数据库函数依赖挖掘作业

2015013212　　2015013222

候建国　　陈超

一. 实验内容：

实现课上学习的一种函数依赖挖掘算法。

二. 数据集：

数据集包括 15 个属性名，共 10 万条以上数据。

三. 实验环境:

开发测试环境：

- 操作系统：Windows 10
- IDE: Visual Studio 2012/2015
- 编程语言：C++

四. 实验过程

1. 算法的选定：本次实验我们综合了算法的难易程度、效率高低、以及小组同学对算法的了解最终选定了 TANE 算法，该算法不仅仅在效率上比较快，同时也适合更大的数据集。 但之后我们接着尝试了 DFD 算法。但是由于时间问题没来及的完成。因此这里仅说明 TANE 算法。

2. 算法的理解：TANE 算法的目的是找到所有有效的最小非平凡的函数依赖。它对数据几何组成的格子结构进行分层，然后对每一层计算函数依赖，计算完之后对该层进行剪枝，接着更新到下一层。对于计算函数依赖的具体步骤，显示如下:

Procedure COMPUTE_DEPENDENCIES(L_ℓ)

1 **for each** $X \in L_\ell$ **do**
2 $C^+(X) := \bigcap_{A \in X} C^+(X \setminus \{A\})$
3 **for each** $X \in L_\ell$ **do**
4 **for each** $A \in X \cap C^+(X)$ **do**
5 **if** $X \setminus \{A\} \rightarrow A$ is valid **then**
6 output $X \setminus \{A\} \rightarrow A$
7 remove A from $C^+(X)$
8 remove all B in $R \setminus X$ from $C^+(X)$

该函数通过上一层的LHS^+找到最小函数依赖。这里 2 到 5 行保证了程序的输出一定是最小的函数依赖。其中第 5 行的完整性检验利用了如下定理：

LEMMA 3.5. *A functional dependency* $X \rightarrow A$ *holds if and only if* $e(X) = e(X \cup \{A\})$.

3. 算法的思想：

（1）数据集存储：使用一个 vector<vector<int>>（FT_table）的 table 存储数据集。

（2）属性的存储：我们使用二进制的方式来存储属性，比如 1 存储属性 1,10 存储属性 2,100 存储属性 3，以此类推；而属性组合就可以通过二进制计算来得到，比如 25 组合属性可以通过 10+10000=10010 来表示，这样做可以不必使用数组来存储每个属性而减少空间浪费，同时还有利于组合属性的计算。

（3）每层的属性存储：用 set<int>来存储一层的属性，vector<set<int>>（FT_levels）存储格子结构的每层属性。

（4）RHS+存储：使用一个2^n的数组 vector<int>（FT_RHS）来存储。

（5）π 存储：对于每个属性，使用 vector<vector<int>>来存储该属性不同分区的实例行，vector<vector<vector<int>>>（FT_partitions）将所有属性包含在内；使用 vector<int>（FT_exists）来保存每个属性的分区信息是否已计算过，计算过的便不用再次计算；使用 vector<int>（FT_piLen）来存储每个属性的分区行的总长度。

4. 算法的主要实现：

在算法的实现过程中，我们参考了相关论文

TANE: An Efficient Algorithm for Discovering Functional and Approximate Dependencies

中的算法的实现步骤。

我们构建了一个 FDTANE 类，用于数据导入导出以及算法的具体实现。

初始情况下，我们调用该类的构造函数 FDTANE(in, out, n)读入数据，构建 FT_table，然后初始化每个存储结构，并初始化格子层状结构的第一层的 FT_levels 和 FT_RHS。

本程序中的主要函数包括：

generateFD()：逐层计算，得到最小函数依赖并输出，同时计时以观察效率。

generateNextLevel(int level)：计算 level 层的属性集。

computeDependencies(int level, ostream& outputstream)：计算 level 层

的函数依赖项并输出。

五. 实验结果分析

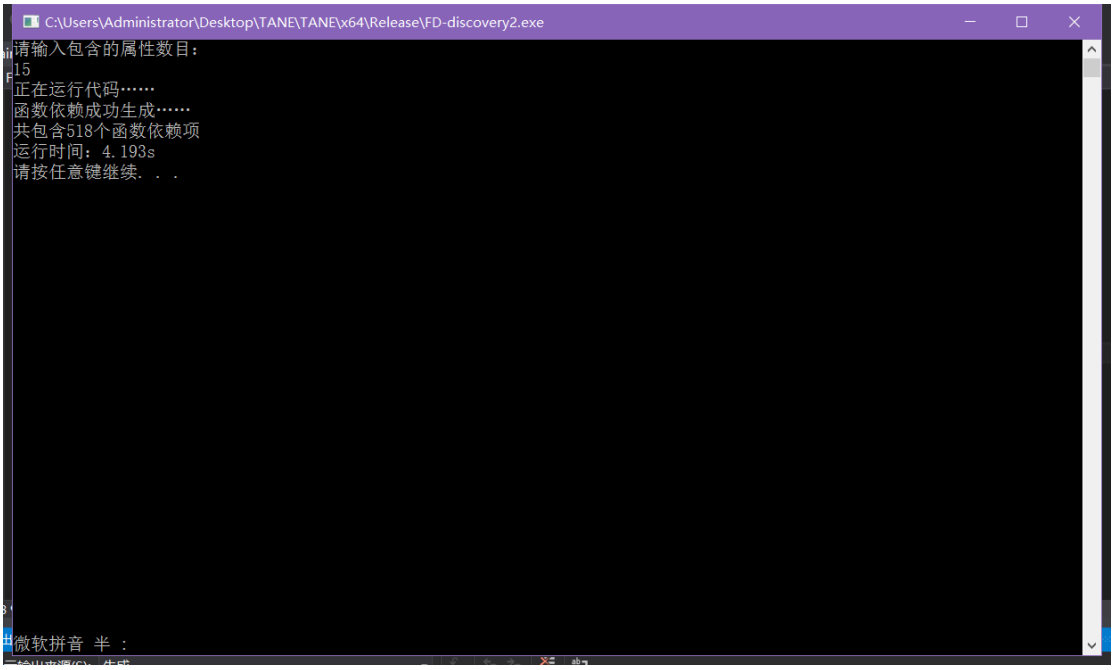
我们使用测试数据 test_data.txt 与 data.txt 进行结果分析

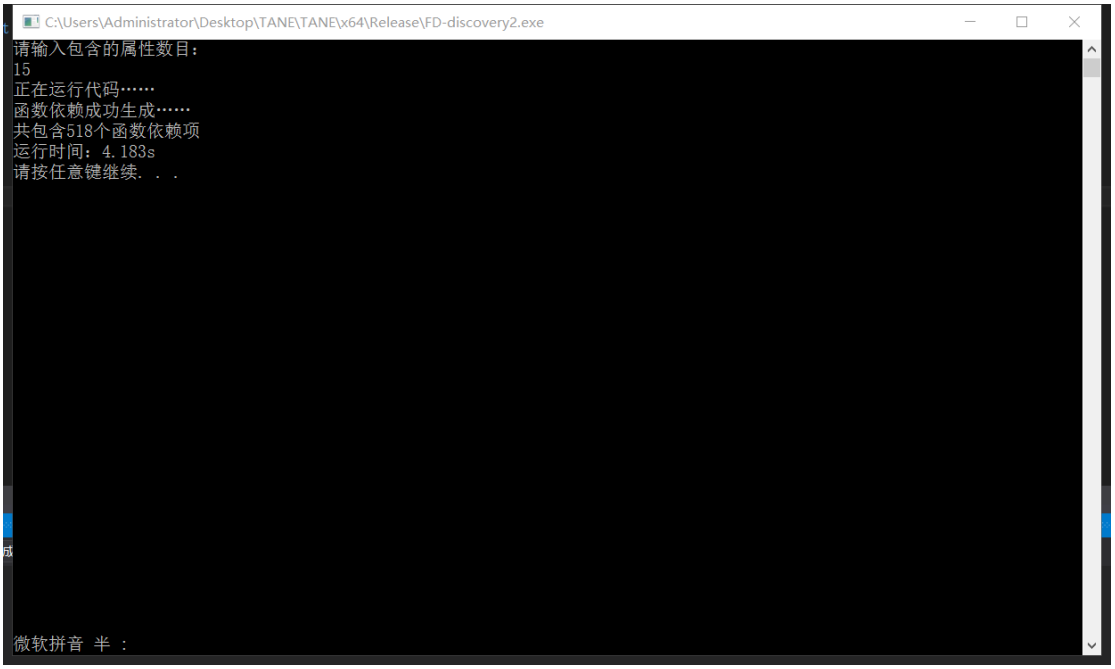
1. test_data.txt（对应于 test_output_data.txt）

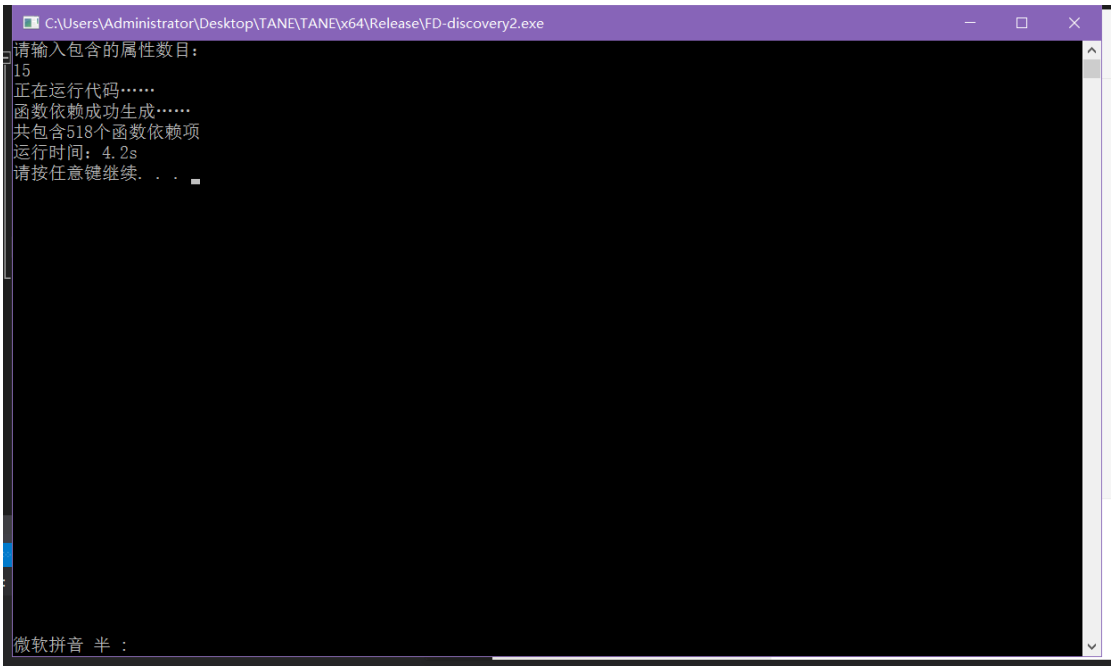
对于测试数据集，我们主要侧重于结果的正确性，经过与助教提供的结果比对，我们的程序结果是正确的。对于该数据集，总共产生了 109 个函数依赖项，平均用时为 0s.

2. data.txt（对应于 output_data.txt）

对于大数据集，我们主要侧重于得出结果的效率。该数据集总共产生了 518 个函数依赖项。我们总共进行了 3 次测试，最终取平均得到平均花费时间为 4.192s，以下为截图：







因此，对于 10 万+的数据量的处理，该算法还是呈现出较好的计算效率。

但是在用 Visual Studio 运行程序时发现在不同电脑上运行时间不同，但呈现出结果较好。

六. 关于 DFD

在之后的工作当中，我们发现 DFD 算法是一个比较新兴的算法，而且通过论文的阅读，可以看出 DFD 是要比传统的 TANE 算法效率高。因此我们尝试再实现一个 DFD 算法。

通过论文的阅读，大致得到了如下思路：

先将属性集合中的主键剔除，接着对剩下的属性集合(RHS)遍历，得到 LHSs. 从而得到最终结果。我们实现的算法的主要函数包括：

void DFD::generateFD() 计算函数依赖的整体函数

void DFD::findLHS(int attr) 寻找当前属性 attr 的所有 LHSs，先生成种子，对种子中的某一个 node 判断或更新其类型,并得到下一个 node 继续执行

int DFD::pickNextNode(int attr, int node) 根据当前 node 寻找下一个 node

vector<int> DFD::generateNextSeeds() 通过 mindeps 和 maxnondeps 生成下一批种子，继续在函数 findLHS 中执行

我们的代码尚不能运行，因为时间问题其中的 bug 来不及调试。但函数的基本结构框架和具体功能已实现，因此我们将 DFD 的代码一并提交，并在之后的时间里，我们会继续将其完善。