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）

```h
#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         64
#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);
```

file.h

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

```h
#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);
```

itemset.h

```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);
```

freqlist.h

3. 定义 database 结构，此结构存储最初始的项集

```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);
```

database.h

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
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;

static int
prase_args(int argc, char *argv[])
```

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);

    Log("Initalize database");

    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

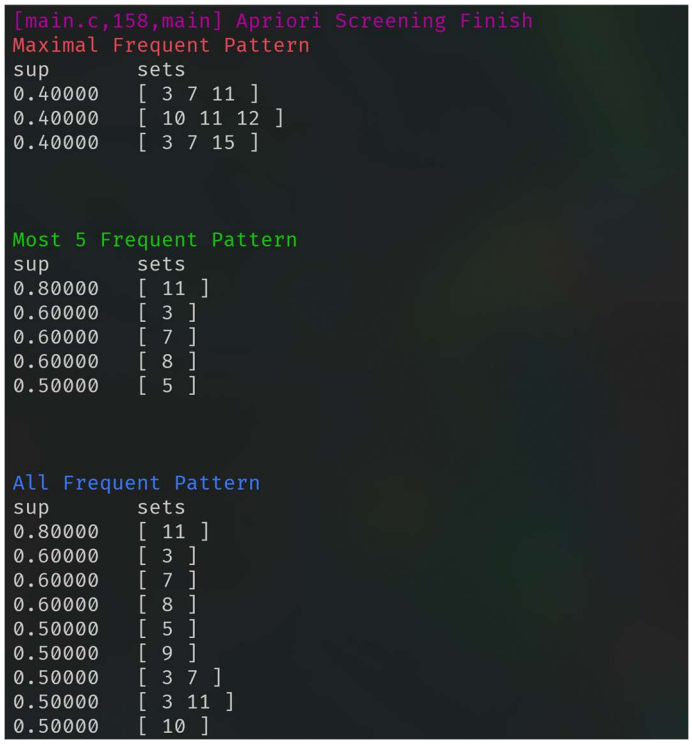
## 八、实验数据及结果分析：

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：**

# 附录：源代码

报告前面已经展示过头文件源代码，故这里只显示 .c 文件与 Makfile 和 debug.h

## main.c

```c
#include <unistd.h>
#include <getopt.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "itemset.h"
#include "freqlist.h"
#include "database.h"
#include "file.h"
#include "heap.h"
#include "debug.h"

#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;

static int
prase_args(int argc, char *argv[])
{
    const struct option table[] = {
        {"src"  , optional_argument, NULL, 's'},
        {"all"  , no_argument      , NULL, 'A'},
        {"max"  , no_argument      , NULL, 'M'},
        {"top"  , optional_argument, NULL, 'T'},
        {"rand" , optional_argument, NULL, 'r'},
        {"Rand" , optional_argument, NULL, 'R'},
        {"min-support", optional_argument, NULL, 'm'},
```

```
        {0      , 0                 , NULL,  0 },
    };
    int o;
    while ((o = getopt_long(argc, argv, "s::AMT::r::m::R::", table, NULL)) != -1) {
        switch (o) {
            case 's': if (optind < argc && argv[optind][0] != '-' && optarg == NULL) {
                        strcpy(filename, argv[optind++]);
                      }
                      break;
            case 'A': outset |= OUTPUT_ALL; break;
            case 'M': outset |= OUTPUT_MAX; break;
            case 'T': outset |= OUTPUT_TOPN;
                      if (optarg)
                        sscanf(optarg, "%d", &topn);
                      break;
            case 'R': rand_only = true;
            case 'r': rand_file = true;
                      if (optind < argc && argv[optind][0] != '-' && optarg == NULL) {
                        strcpy(filename, argv[optind++]);
                      }
                      break;
            case 'm': min_support = (optarg ? atoi(optarg) : min_support);
                      break;
            default:
                printf("Usage: too lazy to write it\n");
        }
    }
    return 0;
}

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;
}

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);

    Log("Initalize database");

    freqlist* fl[BUF];
```

```
    fl[0] = scan_db(db, min_support);
    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");


    heap *h = init_heap();
    for (int i = 0; i < j; i++)
        add_freqlist(h, fl[i]);

    if (outset | OUTPUT_MAX) {
        printf(L_RED "Maximal Frequent Pattern\n" NONE);
        heap* hmax = init_heap();
        freqitem* ptr = fl[j - 1]->list.next;
        while (ptr) {
            insert_heap(hmax, ptr->items, ptr->sup);
            ptr = ptr->next;
        }
        printf("sup       sets\n");
        while (hmax->n) {
            heapitem item = pop_heap(hmax);
            printf("%.5lf   ", SUP(item.sup));
            print_itemset(item.items, db.size);
        }
        puts("\n\n");
    }

    heapitem *top = (heapitem*) malloc(topn * sizeof(heapitem));
    int ptop = 0;
    if (outset |= OUTPUT_TOPN) {
        printf(L_GREEN "Most %d Frequent Pattern\n" NONE, topn);
        printf("sup       sets\n");
        int cnt = topn;
        while (cnt--) {
            top[ptop] = pop_heap(h);
            printf("%.5lf   ", SUP(top[ptop].sup));
            print_itemset(top[ptop].items, db.size);
            ptop++;
        }
        puts("\n\n");
    }

    if (outset | OUTPUT_ALL) {
        printf(L_BLUE "All Frequent Pattern\n" NONE);
        printf("sup       sets\n");
        if (outset | OUTPUT_TOPN) {
            for (int i = 0; i < ptop; i++) {
                printf("%.5lf   ", SUP(top[i].sup));
                print_itemset(top[i].items, db.size);
```

```
            }
        }
        while (h->n) {
            heapitem item = pop_heap(h);
            printf("%.5lf   ", SUP(item.sup));
            print_itemset(item.items, db.size);
        }
        puts("\n\n");
    }

    for (int i = 0; i < j; i++)
        free_freqlist(fl[i]);
    free_heap(h);
    free(top);
end:
    free_db(&db);
    puts("Over");
    return 0;
}
```

## itemset.c

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "itemset.h"

itemset
init_itemset(int size)
{
    itemset ret = (itemset) malloc(size);
    memset(ret, 0, size);
    return ret;
}

int
contain(itemset p, itemset c, int size)
{
    int len = size / NBYTES;
    for (int i = 0; i < len; i++) {
        if ((p[i] | c[i]) != p[i]) {
            return -1;
        }
    }
    return 0;
}

static inline int
count_block_bits(itemblock n)
```

```
{
    int cnt = 0;
    while (n) {
        n &= (n - 1);
        cnt++;
    }
    return cnt;
}

/* conjunct two itemset. if the num of 1s > k, return NULL */
/* set alloc=1 to malloc a new itemset */
/* otherwise, the result will be stored in s1 */
itemset
conjunct(itemset s1, itemset s2, int size, int k, int alloc)
{
    itemset ret;
    if (alloc) ret = init_itemset(size);
    else ret = s1;

    int len = size / NBYTES;
    int cnt_k = 0;
    for (int i = 0; i < len; i++) {
        ret[i] = (s1[i] | s2[i]);
        cnt_k += count_block_bits(ret[i]);
        if (cnt_k > k) {
            if (alloc) free(ret);
            return NULL;
        }
    }
    return ret;
}

/* no starts from 1 */
void
add_itemset(itemset items, int no)
{
    no -= 1;
    int idx = no / NBITS;
    int off = no - idx * NBITS;
    items[idx] |= (1 << off);
}

void
del_itemset(itemset items, int no)
{
    no -= 1;
    int idx = no / NBITS;
    int off = no - idx * NBITS;
    items[idx] &= ~(1 << off);
}

itemset
copy_itemset(itemset items, int size)
```

```
{
    itemset ret = init_itemset(size);
    int len = size / NBYTES;
    for (int i = 0; i < len; i++) {
        ret[i] = items[i];
    }
    return ret;
}

void
print_itemset(itemset items, int size)
{
    int len = size / NBYTES;
    printf("[ ");
    for (int i = 0; i < len; i++)
        for (int j = 0; j < NBITS; j++)
            if (CHECKBIT(items[i], j))
                printf("%d ", NO(i, j));
    printf("]\n");
}
```

## freqlist.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "itemset.h"
#include "freqlist.h"

freqlist*
init_freqlist(freqlist* fl, int k, int size)
{
    if (fl == NULL)
        fl = (freqlist*) malloc(sizeof(freqlist));
    memset(fl, 0, sizeof(freqlist));
    fl->k = k;
    fl->size = size;
    return fl;
}

void
insert_freqlist(freqlist *fl, itemset items, int sup)
{
    freqitem* ptr = &fl->list;
    if (fl->len == 0)
        goto load;

    while (ptr = ptr->next) {
        if (contain(ptr->items, items, fl->size) == 0) {
```

```
            return;
        }
        if (ptr->next == NULL)
            break;
    }

load:
    freqitem* fi = (freqitem*) malloc(sizeof(freqitem));
    fi->items = items;
    fi->sup = sup;
    fi->next = NULL;
    ptr->next = fi;
    fl->len++;

}

void
print_freqlist(freqlist *fl)
{
    printf("freqlist info at %p\n", fl);
    printf("len: %d\tsize: %d\n", fl->len, fl->size);
    freqitem *ptr = fl->list.next;
    printf("sup   itemset\n");
    while (ptr) {
        printf("%3d   ", ptr->sup);
        print_itemset(ptr->items, fl->size);
        ptr = ptr->next;
    }
}

void
free_freqlist(freqlist *fl)
{
    if (fl == NULL)
        return;
    freqitem *ptr = fl->list.next;
    while (ptr) {
        freqitem *tmp = ptr;
        ptr = ptr->next;
        free(tmp);
    }
    free(fl);
}

int
itemset_in_freqlist(freqlist* fl, itemset items)
{
    freqitem* ptr = fl->list.next;
    while (ptr) {
        if (contain(ptr->items, items, fl->size) == 0) {
            return 0;
        }
        ptr = ptr->next;
```

```
    }
    return -1;
}
```

## database.c

```
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include "itemset.h"
#include "freqlist.h"
#include "database.h"

database*
init_db(database* db)
{
    if (db == NULL)
        db = (database*) malloc(sizeof(database));
    memset(db, 0, sizeof(database));
    return db;
}

void
insert_db(database* db, itemset items)
{
    dbitem* ptr = &db->list;
    while (ptr->next)
        ptr = ptr->next;
    ptr->next = (dbitem*) malloc(sizeof(dbitem));
    ptr = ptr->next;
    ptr->items = items;
    ptr->next = NULL;
}

int
count_minsup(database* db, itemset items)
{
    int cnt = 0;
    dbitem* ptr = db->list.next;
    while (ptr) {
        if (contain(ptr->items, items, db->size) == 0) {
            cnt++;
        }
        ptr = ptr->next;
    }
    return cnt;
}

void
```

```
print_db(database db)
{
    printf("database info:\n");
    printf("len: %d\n", db.len);
    printf("size: %d\tmax_index: %d\n", db.size, db.max_index);
    dbitem* ptr = db.list.next;
    while (ptr) {
        print_itemset(ptr->items, db.size);
        ptr = ptr->next;
    }
    printf("\n");
}

freqlist*
scan_db(database db, int minsup)
{
    freqlist* fl = init_freqlist(NULL, 1, db.size);

    dbitem* ptr = db.list.next;
    int len = db.size / NBYTES;
    int* support = (int *) malloc(db.max_index * sizeof(int));
    memset(support, 0, db.max_index * sizeof(int));
    while (ptr) {
        for (int i = 0; i < len; i++)
            for (int j = 0; j < NBITS; j++)
                if (CHECKBIT(ptr->items[i], j))
                    support[NO(i, j) - 1]++;
        ptr = ptr->next;
    }

    for (int i = 0; i < db.max_index; i++) {
        if (support[i] >= minsup) {
            itemset items = init_itemset(db.size);
            add_itemset(items, i + 1);
            insert_freqlist(fl, items, support[i]);
        }
    }
    free(support);
    return fl;
}

void
free_db(database* db)
{
    if (db == NULL)
        return;
    dbitem* ptr = db->list.next;
    while (ptr) {
        dbitem* tmp = ptr;
        ptr = ptr->next;
        free(tmp);
    }
}
```

## file.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include "itemset.h"
#include "freqlist.h"
#include "database.h"
#include "file.h"

int
read_file(database *db, const char *filename)
{
    FILE *fp = fopen(filename, "r");
    if (fp == NULL) {
        return -1;
    }

    fscanf(fp, "%d\n", &db->len);
    fscanf(fp, "%d\n", &db->max_index);
    db->size = IDX2SIZE(db->max_index);

    char line[BUF];
    char *token;
    itemset items;
    while (fgets(line, BUF, fp) != NULL) {
        items = init_itemset(db->size);
        token = strtok(line, " \t\n");
        while (token != NULL) {
            int no = atoi(token);
            add_itemset(items, no);
            token = strtok(NULL, " \t\n");
        }
        insert_db(db, items);
    }

    fclose(fp);
    return 0;
}

int
random_file(const char *filename)
{
    FILE *fp = fopen(filename, "w");
    if (fp == NULL) {
        return -1;
    }
```

```
    srand((unsigned)time(NULL));
    int len = RAND(MIN_LEN, MAX_LEN);
    int max_index = RAND(MIN_INDEX, MAX_INDEX);
    fprintf(fp, "%d\n%d\n", len, max_index);

    for (int i = 0; i < len; i++) {
        int size = RAND(MIN_GROUP_NUM, MAX_GROUP_NUM);
        for (int j = 0; j < size; j++) {
            int no = RAND(1, max_index);
            fprintf(fp, "%d ", no);
        }
        fprintf(fp, "\n");
    }

    fclose(fp);
    return 0;
}
```

## heap.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "itemset.h"
#include "freqlist.h"
#include "heap.h"
#include "debug.h"

heap*
init_heap()
{
    heap* ret = (heap*) malloc(sizeof(heap));
    ret->n = 0;
    ret->capacity = HEAP_INIT;
    ret->pq = (heapitem*) malloc(HEAP_INIT * sizeof(heapitem));
    memset(ret->pq, 0, HEAP_INIT * sizeof(heapitem));
    return ret;
}

void
free_heap(heap* h)
{
    if (h == NULL) return;
    if (h->pq != NULL) free(h->pq);
    free(h);
}

static inline void
```

```
resize_heap(heap* h, int len)
{
    heapitem* prev = h->pq;
    h->pq = (heapitem*) malloc(len * sizeof(heapitem));
    memset(h->pq, 0, len * sizeof(heapitem));
    // zero to h->n, total h->n + 1
    memmove(h->pq, prev, (h->n + 1) * sizeof(heapitem));
    free(prev);
    h->capacity = len;
}

static inline bool
legal(heap* h, int idx)
{
    return (idx >= 1 && idx <= h->n);
}

static inline void
exch(heap* h, int idx1, int idx2)
{
    heapitem tmp;
    memmove(&tmp, &h->pq[idx1], sizeof(heapitem));
    memmove(&h->pq[idx1], &h->pq[idx2], sizeof(heapitem));
    memmove(&h->pq[idx2], &tmp, sizeof(heapitem));
}

static inline bool
smaller(heap* h, int idx1, int idx2)
{
    return (h->pq[idx1].sup < h->pq[idx2].sup);
}

static inline void
swim(heap* h, int k)
{
    while (k > 1 && smaller(h, k / 2, k)) {
        exch(h, k / 2, k);
        k /= 2;
    }
}

static inline void
sink(heap* h, int k)
{
    while (k * 2 < h->n) {
        int j = k * 2;
        if (j < h->n && smaller(h, j, j + 1))
            j++;
        if (!smaller(h, k, j))
            break;
        exch(h, k, j);
        k = j;
    }
```

```
}

static inline bool
is_heap_order(heap* h, int k)
{
    if (k > h->n) return true;
    int l = k * 2;
    int r = l + 1;
    if (l <= h->n && smaller(h, k, l)) return false;
    if (r <= h->n && smaller(h, k, r)) return false;
    return is_heap_order(h, l) && is_heap_order(h, r);
}

static inline bool
is_heap(heap* h) {
    for (int i = 1; i <= h->n; i++)
        if (h->pq[i].items == NULL) {
            Log("pq[%d] should not be NULL, h->n = %d", i, h->n);
            return false;
        }
    for (int i = h->n + 1; i < h->capacity; i++)
        if (h->pq[i].items != NULL) {
            Log("pq[%d] should be NULL, h->n = %d", i, h->n);
            return false;
        }
    if (h->pq[0].items != NULL)
        return false;
    return is_heap_order(h, 1);
}

void
insert_heap(heap* h, itemset items, int sup)
{
    if (h->n == h->capacity - 1)
        resize_heap(h, h->capacity * 2);
    heapitem item = (heapitem){.items = items, .sup = sup};
    memmove(&h->pq[++h->n], &item, sizeof(heapitem));
    swim(h, h->n);
    assert(is_heap(h));
}

heapitem
pop_heap(heap* h)
{
    assert(h->n > 0);
    heapitem ret;
    memmove(&ret, &h->pq[1], sizeof(heapitem));
    exch(h, 1, h->n--);
    sink(h, 1);
    memset(&h->pq[h->n + 1], 0, sizeof(heapitem));
    if ((h->n > 0) && (h->n == (h->capacity - 1) / 4) && h->capacity >= HEAP_INIT)
        resize_heap(h, h->capacity / 2);
    assert(is_heap(h));
```

```
    return ret;
}

void
add_freqlist(heap* h, freqlist *fl)
{
    freqitem* ptr = fl->list.next;
    while (ptr) {
        insert_heap(h, ptr->items, ptr->sup);
        ptr = ptr->next;
    }
}
```

## debug.h

```
#define Log(format, ...) \
  printf("\33[1;35m[%s,%d,%s] " format "\33[0m\n", \
    __FILE__, __LINE__, __func__, ## __VA_ARGS__)

#undef panic
#define panic(format, ...) \
  do { \
    Log("\33[1;31msystem panic: " format, ## __VA_ARGS__); \
    exit(-1); \
  } while (0)

#define assert(cond) \
  do { \
    if (!(cond)) { \
      panic("Assertion failed: %s", #cond); \
    } \
  } while (0)

#define L_RED                "\e[1;31m"
#define L_GREEN              "\e[1;32m"
#define L_BLUE               "\e[1;34m"
```

## Makefile

```
CC = gcc
CFLAGS = -g -Wall -O3
OBJS = main.o file.o itemset.o database.o freqlist.o heap.o
HEAP = heap.o itemset.o freqlist.o
TARGET = main
minsup = 4
```

```
topn = 5
TEST = test.txt
SRC = src.txt

all: $(TARGET)

run: all
	./$(TARGET) -m$(minsup) -s $(SRC) -A -M -T$(topn)

rand: all
	./$(TARGET) -R $(SRC)

test: all
	./$(TARGET) -m$(minsup) -r $(TEST) -A -M -T$(topn)

retry: all
	./$(TARGET) -m$(minsup) -s $(TEST) -A -M -T$(topn)

record: all
	./$(TARGET) -m$(minsup) -s $(TEST) -A -M -T$(topn) > record.txt

gdb: all
	gdb --args $(TARGET) -m5 -r $(TEST) -A -M -T$(topn)

heap: $(HEAP)
	$(CC) $(CFLAGS) -o heap $(HEAP)
	./heap

$(TARGET): $(OBJS)
	$(CC) $(CFLAGS) -o $(TARGET) $(OBJS)

main.o: main.c
	$(CC) $(CFLAGS) -c main.c

file.o: file.c
	$(CC) $(CFLAGS) -c file.c

itemset.o: itemset.c
	$(CC) $(CFLAGS) -c itemset.c

freqlist.o: freqlist.c
	$(CC) $(CFLAGS) -c freqlist.c

heap.o: heap.c
	$(CC) $(CFLAGS) -c heap.c

clean:
	rm -rf $(OBJS) $(TARGET)


.PHONY: all clean run test gdb retry record
```