高级程序设计II 大实验作业三

项目地址：Github

本次作业分工：

农钧翔 2019201407：生成关联规则以及置信度

于倬浩 2019201409：`Apriori`算法生成频繁项集和非频繁项集

汪元森 2019201420：`calcu_support`算法计算项集的支持度

实验环境

操作系统：Ubuntu 20.04 LTS

```
$ clang --version
clang version 11.0.0 (https://github.com/llvm/llvm-project.git
ca376782ff8649d1a5405123f06a742e0e94b701)
Target: x86_64-unknown-linux-gnu
Thread model: posix
```

算法实现

`calcu_support`函数的实现

1. 问题的抽象化
   首先希望实现一个 `calcu_support()` 函数，用于计算某个项集 itemset 的支持度。而计算项集支持度的核心在于判断该项集是否被某个给定的 transaction 所支持。

   由支持度的定义，我们可以将该判断过程抽象为如下图论模型:
   若某个 itemset 满足:

1. 在 transaction 的数据依赖共享图中，包含了该 itemset 的所有元素，且每个元素至少与一个 itemset 中的其他元素相邻。
2. 在 transaction 的控制流图中，存在一条链，使得该链包含了该 itemset 的所有元素。

   则称该 transaction 支持该 itemset。

2. 算法设计

   我们约定 itemset 中的元素所对应的图中的节点为关键点。

   对于上述判断条件 1，首先依次遍历数据依赖共享图的节点，判断是否全部包含；然后对于每一个关键点，遍历它的前驱 / 后继，以判断是否有相邻关键点。

   对于上述判断条件 2，首先将控制流图缩点，得到强连通分量构成的有向无环图 (DAG)；然后从入度为 0 的节点开始 DFS，访问至出度为 0 的节点时，得到一条链，判断该链是否满足条件，如果是，搜索结束；如果不是，回溯并继续搜索，直到找到一条满足条件的链，或者 DAG 中所有极长链被搜索完毕。
3. 代码实现与细节

   使用 `std::string` 类对 instruction (函数的调用) 进行进一步简化，方便调试，为最后改为哈希值做好过渡工作；

   使用 LLVM 库中对控制流图缩点的实现：`sccNode`；

   实现 `FuncInfo` 类，整合关于某个特定 transaction 的所有图论信息，并实现关键判断函数`calcu_Func_support()`；

   实现 `SupportInfo` 类，作为后续函数调用的接口；由于会多次调用 `calcu_support()` 函数，这里采用记忆化搜索的技巧，避免多次调用算法复杂度瓶颈函数，提高效率；实现 `get_single_frequency()` 函数，用于后续的 `Apriori`算法。

`Apriori`算法的实现

1. 算法流程

   `Apriori`算法的本质是基于频繁项集阈值和非频繁项集阈值限制进行剪枝的广度优先搜索。

   用于剪枝的核心性质：若一个集合的某个子集为非频繁项集，那么它一定是非频繁项集。(同时，出于算法效率的考虑，可以忽略这个较大的非频繁项集。)

   算法流程大致如下：

   1. 按照项集大小从小到大依次处理。不妨记当前大小为 k。我们只需要处理由大小为 k - 1 的频繁项集“合并”而来的项集。每次将两个大小为k-1且交集大

小为k的集合合并起来，去重后加入新集合。
2. 利用核心性质剪枝，去掉无用项集，这一点将使用哈希函数进行快速判断。
3. 通过依次计算各个项集的支持度，判断待处理项集是否为频繁项集或非频繁项集。

2. 代码实现与细节

基于代码实现、时间效率等方面，决定使用 `std::vector` 对项集进行存储。

将第一步中"合并"的步骤抽象为私有成员函数 `expandItemset`，有助于代码模块化，增加可读性。

使用库函数 `std::set_difference` 等进行集合操作。

实现函数 `ItemSetHash()` 对项集整体进行哈希，优化多次出现的项集比较操作。在哈希的过程中，采用BKDRHash将项集中的每个元素进行Hash，存入`unsigned int`中并使用自然溢出。对于每个项集，同样采用类似BKDRHash的思路，将每个元素的Hash值乘以对应Base的k次幂，存入`unsigned long long`中使用自然溢出。对于两部分采用不同的base，以及不同的数据类型，尽可能避免碰撞。实际测试中，这样的hash策略表现优秀。

尽管如此，受Apriori算法本身的指数级复杂度限制，并不能在短时间内运行出较大型的测试代码。对此，在后续时间允许的情况下，可以改用FP-Growth算法，通过FP-Tree构造不同的K频繁项集，减少无用的集合枚举，可以大大改善这一步的运行效率。

关联规则的生成

1. 算法流程
    1. 正关联规则的生成

    枚举每个频繁项集 I 的大小 -1 的子集 A；如果：$\frac{support(I)}{support(A)} \geq min_{conf}$ 则称二元有序对 $R = (A, C_I A)$ 为一个正关联规则 R，

    $conf(R) = \frac{support(I)}{support(A)}$

    为该规则的置信度；
    2. 负关联规则的生成

    找到频繁项集 I 的大小 -1 的，支持度子集 A；如果：

    $1 - \frac{support(I)}{support(A)} \geq min_{conf}$ 则称二元有序对 $R = (A, C_I A)$ 为一个负关联规则 R，

$$conf(R) = 1 - \frac{support(I)}{support(A)}$$

为该规则的置信度。

2. 代码实现与细节

使用 `std::vector` 进行集合操作。

使用 `std::swap` 以及成员函数 `pop_back` 实现生成大小 -1 的子集的操作。

使用 `std::tuple` 对规则进行存储。

注意事项：需要特判子集置信度为 0 的情况，防止出现除以 0 的错误。

运行结果

如下为样例 `Test6.c` 与 `Test7.c` 的运行结果，与作业 3, 4 中的结果基本一致。

File: `Test6.out`

```
Frequent Itemsets:
        { "free(i8*)" "i8* = malloc(i32)" }

Infrequent Itemsets:
        { "free(i8*)" "free(i8*)" }

PARs:
        Rule: { free(i8*) } → { i8* = malloc(i32) }
        confidence rate = 1.000

        Rule: { i8* = malloc(i32) } → { free(i8*) }
        confidence rate = 0.923

NARs:
        Rule: { free(i8*) } → { free(i8*) }
        confidence rate = 0.917

```

File: `Test7.out`

```
Frequent Itemsets:
        { "bar(i32)" "i32 = is_valid(i32)" }
        { "bar(i32)" "i32 = read2( ... )" }
        { "foo(i32)" "i32 = is_valid(i32)" }
        { "foo(i32)" "i32 = read2( ... )" }
        { "i32 = is_valid(i32)" "i32 = read2( ... )" }
        { "bar(i32)" "i32 = is_valid(i32)" "i32 = read2( ... )" }
```

```
        { "foo(i32)" "i32 = is_valid(i32)" "i32 = read2( ... )" }

Infrequent Itemsets:
        { "bar(i32)" "foo(i32)" "i32 = is_valid(i32)" }

PARs:
        Rule: { i32 = is_valid(i32) } → { bar(i32) }
        confidence rate = 1.000

        Rule: { bar(i32) } → { i32 = is_valid(i32) }
        confidence rate = 1.000

        Rule: { i32 = read2( ... ) } → { bar(i32) }
        confidence rate = 1.000

        Rule: { bar(i32) } → { i32 = read2( ... ) }
        confidence rate = 1.000

        Rule: { i32 = is_valid(i32) } → { foo(i32) }
        confidence rate = 1.000

        Rule: { foo(i32) } → { i32 = is_valid(i32) }
        confidence rate = 1.000

        Rule: { i32 = read2( ... ) } → { foo(i32) }
        confidence rate = 1.000

        Rule: { foo(i32) } → { i32 = read2( ... ) }
        confidence rate = 1.000

        Rule: { i32 = read2( ... ) } → { i32 = is_valid(i32) }
        confidence rate = 1.000

        Rule: { i32 = is_valid(i32) } → { i32 = read2( ... ) }
        confidence rate = 1.000

        Rule: { i32 = read2( ... ) i32 = is_valid(i32) } → { bar(i32) }
        confidence rate = 1.000

        Rule: { bar(i32) i32 = read2( ... ) } → { i32 = is_valid(i32) }
        confidence rate = 1.000

        Rule: { bar(i32) i32 = is_valid(i32) } → { i32 = read2( ... ) }
        confidence rate = 1.000

        Rule: { i32 = read2( ... ) i32 = is_valid(i32) } → { foo(i32) }
        confidence rate = 1.000

        Rule: { foo(i32) i32 = read2( ... ) } → { i32 = is_valid(i32) }
```

```
        confidence rate = 1.000

        Rule: { foo(i32) i32 = is_valid(i32) } → { i32 = read2( ... ) }
        confidence rate = 1.000

NARs:
        Rule: { bar(i32) i32 = is_valid(i32) } → { foo(i32) }
        confidence rate = 0.917

```