**函数依赖挖掘实验报告**

软件51 谢运帷 2015013185

软件52 石耕源 2015013219

代码已托管在github库上：<https://github.com/shi-gy15/FDMine-TANE>

1. 实验环境

本次实验采用要求的C++进行编码，所以我们采用Visual Studio 2017进行编写、编译和生成可执行文件。

操作系统：64位win10。

CPU：i7-4720HQ

内存：8GB

1. 编程语言

按照要求使用C++

1. 实验算法

通过对比课上7种算法的挖掘时间，我们一致同意使用TANE作为我们的挖掘算法。相比之下，我们本次的实验数据只有15列，列数较少，使用TANE这种枚举列属性的组合比较好；而它与其他的使用TANE的结构枚举属性的算法相比，在列数为100000时表现相差不大，而算法更加稳定，实现比较直接。

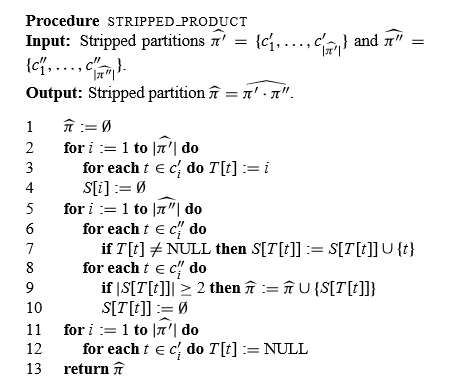
通过学习TANE算法的论文，我们抓住它的核心算法的过程，有针对性的实现了存储诸如属性集、C+(RHS+)，算法节点层。首先我们先来介绍一下TANE除去课上讨论过的部分的数据结构和算法。

我们可以知道，判断一个函数依赖的方法为：

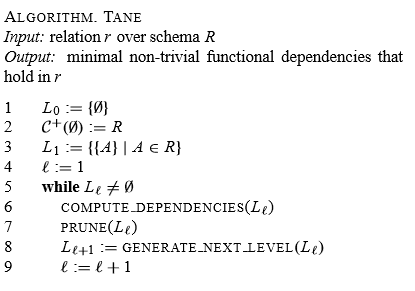
但实际上，我们不需要存储中所有的等价类，就可以计算。我们可以除去中的只有一条记录的等价类（也就是说，这条记录在数据集中没有对应属性都其他相同的记录了），这样存储的更小，而同样可以算出这个包含所有的等价类的划分。论文中将这种新的等价类的存储方式称为：Stripped Partition。

TANE是一个使用按层计算函数依赖的方法，新的一层可以由之前的一层生成出来，新的一层的partition也可以由生成它的两个父亲来计算出来：

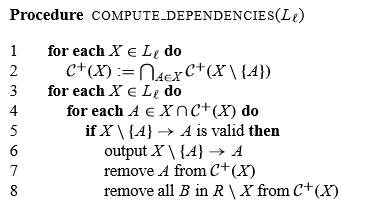
而使用了Stripped Partition之后，下一层Stripped Partition也可以通过上一层的两个Stripped Partition计算出来。具体算法在论文中以如下方式给出：



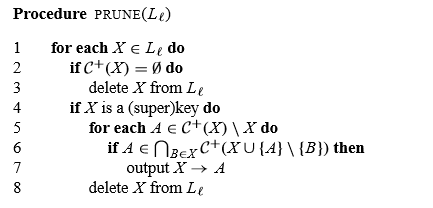
以上就是课上未讨论的部分，以下我们展示一下论文中的**核心算法**：



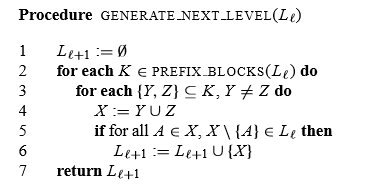
TANE的核心算法基本处于按层遍历的过程，所以我们可以抽象出一个层Layer的类。



Compute\_dependency是在遍历一层的所有属性集X，不断地遍历的属性，判断是不是。具体判断的方法已经在前文给出，即，其中为Stripped Partition。



这里prune的剪枝算法比较直接，直接从一层删掉为空的集合X。还有删掉超键的剪枝，这里需要去查询，注意而这个是很有可能不属于当前层的，这就告诉我们不能存储在层Layer这个类中，需要单独存储，每个和一个属性集相对应。



生成下一层的方法中提示了我们一个属性集该如何去定义，这里告诉我们需要判断两个属性集是不是属于同一个prefix\_blocks。Prefix\_blocks的定义在论文中给出：两个属性集a,b属于同一prefix\_blocks当且仅当a和b拥有前个相同的属性，只有最后一个属性不一样。这里属性有一个编号，在属性集中按编号从小到大排列。结合属性集有很多求交，求差，求并的需求，我们认为可以通过一个unsigned int（32位）来表达一个属性集，如果对应位为1则代表拥有这个属性，反之则没有，这样可以简单地通过位操作来实现集合操作，而且也很容易定义prefix\_blocks。

经过以上对于TANE算法地分析，我们已经分析出了整个实现的方法：

1. 属性集对应为一个unsigned int
2. 一个Stripped Partition为一个vector<vector<int>>，直接地表达了一个划分
3. TANE\_Node表达一个层中一个结点，它有一个属性集，一个划分
4. TANE\_Layer表达一层，它有一个map，键为属性集，值为一个TANE\_Node，是对于TANE搜索的直接表达。它可以根据这个键（属性集）进行排列，遍历时可以保证一个prefix\_blocks中的键相邻。
5. 额外一个unordered\_map来记录，键值为属性集，值也为属性集（）。

随后按照之前展示的TANE论文中的算法来实现，再细节的实现可以参见我们的代码。

1. 代码构建、优化过程

实验中我们编写了面向对象的算法实现，主要的类由下表所示。

|  |  |  |
| --- | --- | --- |
| 类名 | 头文件 | 作用 |
| AttributeSet | AttributeSet.h | 属性集 |
| ECSet | DisjointSet.h | 单个属性集的等价类划分 |
| TANE\_Node | TANE-tree.h | TANE算法中的结点 |
| TANE\_Layer | TANE-tree.h | TANE算法中的层 |
| FD | TANE-tree.h | 单个函数依赖关系 |
| Database | Database.h | 数据表 |

同时Solver类负责该算法的运行步骤。下面主要描述优化过程。

1. 读取数据库表

数据库表的存储方式是vector<vector<string>>，这是一种朴素的想法。从文件中载入该表后，对每一列求partition的过程是各自独立的，因此我们采用了多线程并行执行进行优化，即每一列拥有单独的map。结果表明线程数3~4能使效率提高较多。

我们也尝试不保存数据库表，在读入时直接将其插入到partition中，但这样效率提升并不明显。同时，在读取表的过程中可以在O(1)时间内求出其等价类个数。

2. 属性集表示

AttributeSet类中属性集合用一个整型数表示，每个属性占该整数的一位。这使得很多对属性集的运算可以转化为位运算，例如求两个属性集的并集可转化为两个整型数的位或操作，这使得属性集运算也达到了近似O(1)的时间。

3. Product运算

对程序执行应用Visual Studio的性能探查器工具发现partition的product运算占据了整个程序接近80%的CPU，因此对这个函数的优化产生的影响较大。我们通过对TANE算法论文的学习发现，计算product所需的两个辅助数组S与T每次求值后不需要全部清空，因此将其设为全局的，使效率有较大提升。

此外，还为该运算实现了短路操作。我们知道一个属性集是超键(super key)等价于其等价类个数与表的列数相等，并且它与任意属性集的并集也必然是超键。那么在求product之前就可以检查读表时求得的等价类个数，如果发现这个属性集是超键，就发生短路，不执行product运算，并令其集合为空即可（因为再也不需要求这个集合），然后进行后续剪枝。

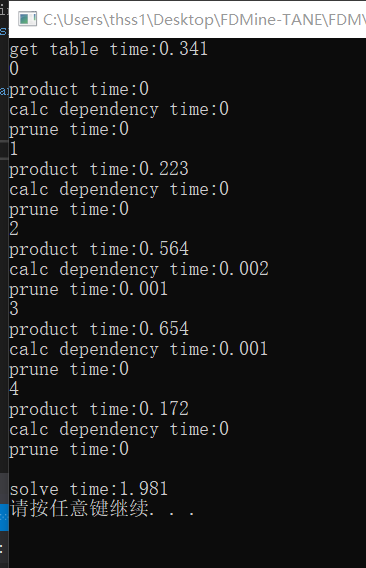
4. 使用unordered\_map

算法中的RHS+集合存储的map不要求顺序关系，因此我们将查找这个集合的map改为unordered\_map，散列函数为对大质数取模，这样从理论上达到近似的O(1)时间查找，实际上也使效率有一定进步。

1. 代码运行时间

我们编译好的程序在之前给出的实验环境上运行，多次重复运行消耗时间如下：

|  |  |
| --- | --- |
| 运行次数n | 消耗时间t |
| 1 | 1.981 |
| 2 | 1.982 |
| 3 | 2.006 |
| 4 | 1.992 |
| 5 | 2.026 |
| 平均时间 | 1.9974 |

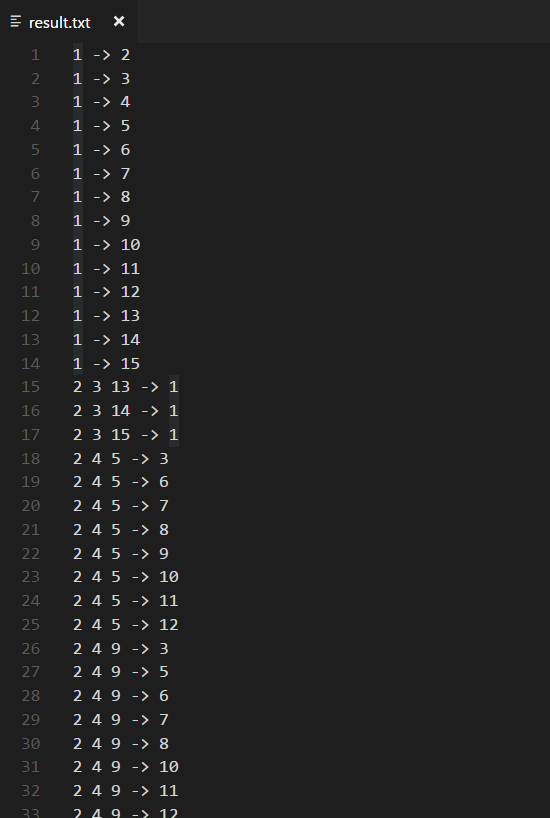


在数据集上的平均运行时间为：1.9974秒。具体分成每一步可以发现，时间主要花费在求product上，其次是读表格的时间，其他的部分几乎没有消耗时间。整个TANE求解过程时间消耗已经相对比较低了。

1. 实验结果分析

最后我们对于输出的result.txt进行分析：

整个result.txt共有518条函数依赖，按照要求递增排列。



首先可以发现右侧的属性都是只有一个，而且没有trival的函数依赖（即右侧的属性没有包含在左侧的属性集之中的），而且~~目测~~没有非minimal的依赖。单从结果来看，我们可以确认没有错误的结果。结合论文中对于TANE算法找到依赖的正确性和完整性，我们可以确认实现的函数依赖挖掘算法的正确性。