离散数学

项目说明文档

最小生成树

作者姓名: 高逸轩

学　　号: 2053385

指导教师: 唐剑锋

学院专业: 软件学院 软件工程

同济大学

Tongji University

1 功能简介

1.1 题目要求

用 Kruskal 算法求解给出图的最小生成树。

1.2 项目需求分析

本项目在实现的过程中，考虑并且满足了以下的需求：

- ✓ 健壮性
  当用户输入的数据不合理时，系统应当给予相应的提示而非直接报错。
- ✓ 执行效率高
  Kruskal 的贪心决策方式可以实现较低的复杂度对稀疏图求解最小生成树。
- ✓ 代码可读性高
  本项目在实现过程中，将代码根据功能的不同划分为了不同的代码块，便于阅读。

1.3 项目要求

首先由用户输入给出当前地图，求解给出图的最小生成树，同时保证健壮性。并利用 Kruskal 算法，保证运行效率。

1.3.1 功能要求

根据用户输入保存地图。然后将已得边进行排序进行贪心，借助并查集判别连通性并进行联通集的合并。

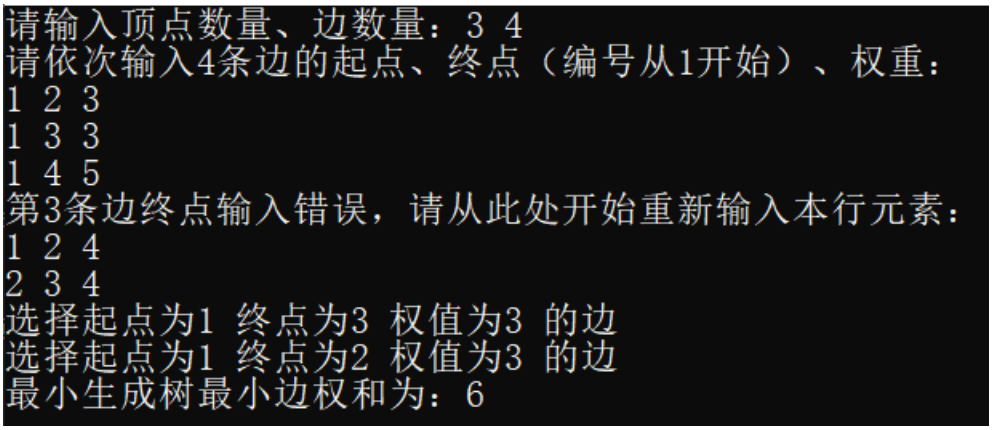
1.3.2 输入格式

以下内容利用 input_tools 函数集实现健壮性

```
顶点数目 vertexNumber   边数量 edgeNumber

FOR i:=1 to edgeNumber
{
    第 i 条边的起点、终点、权值
}
```

1.3.3 项目简单示例



2 项目实现

本项目核心部分：

- ✓ 健壮性实现
- ✓ 并查集功能实现
- ✓ 快速排序实现
- ✓ Kruskal 实现

下面将对本项目核心的部分进行介绍，部分代码如下：

```
// 邻接链表 边
struct Edge
{
    int src = -1;           // 起始点编号
    int dst = -1;           // 目标点序号
    int value = 0;          // 边长度权重
    bool operator<(const Edge& other) { return value < other.value; };// 重载小于比较运算符
    bool operator>(const Edge& other) { return value > other.value; };// 重载大于比较运算符
};

// 邻接链表 点
struct Vertex
{
    int ancestor;           // 记录在并查集中的祖先
};
```

```
// 邻接链表
struct GraphList
{
    // 构造函数
    GraphList(int n = 0, int m = 0);

    // 析构函数
    ~GraphList();

    // 构建图
    void SetGraphList();

    // 并查集寻父节点
    int FindAncestor(int i);

    // Kruskal算法求得最小生成树
    void Kruskal();

    int vertexNumber = 0, edgeNumber = 0; // 记录顶点数、边数
    Edge* edges = NULL;                   // 顶点数组
    Vertex* vertexs = NULL;               // 边数组
};
```

2.1 健壮性

通过 input_tools 函数集以及下列代码来生成对应错误提示，精确定位错误位置到每条边的起点、终点、权值，帮助用户得到错误输入处，并重新输入。以下为设置地图时，所对应的错误提示生成及调用代码：

```
FOR(i, 1, edgeNumber + 1)
{
    string errorTips = "第";
    errorTips += char(i + '0');
    errorTips += "条边起点输入错误，请从此处开始重新输入本行元素：";
    errorTips += '\n';                                          // 得到当前边 起点 输入错误对应提示
    edges[i].src = getint(1, vertexNumber, errorTips);

    errorTips = "";
    errorTips += "第";
    errorTips += char(i + '0');
    errorTips += "条边终点输入错误，请从此处开始重新输入本行元素：";
    errorTips += '\n';
    errorTips += char(edges[i].src + '0');
    errorTips += ' ';                                           // 得到当前边 终点 输入错误对应提示
    edges[i].dst = getint(1, vertexNumber, errorTips);

    errorTips = "";
    errorTips += "第";
    errorTips += char(i + '0');
    errorTips += "条边权值输入错误，请从此处开始重新输入本行元素：";
    errorTips += '\n';
    errorTips += char(edges[i].src + '0');
    errorTips += ' ';
    errorTips += char(edges[i].dst + '0');
    errorTips += ' ';                                           // 得到当前边 权值 输入错误对应提示
    edges[i].value = getint(1, INT_MAX, errorTips);

}
```

2.2 并查集

2.2.1 运算原理

以下为我通过课件与网络学习后得到的并查集概念：

并查集是一种树型的数据结构，用于处理一些不相交集合的合并及查询问题（即所谓的并、查）。比如说，我们可以用并查集来判断一个森林中有几棵树、某个节点是否属于某棵树等。

主要构成：

并查集主要由一个整型数组 pre[ ]和两个函数 find( )、join( )构成。

数组 pre[ ] 记录了每个点的前驱节点是谁，函数 find(x) 用于查找指定节点 x 属于哪个集合，函数 join(x,y) 用于合并两个节点 x 和 y 。

作用：

并查集的主要作用是求连通分支数（如果一个图中所有点都存在可达关系（直接或间接相连），则此图的连通分支数为 1；如果此图有两大子图各自全部可达，则此图的连通分支数为 2……）

2.2.2 前驱数组

pre[]数组我采取 Vertex 结构体实现，用以记录当前点的前驱节点编号。

```
// 邻接链表 点
struct Vertex
{
    int ancestor;          // 记录在并查集中的祖先
};
```

2.2.3 查找函数（路径压缩优化）

根据并查集的定义，可以简单得到查找函数的朴素递归写法：

```
int find(int x)
{
    while(pre[x] != x)
        x = pre[x];
    return x;
}
```

但在这种情况下，对于含有 n 个节点的并查集来说，每一次查询操作的时间复杂度最坏都是 O(n)的，对于大规模数据和多次查询的情况，若对图进行联通集数目的统计，时间复杂度可达 O(n^2)，这样的复杂度并不优秀。在这里，我们利用记忆化搜索的方式（在并查集中叫做路径压缩优化），可以对时间复杂度进行优化，代码实现如下：

```
// 并查集寻父节点，采用记忆化搜索
int GraphList::FindAncestor(int i)
{
    if (i == vertexs[i].ancestor) return i;
    return vertexs[i].ancestor = FindAncestor(vertexs[i].ancestor);
}
```

在这段代码中，我们将递归搜索先驱节点时所经历的点的先驱节点均进行了更新，即这条递归传递路线上所有的元素的父亲指针都会指向那个根节点了，也就完成了我们的设计。在这种情况下，如果第一次递归搜索时所用了 O(n)次，那么之后的每次查询，时间复杂度都会变为 O(1)，从而保证了在统计连通集数目时，不会对某一路径进行重复的递归搜索，严格保证时间复杂度为 O(n)。

2.2.4 合并函数

根据并查集定义，易得合并前驱节点的代码部分如下：

```
ans += edges[i].value;      // 树的边权之和更新
vertexs[f1].ancestor = f2; // f1的祖先设置为f2，两点属于同一个联通分支
```

2.3 快速排序

2.3.1 运算原理

根据网络资料，学习快速排序概念与优秀原因如下：

1、快速排序的基本思想：

快速排序使用**分治**的思想，通过一趟排序将待排序列分割成两部分，其中一部分记录的关键字均比另一部分记录的关键字小。之后分别对这两部分记录继续进行排序，递归地以达到整个序列有序的目的。

2、快速排序的三个步骤：

(1)选择基准：在待排序列中，按照某种方式挑出一个元素，作为 "基准"（pivot）

(2)分割操作：以该基准在序列中的实际位置，把序列分成两个子序列。此时，在基准左边的元素都比该基准小，在基准右边的元素都比基准大

(3)递归地对两个序列进行快速排序，直到序列为空或者只有一个元素。

3、选择基准的方式

对于分治算法，当每次划分时，算法若都能分成两个等长的子序列时，那么分治算法效率会达到最大。也就是说，基准的选择是很重要的。选择基准的方式决定了两个分割后两个子序列的长度，进而对整个算法的效率产生决定性影响。

2.3.2 时间复杂度分析与快速排序的优化

在最坏的情况下，仍可能是相邻的两个数进行了交换。因此快速排序的最差时间复杂度和冒泡排序是一样的都是 $O(n^2)$，平均的时间复杂度为 $O(nlogn)$. 根据课本讲解，选择序列第一个元素为比较基准，代码实现如下：

```
void QuickSort(int a[], int begin, int end)
{
    if (begin > end)
        return;
    int i = begin, j = end;
    while (i != j)
    {
        while (a[j] >= a[begin] && j > i)
            j--;
        while (a[i] <= a[begin] && j > i)
            i++;
        if (j > i)
            swap(a[i], a[j]);
    }
    swap(a[i], a[begin]);
    QuickSort(a, begin, i - 1);
    QuickSort(a, i + 1, end);
}
```

测试数据分析：如果输入序列是随机的，处理时间可以接受的。如果数组已经有序时，此时的分割就是一个非常不好的分割。因为每次划分只能使待排序序列减一，此时为最坏情况，快速排序沦为冒泡排序，时间复杂度为 $O(n^2)$。而且，考虑到输入的数据是有序或部分有序的情况是相当常见的。因此，使用第一个元素作为枢纽元是非常糟糕的，为了避免这个情况，我们更换了选择比较基准的元素，取中间元素为比较基准，实现优化：

```
// 类模板实现优化后的快速排序，排序区间为[l,r]
template<typename T>void QuickSort(T a[],int l,int r)
{
    if (l > r)
        return;
    T mid = a[(l + r) / 2];      // 取中间元素
    int i = l, j = r;
    while (i < j)
    {
        while (a[i] < mid) i++;  // 查找左半部分比中间数大的数
        while (a[j] > mid) j--;  // 查找右半部分比中间数小的数
        if (i <= j)              // 如果有一组不满足排序条件（左小右大）的数
        {
            swap(a[i], a[j]);    // 交换
            i++;
            j--;
        }
    }
    if (l < j) QuickSort(a, l, j);// 递归排序左半部分
    if (i < r) QuickSort(a, i, r);// 递归排序右半部分
}
```

2.4 Kruskal 求最小生成树

2.4.1 运算原理

Kruskal 是一种典型的贪心策略算法，其基本思想 是：按照权值从小到大的顺序选择 n-1 条边，并保证这 n-1 条边不构成回路。具体做法为：首先构造一个只含 n 个顶点的森林，然后依权值从小到大从连通网中选择边加入到森林中，并使森林中不产生回路，直至森林变成一棵树为止。

2.4.2 代码实现

根据以上运算过程，借助已经实现的并查集功能，实现代码如下：

```
// 实现Kruskal贪心算法构建最小生成树
void GraphList::Kruskal()
{
    // 快速排序，区间[1,edgeNumber]
    QuickSort(edges, 1, edgeNumber);

    // ans记录最小生成树边权之和，cnt记录已选边数量
    int ans = 0, cnt = 0;
    FOR(i, 1, edgeNumber + 1)
    {
        int f1 = FindAncestor(edges[i].src); // f1为边起点的祖先
        int f2 = FindAncestor(edges[i].dst); // f2为边终点的祖先

        // 若两点祖先不同，则分属不同的连通分支，需要合并，将此边加入最小生成树
        if (f1 != f2)
        {
            cnt++;                                  // 已选边数+1
            ans += edges[i].value;      // 树的边权之和更新
            vertexs[f1].ancestor = f2; // f1的祖先设置为f2，两点属于同一个联通分支
            cout << "选择起点为" << edges[i].src << " 终点为" << edges[i].dst << " 权值为" << edges[i].value << " 的边" << endl;
        }
        if (cnt == vertexNumber - 1)                    // 当最小生成树终已选边数=顶点数-1时，最小生成树构造完成
        {
            cout << "最小生成树最小边权和为：" << ans << endl;
            return;
        }
    }
    // 若未能选满vertexNumber-1条边，则不能构成最小生成树
    cout << "以上的边不能构成最小生成树" << endl;
}
```

2.4.3 Kruskal 与 Prim 的对比

在此题中，老师给出的源代码采用了 Prim 算法来实现最小生成树的求解。为此我学习了 Kruskal 和 Prim 算法的对比，并总结发表在自己的博客中，部分内容如下：

Prim 算法是从图中某一个顶点出发，寻找它相连的所有结点，比较这些结点的权值大小，然后连接权值最小的那个结点。然后将寻找这两个结点相连的所有结点，找到权值最小的连接。重复上一步，直到所有结点都连接上。

设图有 n 个点，m 条边。

易分析得到，Prim 算法对于每一个点都需要将全部点遍历一遍，以找到最近的点加入集合。故时间复杂度为 $O(n^2)$.

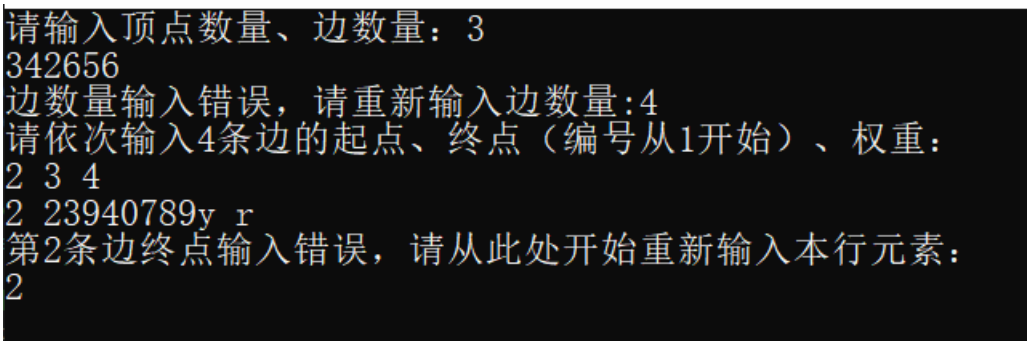
对 Kruskal 算法分析，对 m 条边的排序所花时间为 $O(mlogm)$。在建立最小生成树时，前面通过路径压缩的方法，已经将每次并查集操作可看为 O(1)，故建立最小生成树的时间复杂度为 $O(n)$。总时间复杂度为 $O(mlogm+n)$.

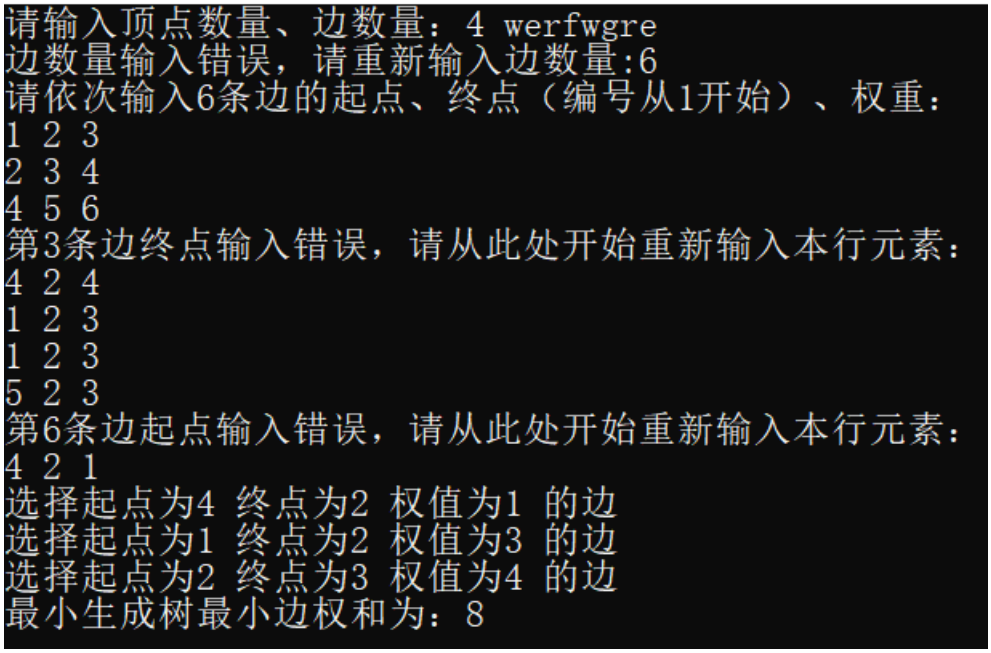
可以分析得到，当图为稠密图，边数接近n^2时，Prim 的复杂度为 $O(n^2)$要优秀于 Kruskal 的（n^2logn），所以 Prim 比 Kruskal 更适合于稠密图的求解。

当图为稀疏图时，边数远远少于n^2时，则 Kruskal 的复杂度优于 Prim。

3 项目测试

本题健壮性实现优秀，帮助用户自动补全已经输入的正确部分，仅需输入错误部分即可。示例如下：（2 为自动补全）





4 心得与总结

在本次最小生成树的作业中，我学习到了并查集及其优化的知识、快速排序的原理及其优化的知识、Kruskal 算法和 Prim 算法的对比。在这次作业中，我得到了很好的机会，来复习高中时信息学奥赛的知识，并且将“仅会应用”提升到了“深刻理解”的层次。并且，在不断搜索资料进行算法优化的过程中，我对自学能力的重要性有了更好的认识。另外，我对算法时间复杂度的分析能力也更上一层楼，可以判断在不同情况、极端情况时，某个算法的表现不同的原因，并对应给出了解决方法。