频繁模式挖掘实习报告

1200012741 史杨勋惟（队长）

1200012826 姜双

综述：

本实验主要目的是对于频繁模式挖掘算法的深入理解和进一步探究。在此次实验中，两位组员分别使用了 apriori 算法，以及 fp-growth 算法进行挖掘。并在此基础上做了一定的优化和性能分析。本实验

报告主要有以下几个部分组成：

1, Apriori 算法的相关实验报告

2, Fp-growth 算法的相关实验报告

3, 组员的总结与感想

FP growth 算法的相关报告 —— By 史杨勋惟

1,主要思想概述

本算法主要通过了两种优化算法对朴素的 fp-growth 算法进行了时间与空间上的优化。其一为数据库的分解。由于需要被挖掘的数据库非常庞大，若将其全部读入内存后再在此内存上进行挖掘则需要巨大的内存开销，而对于数据库的分解可以很好的弥补这个缺陷。另一算法为基于被约束子树的 DCMine 算法，本算法主要参考了范明，李川在 2003 年发表的一篇论文，并在此基础上进行细节处理与具体实现，本算法主要优点在于无需递归构造条件模式的 FP 树，并通过有底向上更精简的搜索方式进一步优化了空间以及时间上的损耗。

2,相关数据结构与算法

在实习过程中，整个程序主要被数据抽象成为了两个数据结构，分别对应上述的两种算法，并提供了接口。所有代码均为亲自手写，无任何复制代码。下面简要的介绍一下这两种数据结构以及一些基本的实现方法。

ADT1：Data 类

主要功能：对原始数据库的扫描读取，删除对结果无碍的低频项，分解数据库。

主要实现方法：

首先，在第一次扫描过程中分别记录下每个项目的支持度，并以支持度作为 key 插入一个 map，由于 map 是已序的，插入完后的遍历中一旦发现低频项直接将其后的所有项删除即可。这样就得到一个合法项集以及各自对应的支持度。仍然以支持度作为 key，从大到小排序，生成一个序项转换表和项序转换表。与此同时对于每一个合法项集，生成一个其专属的空数据库数据，将这个项集成为此数据库的所有者。

第二次扫描中，对每一个项集，先删除其中的低频几，按照相序转换表转换后进行排序。然后将这个项集插入到此序列剩余项集中序号最大的项集所属的数据库。这样一来，就可以把整个大数据库处理并分解成为一些小的数据库了，由于把项集插入到支持度最小的合法项集所属的数据库，就可以保证每个数据库都不会特别大。

进而按照序从大到小在每个数据库上进行挖掘，挖一个数据库掘完后，对这个数据库的所有项集，先删除数据库的所有者，然后把剩余项集插入到某一个数据库中，完成一次更新。

将上述方法抽象到 Data 类中，进行有效的数据库处理，即可大大降低内存消耗。

ADT2：FPTree 类

主要功能：以 FPTree 作为基础模板，通过一些对结构上的细微调整，再借助一个 ST 的子结构，进而改进了 FPTree 上的数据挖掘，使得不需要递归建立较为复杂的条件模式树。

主要实现方法：

参考了范明的论文，在我看来，这个算法的核心在于 ST 子结构（论文中称之为被约束子树），对于任意一个频繁模式（a1,a2,...,ak),从底向上删除所有包含该模式的最短路径后，即将原树调整成一棵被该模式约束的 ST。进而在 ST 上任找一点，如果新的模式依然符合最小支持度，可将其加入该模式后继续搜索。从本质上来看，这依然是一个深度优先搜索的过程，但是相比于传统的 fp-growth,此算法免除了递归构造新树的时间，而只需维护几个基本的数组，以及一些频度计数即可。在时间和空间上都对原算法有了改进。

具体细节方面，在构造完 FP 树后，首先对指针进行调整，每个点的 ahead 指向父节点，每个点的 next 指向项头表中的同项下一个结点。此调整可以为以后从底向上的搜索提供便利。在此程序中，FP 树采用静态建树的方法，而子结构 ST 中的数组分别采用了 list 和 vector 两种不同的尝试（在后面会有对比），用一个 stack 维护每次搜索时的频繁模式向量。而输出的结果则保存在一个 map<int , vector<int> >中。以方便传递回 Data 类进行输出处理。

4,测试结果分析

测试环境如下：

OS：Ubuntu 13.10

CPU：Intel i5 2450

MEM：4G

Compiler：GCC 4.8

一共有五个不同的程序，分别为 typical， fast , sorted , undivided 和 list 版

	是否输出结果	结果是否排序	是否分解数据库	ST 数组数据类型
typical	是	否	是	vector
fast	否	否	是	vector
sorted	是	是	是	vector
undivided	是	否	否	vector

list	是	否	是	list

测试数据为 mushroom 的五个不同阈值， 25,20,15,10,5

对应的频繁项集输出数与合法项数为：

25：5545， 35

20：53583， 43

15：98575 , 48

10：574431 , 56

5：3755543， 73

时间消耗（s）：

	25	20	15	10	5
typical	1.13	1.57	2.01	5.3	28.35
fast	1.14	1.39	1.68	3.43	15.76
sorted	1.15	1.73	2.21	6.93	41.26
undivided	0.23	0.62	1.02	5.19	33.41
list	1.08	1.98	2.59	10.41	70.66

空间消耗（kb）：

	25	20	15	10	5
typical	3976	5520	5708	13840	60736
fast	3976	5520	5700	13840	60736
sorted	4068	7868	10892	48876	313296
undivided	3968	8508	12512	60068	384124
list	3980	5512	5708	13796	60572

分析：

（1）各版本对比与分析：

typical 和 fast 是几乎完全相同的两个版本，唯一的区别在于 fast 版本去掉了向文件输出的语句。由此可见,通过向最后文件输出答案的时间还是很长的，原因在于分解数据库后对每个数据库的输出都要进行文件流操作，这个操作个数远远大于传统 FP 树的一次输出。此时间的长度接近正比于最后输出的频繁模式个数。

typical 和 sorted 版本的对比可以看出对输出排序与不排序的影响，由此可见，最后输出量越大，排序所占的时间越长，（应该是与 nlgn 成正比的），同时，排序所消耗的空间接近于全空间（即不分解数据库），这是因为排序需要对所有输出进行内排序，损耗内存空间很大。

undivided 是不分解数据库的版本。由于不分解数据库，在空间上的大规模消耗事显而易见的，另一方面，由于不分解数据库，在时间上，输出时间大规模下降，但是当数据库一旦庞大，搜索量就会指数级增长，这就是为什么不分解数据库在小数据上特别快而在大数据上相对变慢的原因。

前面提到，ST 结构的数组可以用 list 来实现，理论上 list 的优点在于每次迭代时只需删除结点而不需复制维护，应该会有时间上的有点。但是事实证明，list 除了在大数据空间上略有优势外，效率完全占据下风，在时间上的效率更是大规模下降。我觉得这个原因和 list 本身的构造有关，动态结构在访问速度上会有明显的劣势。

（2）时空复杂度分析：

时间复杂度：除去输出时间，此算法的主要时间是在深度优先搜索上，而这个时间复杂度和每个数据库的 FPTree 的深度有关。时间复杂度应该与（平均深度×每个数据库平均项数）成比例。

空间复杂度：由于内存测试和空间复杂度的分析不能直接挂上相关的标签，但是除去一些固有内存，再加上对上述数据一定分析，我们可以认为此算法的空间复杂与最终的输出模式数应该成正比关系。而对于分解的数据库，此复杂度与合法项数也有关系。

5,更多优化空间

从上述测试结果来看，本程序的执行效率还是表现的比较均衡可观，但是仍然有进一步优化的空间，以下的方面是我有所思考的，但由于时间以及一些其他条件的局限性并未实际实现。

时间方面：通过对对每个步骤时间的测定，我发现本程序在时间上的主要开销在于挖掘和输出两个部分。在挖掘上的时间开销的减少需要更高级算法的支持，其实此挖掘算法（即 ST 挖掘法）依然有改进的空间，例如可以进行一定程度上的单路径优化，以及将 ST 中数组加深层次，进一步优化深度优先搜索的速度。而输出方面的优化则有一定的局限性，因为本程序优化的侧重点在于空间节省，这样就必然牵扯到一些外部访问，这对于输出时间的影响是比较大的，由此我们也可以发现时间和空间之间会有着一些相互的冲突，这也是一个计算机领域中比较基本的道理吧～

空间方面：有上述测试可知，如果对频繁模式的输出有序的要求，那么此程序依然需要相对较大的内存开销，由于需要排序的数据较为庞大，所以这一方面的内存节省唯一可以借助的工具应该是外部排序。这样就可以在同样不消耗太大内存的前提下输出符合要求的序列。另外，本算法采用的是一次分解数据库，也可以说是一级分解。其实，在我们挖掘某一个数据库的时候，可以把问题堪称一个频繁模式挖掘的子问题，所以我们还可以进一步分解这个数据库，即我们可以使用将数据库多级分解的方法进一步压缩内存开销。

2 Apriori 算法的相关报告——By 姜双

诚信保证：

// 我真诚地保证：

// 我自己独立地完成了整个程序从分析、设计到编码的所有工作。

// 在此，我感谢史杨勋惟对我的启发和帮助。下面的报告中，我还会具体地提到

// 他们在各个方法对我的帮助。

// 我的程序里中凡是引用到其他程序或文档之处，

// 例如教材、课堂笔记、网上的源代码以及其他参考书上的代码段，

// 我都已经在程序的注释里很清楚地注明了引用的出处。

// 我从未抄袭过别人的程序，也没有盗用别人的程序，

// 不管是修改式的抄袭还是原封不动的抄袭。

// 我编写这个程序，从来没有想过要去破坏或妨碍其他计算机系统的正常运转。

// 小组成员： 姜双

本次实习大作业我和史杨勋惟合作完成。史杨勋惟负责写 FP 树的程序，我主要负责 Apriori 算法的编写。

我理解 Apriori 算法的主要思想就是“由少生多”，即在满足最小支持度的情况下，先生成 1-项频繁集， 然后由 1 项频繁集生成 2 项候选集，再从中找出频繁模式，一直重复下去直到不再有新的频繁模式从而避免了对整个事务集的所有子集进行枚举。其原理是“一个事项集的支持度不可能比它的任何一个子集大”。

在实现过程中，对给出的测试数据，用 STL 中 vector<vector<int> >的方式来实现的二维数组存储原始事务集，每个事务为 1 个 vector，频繁项集和候选集采用同样的方式存储。用 map<vector<int>, int>的结构记录频繁模式的支持度计数。将测试数据按行读入（每行结尾是空格+回车）字符串再转为数字存储。

在进行所有挖掘操作之前，对每个事务集内的元素进行排序，以保证之后所有操作都可以不再排序。首先遍历所有事务求出 1-项频繁集。以此为基础生成多项候选集筛选频繁集。在生成候 k-项选集的时候，首先判断两个(k-1)-项频繁集是否只有最后一项不同，如果是则进行合并，这个判断过程采用对集合中的元素从后向前依次朴素匹配的方法。然后判断合并后的集合是否为频繁集，这里利用 STL 中 set 存储元素的有序性，调用 find（）函数对候选集的元素在事务集中匹配并计数，达到最小支持度要求的即 k-项频繁集。

我先按照原始的 Apriori 算法进行编写，在生成候选集、从候选集选择频繁集时都用朴素匹配的方式进行判断筛选，效率很低。全朴素匹配的方法下，需要的空间开销包括：存储读入的原始数据，和数据个数成正比，是 O（n）；存储频繁模式的集合、每次生成候选集时存储候选集的集合，这些也和数据个数成正比，同时和支持度下界有关，量级是 O（n）。总的空间复杂度即 O（n）。在时间开销上，最大的时间开销是判断一个集合是否为频繁项

集，这一步需要对所有原始数据进行一次扫描，然后对候选集中的元素在每个原始事务集中依次匹配，需要重循环，这算法的时间复杂度是 O（n^2）。还有频繁集合并为候选集的过程，所需时间复杂度也为 O（n^2）,对原始数据集排序的时间开销是 O（n*logn），总体来讲算法的时间复杂度为 O（n^2）。由于数据库很大，这个算法的时间消耗非常大。

对原始的 Apriori 算法，进行了一些简单的剪枝优化。生成 k-项候选集的时候，先将之前可合并的两个集合合并，但并不全部加入候选集，我们应该判断这个集合的所有(k-1)-项子集是否都是频繁集，只有都是，这个集合才可能是频繁集。这一步操作需要在上一轮的所有频繁集中去查找是否存在合并后集合的子集。如果采用原始的朴素模式，这个操作和判断集合是否为频繁的时间代价是几乎相同的，在大数据量时消耗非常大。对此做出的改进是：在上一轮频繁集中查找时使用二分查找的方式，由于最初对原始事务集的排序操作，使得每次生成的频繁集都是长度有短到长、每个集合里元素升序排列的。这样利用二分查找可以将查找的时间由 O（n^2）降至 O（n*logn），数据很大时产生了较好的省时效果。

最终我完成的 Apriori 算法版本中，除去注释和框架，实际代码长度约 300 行，能保证结果有较好的准确性，但是时间复杂度没有什么改观，耗时长，效率并不高。例如对本次测试的第三个事务集，运行时间仍然达到了四五个小时（=。=）

以下为各组测试数据的测试时间：

测试环境：32 位 Linux 下用 G++编译

1. mushroom.dat

   改进版：

阈值（%）	5	10	15	20	25
运行时间（s）	约 2 小时	1550.12	116.85	44.03	2.68

   原版：

阈值（%）	5	10	15	20	25
运行时间（s）	约 3 小时	3012.27	203.07	90.14	6.76

2. retail.dat

   这组数据只有阈值为 40%和 45%时有 2 个 1 项集和 1 个 2 项集，50%和 55%时有 1 个 1 项集，阈值更大时没有频繁模式，所有输出时间在计算精确度下均为 0.00s

3. T10I4D100K.dat

   对每个阈值运行时间都达到了 4 小时= =

总结与收获：

姜双：

这次实习让我好好体验了一次信息爆炸的感觉，大数据量下程序运行消耗大量时间。编写过程提高了我对数据结构的选择能力，选取恰当结构存储对时空效率和操作难易的影响非常大。改进和调试较大的程序也让我积累了很多调试代码的经验，如通过增加输出或注释掉部分程序来查看某些变量，对我以后的学习有很大帮助。和我合作的史杨勋惟同学在完成自己任务的同时还给了我很多关于调试程序、基本编程操作上的建议和帮助，让我学习到很多有用的东西。考虑到我的编程水平实在有限，史杨勋惟同学对我进行了多方面的指点，在此我对史杨勋惟表示深深的感谢。

这次实习在锻炼了我能力的同时也让我深深感受到自己的水平太低，有太多需要学习的东西，我会努力去提高自身的水平。

史杨勋惟：

与上次文本编辑器的实习相比，本次试验在代码量上有所减少，但是对于问题的分析以及理解方面却有了更高的要求。

就我个人而言，我觉得此次实习最大的收获就是在对于算法的复杂度的分析方面。虽然说算法分析是一门非常复杂的学问，以前经常自己会觉得根本无法分析清楚。通过这个实习，我发现了其实算法分析有一个最重要的最有用的方法就是亲自实践。通过对不同算法，不同结构的测试，我们可以很好的对它们进行理解。在此基础上再从理论上分析就可一对算法有更好的分析与此同时，我还掌握了一些最基本测试方法，如怎么在程序中查看本进程所占内存的峰值，以及怎么样通过 system 命令进行系统调用等等。

另一方面的收获在于文件操作，以前自己接触的对于文件操作的程序比较少，而这个实验需要大量的文件访问，尤其是在数据库的分解以及建立方面，如何更好地优化文件操作也是一个比较有意思的问题。通过这个实验我在这一方面的了解也有了加深。

通过本实习，我对于数据挖掘这一概念也有了深入的了解，数据挖掘无疑是当今计算机领域一个炙手可热的方向。以前对于这一方面的了解不是很深，通过本次实验，我对数据挖掘有了一个大致的了解，虽然只是写了一些这个方面最基本最简单的代码，但在写代码的过程中，我也对数据挖掘的一些思想，一些概念以及对问题的分析方法和对数据挖掘方法的基本设计都有了一些浅识。

当然还有就是在整个实习过程中通过与队友以及其他同学的相互讨论和交流过程中也了解了一些其他好想法，以及各式各样的挖掘方法，也让我开了眼界。