电子科技大学计算机科学与工程学院

标 准 实 验 报 告

（实验）课程名称 数据结构与算法

电子科技大学教务处制表

电 子 科 技 大 学

实　验　报　告

学生姓名：陶浩轩　　学 号：2023080902011　　指导教师：陈端兵

一、 实验室名称：

学知三组团 10 栋 139

二、实验项目名称：

频繁模式挖掘

三、实验原理：

实验基于 Apriori 算法，利用先验性质，通过连接和剪枝操作，从给定的数据集中挖掘出频繁项集和关联规则。

四、实验目的：

1. 理解频繁模式挖掘的概念和重要性。
2. 掌握 Apriori 算法的基本原理和实现方法。
3. 利用 Apriori 算法挖掘数据集中的频繁项集和关联规则。
4. 分析频繁模式挖掘的结果，并进行解释和应用。

五、实验内容：

数据准备： 随机生成指定范围的数据

参数设置： 设置支持度阈值，确定挖掘频繁项集的最小支持度。

算法实现： 编写 Apriori 算法的代码，实现频繁项集的挖掘。

结果分析： 分析挖掘出的频繁项集和关联规则，并进行解释和应用。

六、实验器材（设备、元器件）：

平台：WSL Ubuntu 22.04 on Windows 10

七、实验步骤：

代码实现在 https://github.com/WhaleFall-UESTC/UESTC-DataStructure/tree/master/proj1

1. 定义 file，随机生成数据以及将文件读取为 database，并设置基本参数（随机变量以及最小支持度的范围）

```c
#define OUTPUT_ALL      1
#define OUTPUT_MAX      2
#define OUTPUT_TOPN     4
#define OUTPUT_DEFAULT  OUTPUT_ALL

#define DEFAULT_FILE "src.txt"
#define DEFAULT_TOPN    5
#define DEFAULT_MIN_SUP 3

#define RAND(l, u) (rand() % (l - u + 1) + l)
#define MAX_LEN         54
#define MIN_LEN         10
#define MAX_INDEX       128
#define MIN_INDEX       64
#define MAX_GROUP_NUM   64
#define MIN_GROUP_NUM   28

#define BUF 512

int         read_file(database *db, const char *filename);
int         random_file(const char *filename);

```

2. 定义项集的存储方式：类似于 bitmap，定义 itemset 结构，描述从某一地址开始一定长度的比特串，其中从开始的第 i 位的比特，1 表示编号 i 的元素存在于该项集，0 表示不存在，并定义 freqitem 与 freqlist 结构，以链表的形式存储 itemset

```c
#define NBITS 8     // itemset 块的位数
#define NBYTES (NBITS >> 3)
#define IDX2SIZE(idx) (((idx) >> 3) + 1)
typedef unsigned char uint8;
typedef uint8* itemset;
typedef uint8  itemblock;

#define CHECKBIT(n, off) ((n) & (1 << off))
#define NO(i, j) (i * NBITS + j + 1)

#define ALLOC   1
#define NOALLOC 0

itemset     init_itemset(int size);
int         contain(itemset p, itemset c, int size);
itemset     conjunct(itemset s1, itemset s2, int size, int k, int alloc);
void        add_itemset(itemset items, int no);
void        del_itemset(itemset items, int no);
itemset     copy_itemset(itemset items, int size);
void        print_itemset(itemset items, int size);

```

```c
struct freqitem {
    itemset items;
    int sup;
    struct freqitem* next;
};
typedef struct freqitem freqitem;

struct freqlist {
    int len;        // length of list
    int size;       // size of itemset
    int k;
    freqitem list;
};
typedef struct freqlist freqlist;

freqlist*           init_freqlist(freqlist* fl, int k, int size);
void                insert_freqlist(freqlist *fl, itemset items, int sup);
void                print_freqlist(freqlist *fl);
void                free_freqlist(freqlist *fl);
int                 itemset_in_freqlist(freqlist* fl, itemset items);

```

3. 定义 database 结构，此结构存储最初始的项集，是文件中读取的数据存储在内存中的结构

```c
struct dbitem {
    itemset items;
    struct dbitem* next;
};
typedef struct dbitem dbitem;

struct database {
    int len;
    int max_index;
    int size;
    dbitem list;
};
typedef struct database database;

database*           init_db(database* db);
void                insert_db(database* db, itemset items);
int                 count_minsup(database* db, itemset items);
void                print_db(database db);
freqlist*           scan_db(database db, int minsup);
void                free_db(database* db);

```

4. 编写 debug.h 与 Makefile 方便调试与测试

5. 基础设施准备完毕，单元测试各模块后，开始在 main 函数实现 Apriori 算法

定义全局变量 db 以及与结果输出，文件读取相关的参数：

```c
#define BUF 512
#define RING_NEXT(p) (((p) + 1) % RING_BUF)
#define RING_PRE(p)  (((p) + RING_BUF - 1) % RING_BUF)

#define LWC_SUP    1    // filter whose sup is less than min_sup
#define LWC_CHILD  2    // filter whose child is not frequent
#define LWC_REPEAT 4    // filter those repeat
#define LWC_ALL    7    // filter all

#define SUP(sup) ((double) 1.0 * (sup) / db.len)

static unsigned outset = OUTPUT_DEFAULT;
static char filename[BUF] = DEFAULT_FILE;
static int topn = DEFAULT_TOPN;
static bool rand_file = false;
static bool rand_only = false;
static int min_support = DEFAULT_MIN_SUP;

static database db;
```

Apriori 算法的连接与剪枝操作。虽然从概念上这两个操作是解耦的，但是一些过滤需要二者的过程信息，出于效率考虑，我选择将它们合并在一起

```
freqlist*
link_with_cut(freqlist* fl, int flag)
{
    freqlist* ret = init_freqlist(NULL, fl->k + 1, fl->size);
    freqitem* pi = fl->list.next;
    bool enter_cycle = false;
    while (pi && pi->next) {
        enter_cycle = true;
        freqitem* pj = pi->next;
        while (pj) {
            itemset comb = conjunct(pi->items, pj->items, fl->size, fl->k + 1, ALLOC);
            if (comb == NULL) {
                pj = pj->next;
                continue;
            }
            bool pass_sup = true, pass_child = true;
            int sup = count_minsup(&db, comb);
            if (flag & LWC_SUP) {
                if (sup < min_support) {
                    pass_sup = false;
                }
            }
            if ((flag & LWC_CHILD) && pass_sup) {
                int* nos = (int*) malloc(fl->size * sizeof(int));
                int nop = 0;
                int len = fl->size / NBYTES;
                for (int i = 0; i < len; i++) {
                    for (int j = 0; j < NBITS; j++)
                        if (CHECKBIT(comb[i], j)) {
                            nos[nop++] = NO(i, j);
                        }
                }
                for (int i = 0; i < nop; i++) {
                    itemset child = copy_itemset(comb, fl->size);
                    del_itemset(child, nos[i]);
                    if (itemset_in_freqlist(fl, child) < 0) {
                        free(child);
                        pass_child = false;
                        break;
                    }
                    free(child);
                }
                free(nos);
            }
            if (pass_sup && pass_child) {
                insert_freqlist(ret, comb, sup);
            } else {
                free(comb);
            }
            pj = pj->next;
        }
        pi = pi->next;
    }

    if (!enter_cycle || ret->len == 0) {
        free(ret);
        return NULL;
    }
    return ret;
}
```

在 main 函数中，先解析参数，生成随机数据，读取数据并建立 database，并以此获取第一个 k = 1 的频繁项集，随后不断调用 link_with_cut 筛选

```c
int
main(int argc, char *argv[])
{
    init_db(&db);
    prase_args(argc, argv);

    if (rand_file)
        random_file(filename);
    if (rand_only)
        return 0;
    read_file(&db, filename);

    freqlist* fl[BUF];
    fl[0] = scan_db(db, min_support);
    print_freqlist(fl[0]);
    fl[1] = link_with_cut(fl[0], LWC_SUP);
    if (fl[1] == NULL) goto end;
    int j = 2;
    while ((fl[j] = link_with_cut(fl[j - 1], LWC_ALL)) != NULL)
        j++;

    Log("Apriori Screening Finish");

```

后续再处理输出结果。为了按照从小到大的顺序输出给定规则的频繁项集，我构建另一个变长的最小堆来存储 Apriori 每一轮的结果

```c
#define HEAP_INIT 8

struct heapitem {
    itemset items;
    int sup;
};
typedef struct heapitem heapitem;

struct heap {
    heapitem *pq;
    int n;
    int capacity;
};
typedef struct heap heap;

typedef unsigned char bool;
#define true  1
#define false 0

heap*           init_heap();
void            insert_heap(heap* h, itemset items, int sup);
heapitem        pop_heap(heap* h);
void            add_freqlist(heap* h, freqlist *fl);
void            free_heap(heap* h);

```

至此，初步实现了 Apriori 算法

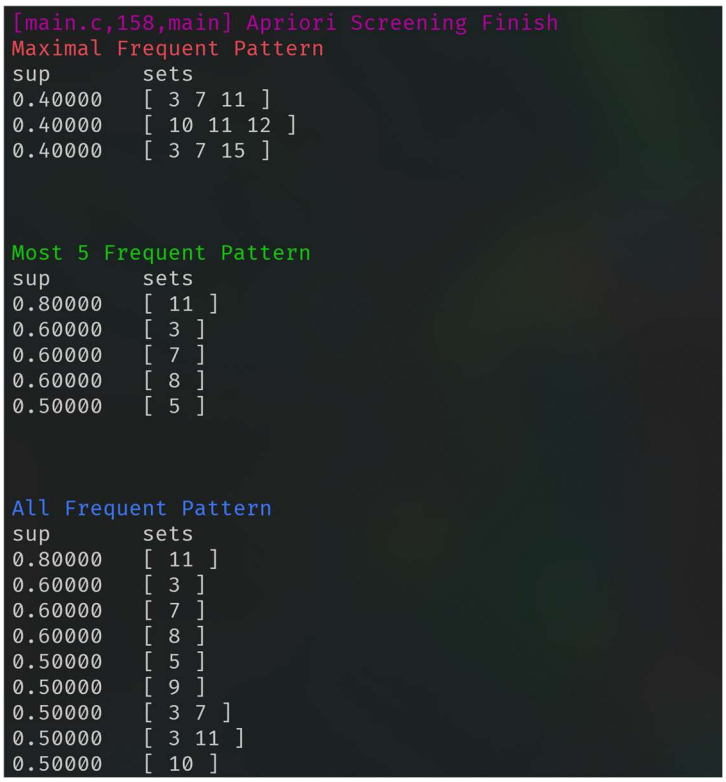
八、实验数据及结果分析：

我生成了一个小的数据集来进行测试，设置 min_support = 0.4

```
2 3 4 6 7 9 10 11 12 15
1 6 7 9 10 11 12
1 4 5 8 9 10 12
1 3 4 6 8 10 11 12 13 14
1 6 8 9 11
2 3 7 8 10 11 12 13 15
3 5 7 8 9 11 13 14 15
1 3 4 5 7 13 14 15
5 8 11 14 15
2 3 4 5 7 11 14
```

这些数据可以表示一个图书馆内的部分顾客借阅过的书籍的类型

通过运行程序，我们可以得到一下的结果：



我们可以通过 Apriori 算法的到频繁项集，可以得知有哪些编号的关联性大

比如 3 类型代表犯罪心理学书籍，7 代表侦探小说，我们可以发现这两类书被同一人借阅的概率大，于是我们可以将这两类书的书架搬近一点

九、实验结论：

Apriori 算法是数据挖掘方面一个经典的算法。通过合并，剪枝，不断地筛选来找到频繁项集。我们可以通过这个方法来获取大量数据中的频繁模式，但是其平方时间复杂度使得其效率有些堪忧（一些较大但有没那么大的数据集甚至跑到了十几秒）。在使用的过程中，我们

应当多采取一下优化的措施，改良算法

十、总结及心得体会：

1. 本次我使用了 bitmap 的形式存储频繁项集，不仅节省了空间（虽然 Apriori 更关心时间复杂度），而且在判断是否为子集时只需要将二者 and，合并时只需要将二者 or，操作上也更加快捷

2. 我尝试设计了一套层级结构：freqitem 与 freqlist 建立在 itemset 之上；database 建立在 freqlist 之上，这套层级结构方便了程序的构建

十一、对本实验过程及方法、手段的改进建议：

1. 我们需要在 database 中查找频繁项集出现的次数，采用的是顺序查找的方式。优化的方案是：我们可以利用哈希表来减小查找次数。根据 itemset 的长度，我们可以按照偏移量均匀地选取 4 个比特，并用这 4 个比特所组成的无符号数的值作为 hashcode
2. 如果 itemset 的长度较小（小于 128），那么这个 itemset 是对 C 而言一个天然的无符号数表示。我们可以在哈希桶内采用二分查找（或者构建二叉搜索树）的方式来减小搜索次数。此外，还可以根据 itemset 的长度与数量的大小决定使用基数排序还是快速排序
3. 对应的，在 database 中可以采用变长数组的方式而非链表（即初始长度固定，若所要的容量超过了这个长度，则再开辟一个大小为原来的两倍的空间，将数据复制到这个空间，释放原来的空间），可以加快查找速度
4. 或者说构建一棵树，以一个项集的第一个元素作为根，若是其他项集有与其共同的部分，但是后续的项目有所不同，则在这棵树上“分支”，并且树要求子节点必须按照某一顺序大于根节点。这样在计数的时候可以快速地获取支持度
5. 最开始写代码的时候可以参照网上的测试集的格式来编写
6. 再分析优化程序时，可以使用 ftrace 跟踪函数调用与开销，针对性地优化

报告评分：

指导教师签字：