

# 数据结构实验报告

丁诗哲 dingshizhe@gmail.com

2017 年 6 月 28 日

**题目：**编制一个利用 Kruskal 算法生成一个网的最小生成树的程序。

## 1 需求分析

### 1.1 基本数据类型

用一个含有两个静态数组变量和两个 `int` 变量的结构表示一个图。其中的一个数组存储图的边，另一个数组存储图的节点；其中的一个 `int` 变量存储图中边的个数，另一个存储图中节点的个数。

一个节点的变量类型为 `int`，一条边的变量类型为一个结构——这个结构中存在这条边带有的两个节点和边的权。

### 1.2 标准输入

用户需要标准化输入一个图，需要按照提示依次输入下列数据：点数，边数，边（包括两个点和一个权值，分别用空格隔开）。每次输入用换行分开。

### 1.3 标准输出

数据输入完毕后，用户将得到一个最小生成树的标准化输出，输出包含下列数据：点数，边数，点的列表以及边的列表。

### 1.4 其他

本程序需要另外一个数据结构——并查集（`MFset`）。

我们用一个结构描述并查集。此结构包含一个 `int` 型的静态数组和 `int` 型的描述数组大小的量。数组的第  $i$  个值描述  $i$  在哪个集合中。它的值初始为  $i$ 。

我们用并查集描述 Kruskal 算法运行过程中图的节点所属的连通分量。

## 2 概要设计

### 2.1 抽象数据结构定义

图（Graph）的抽象数据结构定义：

```
ADT Graph{
    数据对象 V: V是具有相同特性的数据元素的集合，称为顶点集；
    数据关系 R:  $R = \{VR\}$ ,  $VR = \{<v,w> | v,w \text{ 属于 } V\}$ ,  $R$ 是 $V \times V$ 的子集，称为边集；
    基本操作 P:
        Create_graph(&G, V, VR)
```

```

        初始条件：图G存在，V是顶点集，VR是边集。
        操作结果：以V和VR为条件生成一个图。
    Destroy_graph(&G)
        初始条件：图G存在。
        操作结果：销毁图G。
    Get_vex(&G, v)
        初始条件：图G存在，边v存在。
        操作结果：得到G中v的权重。
    Put_vex(&G, v, weight)
        初始条件：图G存在，边v存在。
        操作结果：给边v赋权weight。
    Add_vex(&G, v)
        初始条件：图G存在，边v不存在。
        操作结果：给G加边v。
    Initial_mini_gtree(&G)
        初始条件：图G存在。
        操作结果：创建一个图，大小和G相同但是不含边。
            （为最小生成树做准备）
    Print_graph(&G)
        初始条件：图G存在。
        操作结果：在终端打印出图G的信息。
}

```

### 并查集（MFset）的抽象定义：

```

ADT MFset{
    数据对象 S: S是一个MFset，则它的元素是一系列子集S[i] (i=1,2...m)，
        每个子集的元素是{0,1...n}中的一个整数。
    数据关系 R: S[1]、S[2]...S[m]的并集为初始定义的全集。
    基本操作 P:
        Initial(&S, n, x1, x2...xn)
            操作结果：
        Find(&S, x)
            初始条件：S存在，x是S子集的一个元素。
            操作结果：返回x所在的S[i]。
        Merge(&S, i, j)
            初始条件：S存在。

```

```

        操作结果：将i和j所在的S的子集合并。
    Destroy(&S)
        初始条件：S存在。
        操作结果：将S销毁。
    IsSame(&S, i, j)
        初始条件：S存在，i和j在S的子集中。
        操作结果：返回i和j是否在同一个子集中。
    PrintMfset(&S)
        初始条件：S存在。
        操作结果：打印出S的值。
}

```

## 2.2 程序构成

本程序包含四个模块：主程序模块、图结构模块、并查集结构模块、克鲁斯卡尔算法实现模块。

**主程序模块：**在 main.c 中，程序结构如下：

```

int main(){
    初始化图G;
    打印图G;
    对G进行Kruskal算法;
    打印图;
    销毁图;
    返回0;
}

```

### 图结构模块：

在文件 graph.h 和 graph.c 中，graph.h 声明了图的结构和相关操作。graph.c 中给出了操作的具体定义。

graph.h 中声明的操作有：

```

Graph create_graph(void);
int delete_graph(Graph G);
int add_edge(edge tmp, Graph G);
Graph create_test_graph(void);

```

```
Graph initial_mini_gTree(Graph G);
int print_graph(Graph G);
```

其中 *Graph create\_graph(void)* 是由用户输入数据生成一个图, *Graphcreate\_test\_graph(void)* 是读取一个文件的数据生成一个特定的测试图, 该文件 (*test.txt*) 已经被写好。其他操作分别对上面 Graph 抽象 ADT 中的各种操作。

**并查集模块：**

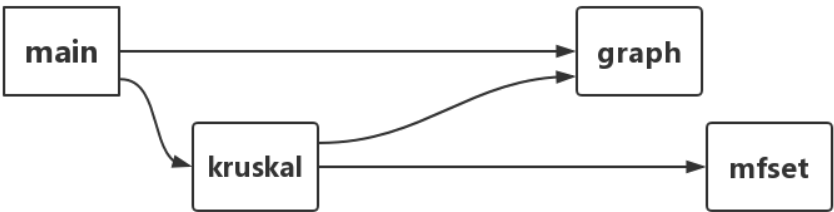
在文件 *mfset.h* 和 *mfset.c* 中, *mfset.h* 声明了图的结构和相关操作。*mfset.c* 中给出了操作的具体定义。 *mfset.h* 中声明的操作有：

```
MFset create_mfset(int size);
int find(int x, MFset MF);
int is_insame(int x, int y, MFset MF);
int merge(int x, int y, MFset MF);
int delete_mfset(MFset MF);
int print_mfset(MFset MF);
```

它们分别对上面 MFset 抽象 ADT 中的各种操作。

**克鲁斯卡尔算法实现模块：**在文件 *kruskal.c* 中。

文件的调用关系如下：



*main.c* 调用 *graph.h* 和 *mfset.h* 中声明的函数, 也调用 *kruskal.c* 中的函数。*kruskal.c* 需要 *graph.h* 和 *mfset.h* 中声明的函数。

### 3 详细设计

#### 3.1 图的结构类型

```
// 边结构
struct edge{
    int from, to;
    int weight;
};

typedef struct edge edge;

// 图结构 ( 结点集合, 边集合, 结点数, 边数 )
struct Graph_box{
    char vertices[30];
    edge edges[100] ;
    int vexnum, edgenum;
};

typedef struct Graph_box * Graph;
```

### 3.2 对图的一些操作

下面是创建一个图 (*Graph create\_graph(void)*) 的程序代码, 包含提示输入数据类型, 获取标准输入, 根据标准输入创建无边图, 依次加入边。

```
// 按输入建立一个图
Graph create_graph(void){

    Graph G = (Graph) malloc(sizeof(struct Graph_box));
    if (!G) return NULL;

    printf("输入顶点数\n");
    scanf("%d", &(G->vexnum));
    printf("输入边数\n");
    scanf("%d", &(G->edgenum));

    int i;
    for (i=0; i<G->vexnum; i++)
        G->vertices[i] = 'A' + i;
```

```

G->vertices[i] = '\0';
printf("输入边\n");

// 初始化
for(i=0; i<100; i++)
G->edges[i].weight = 0;

// 临时
edge tmp;

// 新的边插入原有的边列表，保持按权重降序
for(i=0; i<G->edgenum; i++){
    scanf("%d_%d_%d", &(tmp.from), &(tmp.to), &(tmp.weight));
    add_edge(tmp, G);
}
return G;
}

```

下面是在图  $G$  中加入一条边  $tmp$  的程序 (*int add\_edge(edge tmp, Graph G)*) 代码。需要注意的是，加入边的过程中，每次加不仅需要判断和已有的边是否重复，如果不重复、还需要加入后边的数组按照边的权重保持降序。这是为了运行 Kruskal 算法的方便。

```

int add_edge(edge tmp, Graph G){
    if(!G) return -1;

    // 加入的边不能和已有的边重复
    for(int i=0; i<G->edgenum; i++)
        if((tmp.from==G->edges[i].from && tmp.to==G->edges[i].to)||
            (tmp.from==G->edges[i].to&&tmp.to==G->edges[i].from))
            return -1;

    // 保证加入边后列表按照权重降序
    for(int j=0; j<=G->edgenum; j++)
        if(tmp.weight > G->edges[j].weight){
            for(int k=G->edgenum; k>j; k--)
                G->edges[k] = G->edges[k-1];

```

```

        G->edges[j] = tmp;
        break;
    }
    G->edgenum++;
    return 0;
}

```

### 3.3 并查集的结构类型

```

// 并查集
struct mfset{
    int set[MAX_NUM];
    int size;
};

typedef struct mfset * MFset;

```

### 3.4 对并查集的部分操作

下面是将并查集  $MF$  中的  $i$  和  $j$  所在的子集合并的程序 ( $intmerge(int\ x, int\ y, MFset\ MF)$ )。现在用  $f(i)$  表示  $MF$  结构内的数组第  $i$  位的值。

我们在前面的定义中规定,  $f(i)$  表示其所在的集合, 不同的  $i$  和  $j$  对应的  $f(i)$ 、 $f(j)$  相等, 意味着他们在同一个集合中。我们将  $i$  和  $j$  所在的集合合并, 只需将和  $f(i)$  以及  $f(j)$  相等的所有的位置  $x$  的  $f(x)$  都设为相同的值即可。但是我需要设置他们的值和其他集合的值不相等, 考虑到数组初始化的时候  $f(i) = i$ , 再  $merge$  操作中我们这样约定: 将并查集  $MF$  中的  $i$  和  $j$  所在的子集合并时, 把所有和  $i$  值相同的元素  $x$  和  $j$  值相同的元素  $y$  的  $f(x)$ 、 $f(y)$  全部设为  $\min(f(i), f(j))$ 。

实现代码如下:

```

int merge(int x, int y, MFset MF){
    if(MF->set[x] == MF->set[y]) return 0;
    int tmp;
    if(MF->set[x] < MF->set[y]){
        tmp = MF->set[y];
        for(int i=0; i<MAX_NUM; i++)
            if(MF->set[i] == tmp)

```



```

        MF->set[i] = MF->set[x];
    }
    else{
        tmp = MF->set[x];
        for(int i=0; i<MAX_NUM; i++)
            if(MF->set[i] == tmp)
                MF->set[i] = MF->set[y];
    }
    return 0;
}

```

### 3.5 Kruskal 算法的实现

Kruskal 算法是一种用来寻找最小生成树的算法。在剩下的所有未选取的边中，找最小边，如果和已选取的边构成回路，则放弃，选取次小边。下面是算法在本程序中的实现：

```

Graph kruskal(Graph G){
    if(!G) return NULL;
    // 初始化最小生成树
    Graph T = initial_mini_gTree(G);

    int from, to;
    // 创建并查集
    MFset MF = create_mfset(G->vexnum);

    // 由于边按照权降序，我们从最后一条边开始循环
    for(int i=G->edgenum-1; i>=0; i--){
        from = G->edges[i].from;
        to = G->edges[i].to;

        // 如果在同一个子集中，进行下一轮循环
        if(is_insame(from, to, MF))
            continue;
        // 否则也就是说他们不在同一个子集中
        // 那么将他们所在的子集合并
        else{

```

```
        add_edge(G->edges[i], T);
        merge(from, to, MF);
    }
}
delete_mfset(MF);
return T;
}
```

此算法根据输入的图  $G$ ，生成一个节点数相等但无边的图  $T$ ，这是初始化了的最小生成树。然后以输入的图  $G$  的节点数为大小创建一个并查集  $MF$ 。

接着从  $G$  的最小边（最后一条边）开始进行循环：如果边的两个节点在并查集中属于同一个子集，那么继续下一轮循环；如果不在，那么把这条边加入图  $T$ ，并把并查集中这两个节点所在的子集合并。

所有的边遍历完，程序结束，返回图  $T$ 。

### 3.6 主程序的测试代码

下面是主程序的测试代码：

```
int main(int argc, char const *argv[])
{
    // 如果有命令行参数，那就生成特定的一个测试用例
    if(argc==2)
        Graph G = create_test_graph();
    // 否则由用户来输入生成图
    else
        Graph G = create_graph();
    print_graph(G);
    Graph T = kruskal(G);
    printf("最小生成树是：\n");
    print_graph(T);
    delete_graph(G);
    delete_graph(T);
    return 0;
}
```

此程序首先根据用户输入或者读取文件创建一个图  $G$  并打印, 然后调用  $Graph\ kruskal(Graph\ G)$  函数, 该函数生成最小生成树  $T$  并打印。

### 3.7 具体的函数调用关系

具体函数的调用关系如下图所示:

