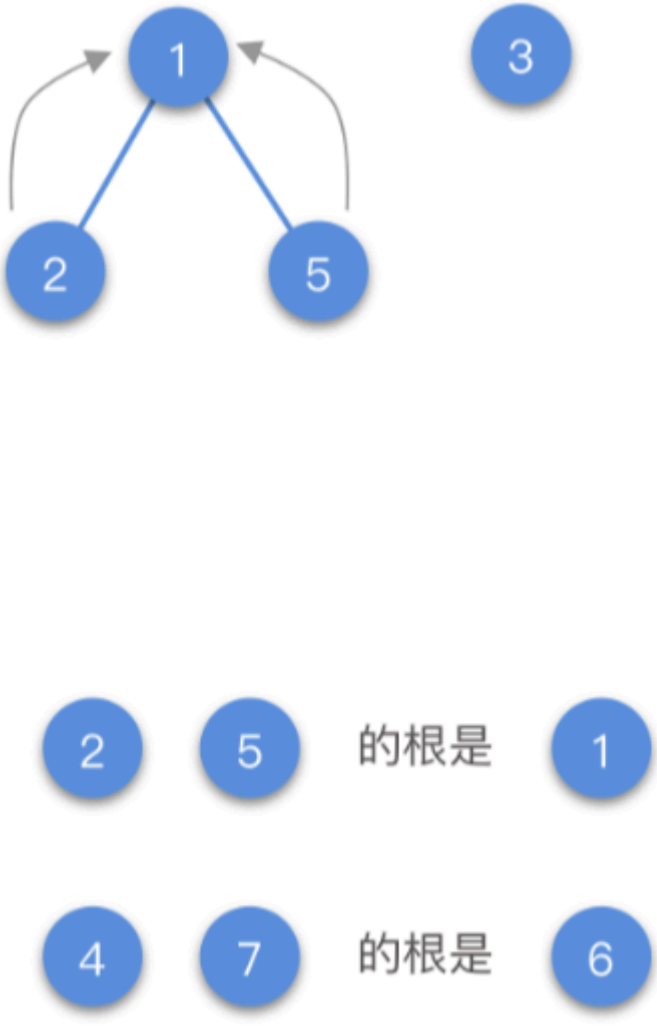
并查集与最小生成树

并查集介绍

并查集（Disjoint Set Union，简称 DSU 或 Union-Find）是一种用于处理一些不交集（Disjoint Set）的集合的数据结构，它支持两种主要的操作：

1. **Find**：确定某个元素属于哪个子集。它可以递归地遍历元素的父节点，直到找到一个根节点（即没有父节点的节点）。
2. **Union**：将两个子集合并成一个集合。这通常通过将一个集合的根节点链接到另一个集合的根节点来实现。

每一个集合都有一个代表元素，称为根。例如：



并查集的主要应用场景包括但不限于：

- 图的连通性问题，如判断图中是否有环。
- 网络流问题，如最小生成树（MST）算法中的Kruskal算法。
- 动态连通性问题，如网络中的动态连接和断开。

并查集的实现通常有以下几种优化方法：

- **路径压缩**：在查找操作中，将查找路径上的所有节点直接链接到根节点，以减少后续查找操作的深度。

- **按秩合并**（Union by Rank）：在合并两个集合时，较小的树连接到较大的树上，以避免树的高度变得过高，影响查找效率。

并查集表示：

```
typedef struct Edge{//并查集的边节点 front,to为边的指向，w代表权值
    int front;
    int to;
    int w;
}Edge;
```

找到所属根节点：

```
int father[maxsize] = {0};      //所属根节点

//查找根节点
int Find(int x){                //递归找到所属根节点
    if(father[x]<=0){           //找到了
        return x;
    }                           //递归查找根节点
    return father[x] = Find(father[x]);
}
```

按秩合并：

```
void Union(int x, int y){       //按秩合并两个并查集  秩：当前节点深度乘以-1
    int fx = Find(x);
    int fy = Find(y);
    if(fx == fy) return;        //两者属于同一个并查集，无需合并
    if(father[fx] < father[fy]){//fy深度 小于 fx深度 ，fy合并到fx中
        father[fy] = fx;
    }
    else{                       //否则 fx合并到 fy中
        if(father[fx] == father[fy]){   //秩相同时，合并深度加1 ，秩要减1，
            father[fy]--;
        }
        father[fx] = fy;
    }
}
```

对边进行排序：

实现从图到并查集的转换，同时使用冒泡排序边结构体进行排序

```
int SortEdge(int graph[][maxsize], int n, Edge edge[]){     //实现两个功能，将图转化为并查集表示，同时对并查集进行排序
    int  edgenum = 0;                   //边的个数
    for(int i = 0; i <= n; i++)         //无向图遍历上半矩阵即可
        for(int j = i+1; j <= n; j++)
            if(graph[i][j]){            //存在边，赋值
                edgenum++;
                edge[edgenum].front = i;
                edge[edgenum].to = j;
                edge[edgenum].w = graph[i][j];
```

```
            }
    //冒泡排序对边进行排序
    for(int i = edgenum; i > 1; i--){
        int flag = 0;
        for(int j =1; j < i; j++){
            if(edge[j].w > edge[j+1].w){     //递增排序   求最大生成树只需要改为递减排序即
可
                Edge temp = edge[j];
                edge[j] = edge[j+1];
                edge[j+1] = temp;
                flag = 1;                    //本趟进行了交换
            }
        }
        if(flag == 0){                       //该趟未排序，已经有序
            break;
        }
    }
    return edgenum;
}
```

上述代码基本上实现了一个并查集的各个功能，下面实现最小生成树算法

思路：

- kruskal算法实现基本步骤
- 对边进行排序
- 每次选取最小的边，该边的两个顶点是否属于同一个并查集
- 如果是，已经在最小生成树当中，如果不是，就加入，同时记录当前最小生成树权值之和
- **添加变量记录并查集个数以及遍历的边的下标**

代码：

```
//基础算法：求最小生成树
void Kruskal_MinTree(int graph[][maxsize], int n){
    Edge edge[maxsize];
    int edgenum = SortEdge(graph,n,edge);        //对当前图进行构造并查集，并排序
    int setnum = n;                              //连通分量个数，最开始为n个顶点，所以有
n个
    int ans = 0;
    while(setnum > 1){                           //最小生成树的算法实现
        int x = edge[k].front;
        int y = edge[k].to;
        int w = edge[k].w;                       //不断取最小的边，判断是否处于同一个并
查集，不同就加入
        if(Find(x) == Find(y)) continue;
        Union(x,y);                              //属于不同并查集，合并
        setnum--;
        ans += w;
        printf("%d--%d:%d ", x, y, w);
    }
    printf("最小生成树的权值为%d", ans);
}
```

全部代码： 实现最小生成树

```
//最小生成树算法

typedef struct Edge{//并查集的边节点 front,to为边的指向，w代表权值
    int front;
    int to;
    int w;
}Edge;

int father[maxsize] = {0};      //所属根节点

//查找根节点
int Find(int x){                //递归找到所属根节点
    if(father[x]<=0){           //找到了
        return x;
    }                           //递归查找根节点
    return father[x] = Find(father[x]);
}

//按秩合并
void Union(int x, int y){       //按秩合并两个并查集  秩：当前节点深度乘以-1
    int fx = Find(x);
    int fy = Find(y);
    if(fx == fy) return;        //两者属于同一个并查集，无需合并
    if(father[fx] < father[fy]){//fy深度 小于 fx深度 ，fy合并到fx中
        father[fy] = fx;
    }
    else{                       //否则 fx合并到 fy中
        if(father[fx] == father[fy]){   //秩相同时，合并深度加1 ，秩要减1，
            father[fy]--;
        }
        father[fx] = fy;
    }
}

//对边进行排序
    int SortEdge(int graph[][maxsize], int n, Edge edge[]){     //实现两个功能，将图转化为并查集表示，同时对并查集进行排序
        int  edgenum = 0;                   //边的个数
        for(int i = 0; i <= n; i++)         //无向图遍历上半矩阵即可
            for(int j = i+1; j <= n; j++)
                if(graph[i][j]){            //存在边，赋值
                    edgenum++;
                    edge[edgenum].front = i;
                    edge[edgenum].to = j;
                    edge[edgenum].w = graph[i][j];
                }
        //冒泡排序对边进行排序
        for(int i = edgenum; i > 1; i--){
            int flag = 0;
            for(int j =1; j < i; j++){
                if(edge[j].w > edge[j+1].w){     //递增排序  求最大生成树只需要改为递减排序即可
                    Edge temp = edge[j];
                    edge[j] = edge[j+1];
                    edge[j+1] = temp;
```

```
                    flag = 1;                       //本趟进行了交换
                }
            }
            if(flag == 0){                          //该趟未排序，已经有序
                break;
            }
        }
        return edgenum;
    }

//基础算法：求最小生成树
void Kruskal_MinTree(int graph[][maxsize], int n){
    Edge edge[maxsize];
    int edgenum = SortEdge(graph,n,edge);       //对当前图进行构造并查集，并排序
    int setnum = n;                             //连通分量个数，最开始为n个顶点，所以有
n个
    int k = 1;
    int ans = 0;
    while(setnum > 1){                          //最小生成树的算法实现
        int x = edge[k].front;
        int y = edge[k].to;
        int w = edge[k].w;                      //不断取最小的边，判断是否处于同一个并
查集，不同就加入
        k++;
        if(Find(x) == Find(y)) continue;
        Union(x,y);                             //属于不同并查集，合并
        setnum--;
        ans += w;
        printf("%d--%d:%d ", x, y, w);
    }
    printf("最小生成树的权值为%d", ans);
}
```