数算实习大作业报告

倪泽堃 (1200012747)　罗翔宇 (1200012779)　顾澄 (1200012882)

November 19, 2013

1　概况

2　分模块介绍

2.1　Apriori 算法

2.2　Apriori 算法改进

本模块由倪泽堃同学负责编写。暴力的 Apriori 算法并没有很显然的改进思路，因而选取了其变种 Eclat 算法进行改进。Eclat 算法对于每一个频繁项集记录包含它的交易编号，每次作扩展的时候，就求出两个项集的交易编号集合之交，判断其元素个数即可。但是首先的问题是对于 mushroom 这项数据与支持度 K=5%，最终结果就有几百万项，如果计算过程中直接用 vector 保存每项的交易编号集合，那么内存消耗极大，一般的 4G 内存电脑不能承受。因而我们需要寻求其他的方式，比如用一个 bitset 来记录交易编号集合，每位 0/1 表示该交易在/不在集合里。这样 32 个 bit 压成一个 4 字节 int 储存，内存消耗大大减小。每次直接对两个 bitset 做与运算并调用库函数 count 获取集合大小。这样最初的改进版本写成，但是对 mushroom 5% 这组数据运行时间达到了 27.9 秒（-O3 优化，不算最后排序输出，运行环境为 Intel Core i5-2430M CPU @ 2.40GHz×4、Arch Linux 64-bit，下同），实在是有点长，需要大力优化。由于其他两个数据风格不一样，在前 4 个优化中，就仅仅使用 mushroom 5% 这个数据。

优化 1：vector 改成静态数组　运用 gprof 对最初版本进行分析显示，运行时间的大部分消耗在 vector 的插入上。显然，vector 实在是太慢了。因此经过一番努力，我将求解部分的大部分用到 vector 的地方改用自己实现的静态 vector 存储，用一个巨大的内存池和一个标明可用空间起点的指针，每次直接计算出需要存储的序列，并将该序列写入内存池中，将指针后移。这样经过改进，所得的优化版本 1 的性能有很大提升。具体可见下表。

版本	优化方法	运行时间	bitset 内存使用	生成项集数	平均使用	项集废弃率
最初版本	-	27.9	-	-	-	
优化版本 1	使用静态数组	14.5	5082240280	5002205	1016	24.9%

优化 2：去除 bitset 中连续 0　显然，使用 bitset 存储之后，还有可以改进之处：k 越大，频繁 k-项集的支持度计数就越小，此时 bitset 中将有许多的连续 0。我们对此作出改进：在 32 位压成一个 int 存储的 bitset 中，将等于 0 的 int 从中去除，用二元组记录不为 0 的元素的下标与值。尽管由一元变成了二元组，作出改进后的优化版本 2 中内存使用仍然减少了，而且时间效率有极大的上升。具体可见下表。

版本	优化方法	运行时间	bitset 内存使用	生成项集数	平均使用	项集废弃率
优化版本 1	使用静态数组	14.5	5082240280	5002205	1016	24.9%
优化版本 2	去除连续 0	7.1	3460183168	5002205	692	24.9%

优化 3：重标号、排序　压缩连续 0 的效果之所以不理想，可以想像，是由于编号过于分散，存储的数据不够紧凑。因此需要对交易重新排序，使得内容相近的交易处在尽量近的位置。这时，可以发现，将含有最频繁的项的交易排在一起，再将含有第二频繁的项的交易排在一起，以此类推，所得的交易序列相邻项相似度非常高，是比较优的。因此先将项按频繁程度重新标号，将交易排序，然后进行求解。加上该优化所得的优化版本 3 效率见下表。

版本	优化方法	运行时间	bitset 内存使用	生成项集数	平均使用	项集废弃率
优化版本 2	去除连续 0	7.1	3460183168	5002205	692	24.9%
优化版本 3	重标号、排序	6.1	2791211776	12782139	218	70.6%

优化 4：频繁序列降序存储 令人稍有惊讶的是，优化版本 3 的优化幅度不甚理想。从表中可以很显然地看到，虽然数据存储更加紧凑了，但是该算法多增长出了大量支持度小于给定阈值的候选项集。经过分析可以发现，该算法由序列 $\{a_0, a_1, ..., a_{n-1}\}$ 与 $\{a_0, a_1, ..., a_{n-2}, a_n\}$ 增长出 $\{a_0, a_1, ..., a_n\}$, 但是由于 a_{n-1}, a_n 比较罕见，增长出的序列支持度可能会剧烈下降。相反，如果从 $\{a_1, ..., a_n\}$ 与 $\{a_0, a_2, ..., a_n\}$ 增长出目标序列，支持度下降幅度则一般不会很大。因此改用降序存储频繁项集，经测试，所得的优化版本 4 效率有极大的提升，如下表所示。

版本	优化方法	运行时间	bitset 内存使用	生成项集数	平均使用	项集废弃率
优化版本 3	重标号、排序	6.1	2791211776	12782139	218	70.6%
优化版本 4	降序增长	2.0	978700888	3780460	259	0.7%

在优化版本 4 中，废弃率已经降低到惊人的程度，数据的存储也很紧凑，运行时间降低到了 2 秒，而之后的排序就要 1.6 秒，可以说效率已经相当高了。将其他两组数据也纳入统计，可以对刚才的所有优化做个总结。

版本	优化方法	mushroom 5%	T10I4D100K 0.1%	retail 0.1%
最初版本	-	27.9	-	-
优化版本 1	使用静态数组	14.5	16.6	123.5
优化版本 2	去除连续 0	7.1	3.6	7.5
优化版本 3	重标号、排序	6.1	2.2	6.8
优化版本 4	降序增长	2.0	2.0	4.4

优化 5：预处理频繁 2 项集 我们发现，T10I4D100K 0.1% 这组数据答案才 27532，但是却花了 2.0 秒才计算出答案。这是因为计算频繁 2 项集时，多扩展了许多支持度很低的项集。因而可以使用一个很简单的方法，就是预处理出频繁 2 项集。优化后的版本 5 效率如下表所示。

版本	优化方法	mushroom 5%	T10I4D100K 0.1%	retail 0.1%
优化版本 4	降序增长	2.0	2.0	4.4
优化版本 5	预处理 2 项集	2.0	0.8	1.0

优化 6：部分基数排序 此时我们的程序已经非常优了。不过如果最后进行排序输出，效率还是比较堪忧的，其中排序就要花费比较长的时间。观察到我们是对序列排序，而每一项并不大。因而在序列长度较长的时候，可以先用基数排序，当每个桶的元素个数小于一定数的时候再用快排排序。经过改进后排序效率有所增加，如下表所示，其中运行时间加上了排序的时间，但不包括输出时间。

版本	优化方法	mushroom 5%	T10I4D100K 0.1%	retail 0.1%
优化版本 5	预处理 2 项集	3.6	0.8	1.0
优化版本 6	部分基数排序	3.0	0.8	1.0

值得注意的是，将输出时间加上，对于 mushroom 5% 这组数据，程序总共运行时间达到了 7s，这说明效率主要瓶颈在输出了。也就是说现在的优化版本 6 已经是非常优了。因此，可以说优化已经基本完成。

2.3 FP 算法

2.4 FP 算法改进

3 心得体会