函数依赖挖掘实验报告

2015013157 王景隆
2015013229 陈书新

1 实验环境

1.1 本机环境

- OS: macOS High Sierra 10.13.4
- CPU: Intel Core i7 2.5GHz
- Memory: 16GB DDR3 1600MHz

1.2 编辑/编译环境

- g++: Apple LLVM version 8.0.0 (clang-800.0.38)
- IDE: Version 8.0 (8A218a)

2 实验结果

2.1 使用说明

源代码目录如下：

```
FDMining/
    testdata/
        data_small.txt
        data_large.txt
        fd_small.txt
        fd_large.txt
        fd_small_comparison.txt
        fd_large_comparison.txt
    BruteForce.h
    BruteForce.cpp
    Tane.h
    Tane.cpp
    TaneOptimized.h
    TaneOptimized.cpp
    Makefile
```

首先运行 `make` 进行编译，若无法使用，也可以手动进行编译：

```
g++ -o FD TaneOptimized.cpp Tane.cpp main.cpp -std=c++11 -O3
```

编译完成后在 `FDMining/` 目录下得到 `FD` 可执行文件，通过命令：

```
./FD <dataset>
```

运行，其中 `<dataset>` 可以是 `small` 或者 `large`，分别对应小数据和大数据。程序会从 `testdata/data_<dataset>.txt` 读入数据，输出结果到 `testdata/fd_<dataset>.txt`，并在控制台中输出每一部分的运行时间。

2.2 实验简述

本次试验中实现了三种算法，分别对应 `BruteForce.h/cpp`，`Tane.h/cpp` 和 `TaneOptimized.h/cpp`。其中 `BruteForce` 是没有任何优化的暴力，仅用来跑出大数据的结果并验证后续需要实现算法的正确性。跑出的结果存放在 `testdata/fd_<dataset>_comparison.txt` 中。`Tane` 是按照论文复现的算法。`TaneOptimized` 是对读入和原论文的某些实现进一步优化的算法。

2.3 运行时间

算法	小数据	大数据
BruteForce	< 0.1s	900s
Tane	< 0.1s	2.3-2.5s
TaneOptmized	< 0.1s	1.8-1.9s

所有算法均在小数据中得到 109 对函数依赖，大数据中得到 518 对函数依赖。经过 `diff` 输出文件对比后，它们的结果均相同。

`./FD <dataset>` 运行时调用 `TaneOptimized` 算法。

3 算法及优化

3.1 Tane 简述

由于课上已经详细讲过 `Tane` 算法了，所以这里只提一点：`Tane` 是一个基于 BFS 的算法，按照层次优先进行搜索，第 `i` 层表示大小为 `i` 的集合。在 `Tane` 的论文中，作者详细给出了算法主框架和每一个函数的框架，方便读者进行复现。然而在这次实验中，我们发现了论文中的一个错误和若干个仍然可以进行优化的地方。

3.2 Prune 中的错误

论文中，作者给出了用来删去无用集合的 `Prune` 函数，伪代码如下：

```
Procedure Prune(L_l)

foreach X in L_l do
    if C+(X) = emptyset do
        delete X from L_l
    if X is a (super)key do
        foreach A in C+(X)\X do
            if A in intersect(B in X, C+(X+{A}\{B})) then
                output X -> A
        delete X from L_l
```

其中 `C+(X)` 即为 `RHS+(X)` 。第一个 `if` 的剪枝中，删去了所有 `C+(X)` 为空的集合，这是显然的，因为如果当前集合的 `C+(X)` 为空， `RHS+(X)` 的计算方式是取所有子集的并，那么它在后续层直接 / 间接相连的集合（即超集）的 `RHS+(X)` 都为空，因此这个集合是可以删除的。

但第二个 `if` 的剪枝是错误的。在这个 `if` 中，如果 `X` 是一个 `key` ，那么就会提前输出 `X` 作为左边时的函数依赖，然后把 `X` 删除。但真的能删除 `key` 吗？由于函数依赖的最小性质，这个 `X` 以后确实不可能以子集的形式出现在函数依赖的左边，但它可能会出现在右边（如果 `X` 只包含一个元素时）。由于删除 `X` 意味着删除了所有 `X` 的超集，那么以后 `X` 永远不会出现在函数依赖的右边了。举个例子：

A	B	C
a	1	2
b	1	3
c	2	3

如上数据表中， `A` 是一个 `key` ，有函数依赖 `A->B` 和 `A->C` ，但同时也有 `BC->A` 。如果使用了第二个 `if` 的剪枝，那么 `BC->A` 这个依赖就永远搜索不到了。小数据集中并没有 `key` ，因此加上这个剪枝后，结果并不会错误，但大数据集中有 `key` （第一个属性），因此就体现出了第二个剪枝的错误性。

3.3 读入优化

读入优化不算对于算法本身的优化，这里就简要说一下。

在实现 `Tane` 时，我们使用了 `std::ifstream` 作为文件流，使用 `std::getline()` 行读入，整个读入时间约为 0.2-0.3s。在实现 `TaneOptimized` 时，我们使用了 `FILE*` 作为文件流，使用 `fgets()` 行读入，整个读入时间小于 0.1s，快了近一倍。

由于本实验的数据量很少，就算大数据也只有几十 MB，因此可以首先使用 `fread()` 将所有数据读入内

存，再把内存作为流来行读入，但这种写法会使得代码显得很难看，而且优化空间也不是很大，就算再快一倍，也只剩下了 0.05s 的时间，还不如去优化计算 `Partition` 的部分。

3.4 集合的存储及推导优化

算法中有很多地方需要用到集合以及集合的并、交、差操作。我们可以使用 `std::bitset<>` 来表示一个集合，第 `i` 个二进制位 1 表示集合中有第 `i` 个属性。由于大数据也只有 15 列，所以用一个 `int` 就能表示一个集合。设两个集合 `A` 和 `B`，对应的操作如下：

- 交操作：A & B
- 并操作：A | B
- 差操作：A & (~B)

这样所有用到的集合操作都可以在 `O(1)` 的时间完成。

我们有时需要知道集合 A 中有哪些位置是 1，可能会这样写：

```
for (int pos = 0; pos < MAX_POS; ++pos)
{
    if (A & (1 << pos))
    {
        // do something
    }
}
```

这样复杂度是 `O(|S|)` 的，但其实可以这样写：

```
int lowbit(x) {return x & (-x);}

while (A)
{
    int pos = lowbit(A);
    A -= pos;
}
```

`lowbit(A)` 会取出 `A` 最右边的那个 1 并返回，复杂度是 `O(c1)`，其中 `c1` 表示 `A` 的二进制表示中 1 的个数。对于一个集合来说，`O(c1)` 和 `O(|S|)` 并没有任何区别，但对于所有集合来说，前者的总复杂度是 `O(|S|*2^(|S|-1))`，后者的总复杂度是 `O(|S|*2^|S|)`。

这种写法中，获得的 `pos` 在数值上实际上是 2 的 `pos` 次方，看起来我们还需要获得真正的 `pos`，还需要很麻烦的取 `log` 操作。但实际上，我们只有在输出的时候才需要知道真正的 `pos` 是多少，大部分时候我们想要知道集合 `A` 中有哪些位置是 1 时，是想要求出 `A` 的所有大小为 `|A| - 1` 的子集。此时，每个 `A - pos` 就是一个对应的子集。

最后一个需要提到的点是论文中给出的 `Generate_Next_Level` 函数，通过第 `L` 层合法的集合来产生

第 `L+1` 层合法的集合，给出的伪代码如下：

```
Procedure Generate_Next_Level(L_l)

L_{l+1} = emptyset
foreach K in Prefix_Blocks(L_l) do
    foreach {Y, Z} in K, Y neq K do
        X = union(Y, Z)
        if forall A in X, X\{A} in L_l then
            L_{l+1} = union(L_{l+1}, {X})
return L_{l+1}
```

这个算法有些不明所以。事实上，我们可以直接预处理出所有第 `L=1,2,...` 层的集合，即二进制表示中有 `L` 个 1 的那些数。统计 `A` 的二进制表示中 1 的个数，可以用之前给出的枚举或者 `lowbit()` 的两个方法，也可以用 `c++` 自带的 `__builtin_popcount()` 函数。我们将所有的 `A` 按照第一关键字 1 的个数，第二关键字本身的大小排序，就预处理出了所有第 `L=1,2,...` 层的集合，并且每一层都是从小到大给出的。

在 Tane 算法中，第 `L` 层的有些集合在剪枝时被删除，因此第 `L+1` 层并不是每个集合都是符合条件的。我们可以用上面提到的方法遍历这个集合的每一个大小为 `L` 的子集，复杂度为 `O(cl)`，只有当所有的子集都没被删除时，这个集合才是符合条件的。

3.5 Partition 计算优化

论文中使用二维向量存储 `Partition`，对应在 `c++` 中即为 `std::vector<std::vector<int>>`，对比一维数组的存放方式，`f[i] = j` 表示第 `i` 行数据等价类 `j`，这样的好处是节省了空间，因为大小为 1 的等价类并不需要存储。计算 `Partition` 需要当前集合的两个大小为 `L` 的子集 `A` 和 `B`，过程分为三步：

- 将 `A` 的等价类还原成一维数组的存放方式
- 遍历 `B` 的等价类，得出 `union(A, B)` 的等价类
- 删去所有大小为 1 的等价类

要想优化其中的任意一步都是十分困难的，最多只能巧妙的利用引用加快寻址进行常数级别的优化。但我们考虑到第一步的结果是可以多次利用的，如果两个不同的集合使用相同的子集 `A` 进行等价类计算，那第一步的结果是可以保存的。此时，我们在 3.4 中提到的预处理就起到了作用。由于同一层的集合都是从小到大给出的，因此相邻两个集合它们有很大的概率拥有相同的高位。取某个集合的子集 `A` 时，我们尽量通过删去低位来获得 `A`，这样相邻的集合获得的 `A` 就很可能相同。例如当 `L=3`，列数为 10 时，`L+1` 层的相邻集合可能如下：

```
...
1001100001 -> A = 1001100000
1001100010 -> A = 1001100000
1001100100 -> A = 1001100000
1001101000 -> A = 1001100000
1001110000 -> A = 1001100000
1010000011 -> A = 1010000010
...
```

前五个集合获得的 `A` 相同，均为 `1001100000`，我们在计算这五个集合的等价类时，就可以可以跳过第一步，直接进行第二步和第三步操作。根据实验，加上此优化后，计算所有函数依赖的总时间（不包括读入的时间）减少了约 28%。

3.6 其它可能的优化

在 `TaneOptimized` 算法中，我们均实现了以上提到的优化。当然还有一些潜在的优化，这些优化大多和算法本身无关，例如多线程优化、针对数据本身的特点进行的优化（例如大数据中函数依赖的 `LHS` 的大小最多为 4）。这些优化我们感觉比较无聊，例如多线程和电脑本身的配置（CPU 内核数量）相关，针对数据本身的特点进行优化会使得算法不具有普遍性，因此我们并没有实现这些优化。

4 实验总结

本来是打算实现一下 `DFD` 算法的，但那篇论文写的实在是太烂了，很多地方都没有给出证明，算法主框架还被我挑出错了，还不知道这算法的正确性（可能大体上的正确的，但是细节没有完善），感觉作者没有认真对待这篇论文啊。`Tane` 的论文就比 `DFD` 早了 5 年，引用量有 500+，而 `DFD` 还没到 100，而且网上几乎找不到 `DFD` 的实现代码，可能已经高下立判了吧。