以项 i 实质上也可看作所有包含 P 的闭合频繁集的前缀,因此,我们可以把所有这样的项从未处理的项中删除,并添加到 P 中形成一个新的前缀项集 P',则必定有 $P'=f(g(P))$.所以,根据定义 1,P'必定是闭合的,且只要 $\sup(P)\geq min\_sup$,P'就是一个闭合频繁集.采用这项针对前缀项集的闭包计算技术,在搜索过程中,可以减少在某个前缀项集下未处理的项的数量,进而减少搜索的范围,提高搜索的效率.

4 Cherry算法

通过前面的介绍,我们把 Cherry 算法对 CLOSET+和 FP-CLOSE 所作的改进了进行详细的分析.下面我们用表 1 这个例子对 Cheery 算法的具体挖掘过程作一些探讨.

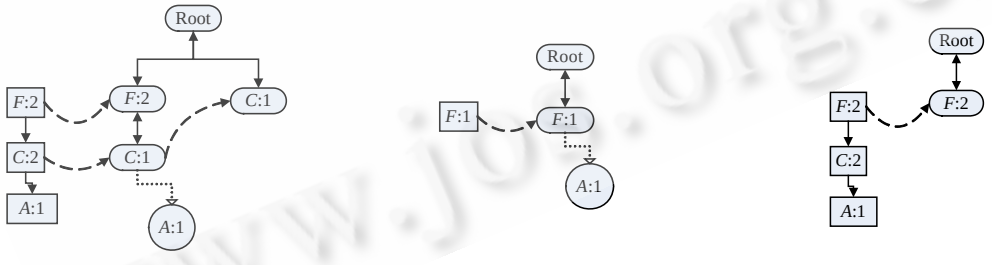
首先,Cherry 算法采用与 CLOSET+和 FP-CLOSE 相同的方法,对事物数据库进行两次遍历,并建立全局 FP-tree,如图 1(a)所示.

其次,Cherry 算法也从支持度最低的项 A 开始搜索,并形成初始的前缀项集={A}.然后对全局 FP-tree 中所有包含项 A 的节点进行遍历,统计在包含项 A 的事物集中局部支持度大于等于最小支持度的未搜索项,得到表 3.由于项 A 在全局 FP-tree 中的支持度为 1,由表 3 可知,项 F 和项 C 的局部支持度均为 1,与前缀项集的支持度相同,所以根据定理 3,可以把这两个项添加到前缀项集中,使其成为闭合频繁集并输出,同时,从未搜索项中删除项 F 和项 C.由于这两个项的删除,在以闭合频繁集 ACF:1 作为前缀项集的情况下,已经不存在未搜索项,所以无须构建子 FP-tree.至此,以项 A 为前缀项集的搜索结束,算法只需把项 A 作为 Cherry Item,并把它挂到其在 FP-tree 中相应的父节点中,如图 2(a)所示.

Table 3 Frequent items sequence with prefix item A

表 3 前缀项 A 的频繁项序列

Sequence	Item	Support
1	F	1
2	C	1



(a) Global FP-tree after item A is processed (b) Projected FP-tree with prefix item C (c) Global FP-tree after item C is processed

(a) 项 A 已处理后的全局 FP-tree (b) 以项 C 为前缀的子 FP-tree (c) 项 C 已处理后的全局 FP-tree

Fig.2 FP-Trees while adopting Cherry Item detecting technique

图 2 采用 Cherry Item 检测技术的 FP-tree

接着,Cherry 算法以项 C 为初始的前缀项集进行搜索.同样对全局 FP-tree 中所有包含项 C 的节点进行遍历,得到表 4 所列的未搜索项.由于前缀项集 C 的支持度为 2,但是在表 4 中没有任何一个项的支持度为 2,所以 C:2 必定是闭合频繁集,可以输出.然后,在图 2(a)的基础上再次遍历所有包含项 C 的节点,并构建子 FP-tree.由图可知,合并事物集{C,F}:1 中包含一个 Cherry Item 项 A,其支持度等于合并事物集{C,F}:1 的支持度,因此,该 Cherry Item 项可能会在检测重复的闭合频繁集中起作用,所以在构建子 FP-tree 时必须保留该信息.于是,算法把合并事物集中的项 F 插入到子 FP-tree 以后,也要把项 A 插入该节点的 Cherry Item 项链表内,得到如图 2(b)所示的子 FP-tree.由图可知,唯一的局部未搜索项 F 所在的事物集中包含一个 Cherry Item 项 A,所以根据定理 2,如果以项

集 CF 为前缀进行搜索,则必然得到重复的闭合频繁集.同时,由于该子 FP-tree 中没有其他未搜索项,所以搜索结束,回退到全局 FP-tree,并把项 *C* 作为 Cherry Item 项,挂到其相应的父节点上.但是,在把节点 *C*:1 挂到其父节点 *F*:2 时发现,Cherry Item 项的支持度小于节点的支持度,这表明,这些 Cherry Item 并没有全部在包含节点 *F*:2 的事物集中出现,不满足定理 2 的要求,因此不能被加入,于是得到如图 2(c)所示的全局 FP-tree.

Table 4 Frequent items sequence with prefix item *C*

表 4 前缀项 *C* 的频繁项序列

Sequence	Item	Support
1	*F*	1

最后,全局 FP-tree 只包含一个未搜索项,所以可以把项 *F* 作为前缀项集进行搜索,得到最后一个闭合频繁集 *F*:2 后,算法结束.

至此,Cherry 算法对 CLOSET+和 FP-CLOSE 算法所做的改进已经全部介绍完毕,下面给出 Cherry 算法的伪代码.

```
main(string pathname,int min_sup) {   //每棵 fptree 包含一个 HeaderTable、一个指向根节点的指针 lpRoot 和闭合前缀项 preClsItems;
  gFp→HeaderTable=scanDB1(pathname,min_sup);
  gFp→lpRoot=scanDB2(pathname,min_sup,gFP→HeaderTable);
  gFp→preClsItems=NULL;
  search-cfi(gFp);
}
search-cfi(fptree pntFp) {
  while (pntFp→HeaderTable!=NULL) {
    fi=pntFp→HeaderTable→getItem();   //取得 HeaderTable 中支持度最小的项
    scanFP1(pntFp,fi,childFp);
    if (childFp→preClsItems!=NULL) output(childFp→preClsItems);
    if (childFp→HeaderTable!=NULL) {
      scanFp2(pntFp,fi,childFp);
      if (childFp→isSingleChail()==TRUE) output(childFp→preClsItems+childFp→HeaderTable) else
      search-cfi(childFp)
    }   //if
    pntFp→HeaderTable→delete(fi);   //删除已搜索过的项
  }   //while
}
scanFp1(fptree pntFp,item fi,fptree chdFp) {
  lpNode=pntFp→HeaderTable→getNode(fi);   //取 fptree 中包含项 fi 的第一个节点
  while (lpNode!=NULL) {
    chdFp→HeaderTable→count_item_support(lpNode→lpPntNode);   //统计在以 fi 为前缀项时其他未处理项的支持度
    //统计在以 fi 为前缀项时其他未处理项所包含的 Cherry Item 的支持度
    chdFp→HeaderTable→count_cherryitem_support(lpNode→lpCherryItem);
    lpNode=pntFp→getNext (lpNode)   //取 fptree 中包含项 fi 的下一个节点
  }   //while
  for each chdItem in chdFp→HeaderTable {
```

```
    if (chdItem→iCherryItemSuppor==chdItem→iSupport) chdFp→HeaderTable→delete(chdItem);  //根据
    定理 2 删除
    if (chdItem→iSupport==fi→iSupport) {
      chdFp→PreClsItems=chdFp→PreClsItems+chdItem;  //根据定理 3 增加,使前缀项闭合
      chdFp→HeaderTable→delete(chdItem);  //根据定理 3 删除
    }  //if
  };  //for each
  if (chdFp→PreClsItems!=NULL) chdFp→PreClsItems=chdFp→PreClsItems +pntFp→preClsItems
}
scanFp2(fptree pntFp,item fi,fptree chdFp) {
  lpNode=pntFp→HeaderTable→getNode(fi);  //取 fptree 中包含项 fi 的第 1 个节点
  while (lpNode!=NULL) {
    chdFp→build(lpNode→lpPntNode);  //构建子 fptree 的分支
    //把父 fptree 包含的 Cherry Item 项传递给子 fptree,确保其能够用定理 2 进行检测
    if (lpNode→iSupport==lpNode→iCherryItemsSupport) chdFp→addCherryItems(lpNode);
    //把父 fptree 中的 fi 节点加入到其父节点的 Cherry Item 项链表中
    pntFp→addCherryItemsToPntNode(lpNode,lpNode→lpPntNode)
    lpNode=pntFp→getNext (lpNode)  //取 fptree 中包含项 fi 的下一个节点
  }  //while
}
```

5 性能比较实验

我们对 Cherry 算法与一些主流算法(如 FP-CLOSE,DCI-CLOSE 和 LCMv2)进行了比较,其中未包括 CLOSET+算法,因为根据文献[2,4]中的性能比较实验,DCI-CLOSE 算法已经全面超越了 CLOSET+算法.这 3 种算法的代码由相应的作者提供,我们在一些公共的事务数据库(mushroom,chess,connect,pumsb,pumsb* 和 accidents)上对这 4 种算法进行了测试.这些公共事务数据库的基本情况和测试点的选择见表 5.

Table 5 General information of the TDB and test points

表 5 事物数据库的基本信息和测试点

TDB name	MUSHROOM		CHESS		PUMSB_*		PUMSB		CONNECT		ACCIDENTS	
Number of transaction	8 124		3 196		49 046		49 046		67 557		340 183	
Test point 1	100	1.23%	1 500	46.93%	8 500	17.33%	25 000	50.97%	11 000	16.28%	70 000	20.58%
Test point 2	50	0.62%	1 200	37.55%	7 000	14.27%	23 000	46.89%	9 000	13.32%	50 000	14.70%
Test point 3	10	0.25%	1 000	31.29%	5 500	11.21%	22 000	44.86%	7 000	10.36%	40 000	11.76%
Test point 4	20	0.12%	800	25.03%	4 000	8.16%	21 000	42.82%	5 000	7.44%	35 000	10.29%
Test point 5	5	0.06%	600	18.77%	3 000	6.12%	20 000	40.78%	3 000	4.44%	30 000	8.82%

我们的测试平台是在 Linux, P4-2.8GHz 和 512M 内存的机器上.我们测试的方法是严格按照 ICDM(2003)及 ICDM(2004)FIMI 主题测试所采用的方法:

首先,要求每种算法都采用 gcc 的 make 命令进行编译和连接,生成在 Linux 下可执行的文件.

其次,要求每种算法均采用统一的参数输入格式,如"*fim_closed* 〈*filename*〉〈minsup〉[〈*outputfile_name*〉]",其中,事物数据库名称及最小支持度是必选参数,而结果文件的生成是可选的.

最后,要求每种算法在搜索过程中不得输出任何信息,搜索结束后必须输出不同长度的闭合频繁集的数量,即第 1 行为所挖掘到的闭合频繁集的总数量,第 2 行为仅包含 1 个频繁项的闭合频繁集数量,第 3 行为包含 2 个频繁项的闭合频繁集数量,以此类推,直到长度最长的闭合频繁集统计数量输出后结束.

采用这种测试方法以后,为了得到输出信息,参加测试的算法必须挖掘到所有的闭合频繁集,所以,保证了测试的完整性.同时,考虑到如果生成数以千兆计的闭合频繁集的结果文件会影响系统的稳定性,因此,我们在测试时选择了不输出结果文件.算法运行时间采用 Linux 下的 time 命令进行统计,每个支持度运行 3 次,取均值作为结果,并予以记录.同时,我们请 DCI-CLOSE 的作者 Claudio Lucchese 在意大利对 Cherry 算法进行了测试,他通过 E-mail 确认了 Cherry 算法的高效性.此次实验结果如图 3 所示,从中可以看出,Cherry 算法在低支持度下的性能要优于其他 3 种算法.

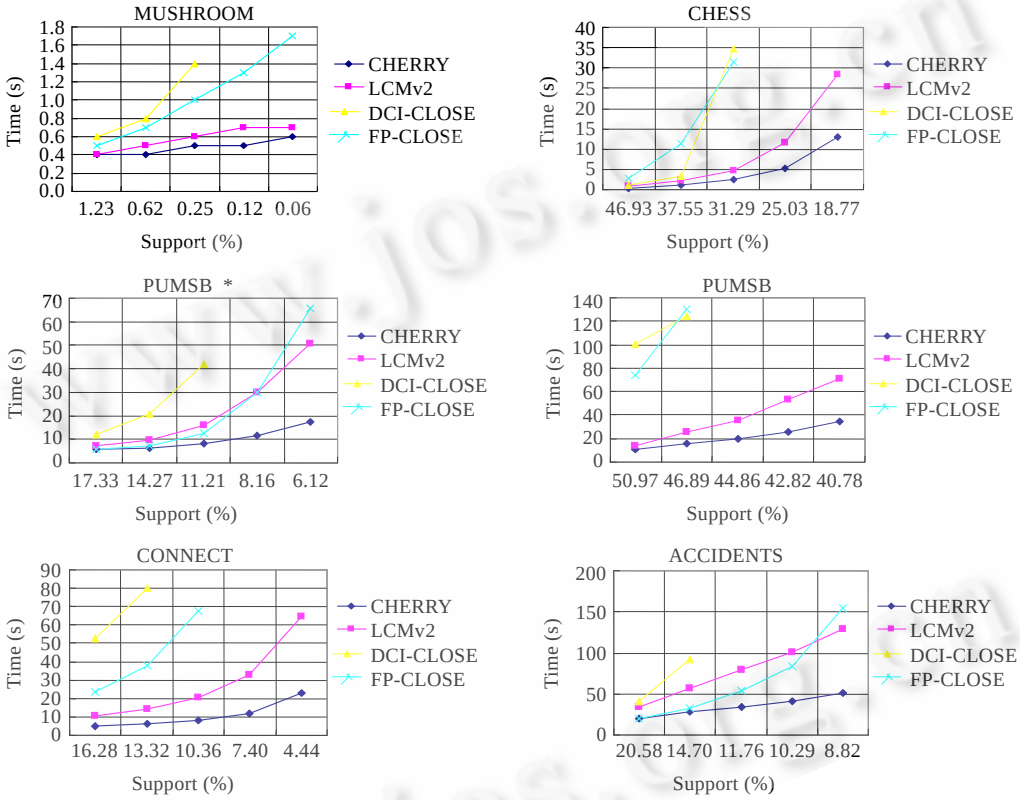


Fig.3 Execution times of Cherry, DCI-CLOSE, LCMv2 and FP-CLOSE when support falls

图 3 当支持度降低时 Cherry,DCI-CLOSE,LCMv2 和 FP-CLOSE 的执行时间

最后,我们对 Cherry 算法在性能上的优势进行一些分析.首先与同样是采用 FP-tree 技术的算法 FP-CLOSE 进行比较.Cherry 算法和 FP-CLOSE 算法都是 CLOSET+算法的变种,FP-CLOSE 主要对子 FP-tree 树的构建进行了改进(详见文献[6]),利用一个额外的数组存储父 FP-tree 中各个项的相对支持度,在构建子 FP-tree 树时,只需扫描该额外的数组即可建立子树的项头表,从而省略了对父 FP-tree 的一次扫描.因此,在 FP-CLOSE 算法中,从父 FP-tree 树构建子树时只需扫描父树一遍.虽然 FP-CLOSE 算法在子树构建上效率较高,但它仍然没有解决重复性的问题,需要采用与 CLOSET+算法相同的子集检查技术来保证算法结果的正确性.相对而言,我们的 Cherry 算法很好地解决了这个问题.从算法的伪代码中可以明确地看出,虽然 Cherry 算法要对父 FP-tree 树进行两次扫描来构建子树,但在两次扫描过程中,我们加入对 Cherry Item 的检测和构建,从而避免了采用子集检查技术.而我们知道,子集检查技术是要在已发现的闭合频繁集中找出某个前缀是否已经存在,因此,该技术所花费的时间也必定与闭合频繁集的数量成正比,因此,随着支持度的逐渐降低,闭合频繁集数量的逐渐增加,该技术对算法的影响会越来越明显.从图 3 所示的实验结果也可以看出,我们的 Cherry 算法与 FP-CLOSE 算法的效率随着支持度的降低,差距越来越大,其中关键的原因就是子集检查技术.由于无须保留已发现的闭合频繁集,因

此,Cherry 算法的效率在低支持度下的性能明显要优于 FP-CLOSE 算法.

我们再来分析 Cherry 算法与 LCMv2 算法和 DCI-CLOSE 算法的差别.这 3 种算法都是无须保留已发现闭合频繁集而直接进行挖掘的算法,在实现的过程中采用了各自的数据结构和检测理论.例如,LCMv2 算法是采用数组结构来存储事物数据库,DCI-CLOSE 算法是采用向量来存储,而 Cherry 算法是采用 FP-tree 结构来存储.但是,从对 LCMv2 算法的性能分析(文献[7])来看,FP-tree 是一种较为理想的事物数据库存储结构,其压缩比非常高.FP-tree 技术与数组结构相比,在某些类型的事物数据库上的压缩比可达到近 1:6 的程度.因此,目前为止,FP-tree 技术被认为是对事物数据库最有效的压缩方法之一,要优于数组和向量结构.这一点可以从 4 种算法在对 ACCIDENTS 这个事物数据库的挖掘上得到体现.例如,我们知道,FP-CLOSE 算法是要进行子集检查的,而 LCMv2 算法是直接进行挖掘的,因此,LCMv2 算法的挖掘效率要高于 FP-CLOSE 算法.但是在 ACCIDENTS 这个事物数据库中却恰恰相反,这是为什么呢?因为 ACCIDENTS 是一个大型的事物数据库,有将近 34 万条记录,FP-tree 结构的高压缩性得到了很好的体现,即便 FP-CLOSE 算法要进行子集检查,但由于是在一个已被压缩的 FP-tree 结构上进行挖掘,而该结构也明显地小于一个有 34 万条记录的数组结构,因此在支持度大于 35 000,也即 10%时,FP-CLOSE 算法的挖掘效率还是比 LCMv2 算法要高.直到支持度降低到 35 000 和 30 000 之间的某一点时,由于闭合频繁集的数目大量增加,LCMv2 算法才在挖掘效率上超过了 FP-CLOSE 算法.由此可以看出,一个高效的压缩结构对挖掘效率的影响也是非常大的.我们的 Cherry 算法也采用了与 FP-CLOSE 算法相同的 FP-tree 结构进行存储,同时又很好地解决了闭合频繁集挖掘中重复前缀项的检测问题,因此,该算法在低支持度下的挖掘性能明显要优于其他 3 种算法.

6 结 论

我们对闭合频繁集挖掘的理论进行了研究,提出了基于 FP-tree 的一项 Cherry Item 检测技术,并可以替代 CLOSET+算法及其变种 FP-CLOSE 算法所采用的子集检查技术.同时,该技术无须在内存中保留已经挖掘到的闭合频繁集,而同样可以检测出可能导致重复的频繁项前缀.此外,我们还对 CLOSET+算法及其变种 FP-CLOSE 算法所采用的频繁项前缀的选择进行了改进,使得每次在构建子 FP-tree 时可以闭合频繁集为前缀.通过这两项技术的运用,我们实现的 Cherry 算法解决了困扰闭合频繁集挖掘的重复性问题,提高了挖掘效率.性能比较实验的测试结果表明我们的理论是正确的.Cherry 算法在低支持度下的性能明显要优于 FP-CLOSE,DCI-CLOSE 和 LCMv2 等目前的一些主流算法.

References:

[1] Lu JP, Yang M, Sun ZH, Ju SG. Fast mining of global maximum frequent itemsets. Journal of Software, 2005,16(4):553−560 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/16/553.htm

[2] Lucchese C, Orlando S, Perego R. Mining frequent closed itemsets without duplicates generation. Technical Report, ISTI-2004-TR-13, 2004. http://dienst.isti.cnr.it/cover/tmp.html

[3] Yan YJ, Li ZJ, Chen HW. Efficiently mining of maximal frequent item sets based on FP-Tree. Journal of Software, 2005,16(2):215−222 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/16/215.htm

[4] Song YQ, Zhu YQ, Sun ZH, Chen G. An algorithm and its updating algorithm based on FP-tree for mining maximum frequent itemsets. Journal of Software, 2003,14(9):1586−1592 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/14/1586.htm

[5] Wang JY, Han JW, Pei J. Closet+: Searching for the best strategies for mining frequent closed itemsets. In: Getoor L, Senator TE, Domingos P, Faloutsos C, eds. Proc. of the SIGKDD 2003. Washington: ACM Press, 2003. 236−245..

[6] Grahne G, Zhu J. Efficiently using prefix-trees in mining frequent itemsets. In: Goethals B, Zaki MJ, eds. Proc. of the IEEE ICDM Workshop on Frequent Itemset Mining Implementations. Melbourne, 2003.

[7] Uno T, Kiyomi M, Arimura H. LCM ver.2: Efficient mining algorithms for frequent/closed/maximal itemsets. In: Jr. Bayardo RJ, Goethals B, Zaki MJ, eds. Proc. of the IEEE ICDM2004 Workshop FIMI 2004. Brighton, 2004.

附中文参考文献:

[1] 陆介平,杨明,孙志挥,鞠时光.快速挖掘全局最大频繁项目集.软件学报,2005,16(4):553–560. http://www.jos.org.cn/1000-9825/16/553.htm

[3] 颜跃进,李舟军,陈火旺.基于 FP-Tree 有效挖掘最大频繁项集.软件学报,2005,16(2):215–222. http://www.jos.org.cn/1000-9825/16/215.htm

[4] 宋余庆,朱玉全,孙志辉,陈耿.基于 FP-Tree 的最大频繁项集挖掘及其更新算法.软件学报,2003,14(9):1586–1592. http://www.jos.org.cn/1000-9825/14/1586.htm

陶利民(1973－),男,浙江杭州人,工程师,主要研究领域为数据挖掘,专家系统.

黄林鹏(1964－),男,博士,教授,博士生导师,CCF 高级会员,主要研究领域为程序设计语言,分布并行计算,人工智能.