中南大学

CENTRAL SOUTH UNIVERSITY

算法分析与实验设计

实验报告（二）

学　　院：计算机学院

专业班级：

学生姓名：R

学　　号：

指导教师：

年 月 日

目录

一 Prim

1. 问题

给定一个无向图，自选一种数据存储结构，实现最小生成树计算的 Prim 算法和 Kruskal 算法，分析基于所选数据存储结构得到的算法复杂度。（100 分）

输入：
第一行输入两个数 m, n，表图示有 m 个顶点（所有顶点的字母各不相同），n 条边；
接下来 n 行每行输入两个顶点，一个值 w，表示这两个顶点之间有边连接，且边的权重为 w；

输出:
分别输出 Prim 算法和 Kruskal 算法得到的最小生成树（若某个节点存在最小生成树的多个选择，则按照字母表顺序进行优先选择，即输出只有一种结果）

例子：

输入：
3,3
A,B,2
A,C,5
B,C,4
输出：
Prim：
A,B,2
B,C,4
Kruskal:
A,B,2
B,C,4

得分标准：
（1） 按照要求实现 Prim 算法获得 40 分
（2） 按照要求实现 Kruskal 算法获得 40 分
（3） 实验报告和分析占 20 分

2. 代码

我用的都是 C 语言，请尽量使用 dev c++运行（而不是 vscode 等，有些题会报错）

```
#include <stdio.h>
#include <string.h>
#include <stdbool.h>
#define N 510
#define INF 0x3f3f3f3f

int n, m;
int g[N][N]; // 图的邻接矩阵
int dist[N]; // 存储每个节点到最小生成树的最短距离
bool st[N]; // 标记节点是否已经加入最小生成树

void prim() {
    char s1, s2;
    memset(dist, 0x3f, sizeof dist); // 初始化距离数组为无穷大
    for (int i = 0; i < n; i++) {
        int t = -1;
        for (int j = 1; j <= n; j++) {
            if (!st[j] && (t == -1 || dist[t] > dist[j])) {
                t = j; // 找到距离最小的节点
            }
        }
        if (i && dist[t] == INF) {
            return; // 如果存在无法连通的节点，说明图不连通，返回
        } else if (i) {
            for (int j = 1; j <= n; j++) {
                if (dist[t] == g[t][j]) {
                    s1 = 'A' + t - 1;
                    s2 = 'A' + j - 1;
                    printf("%c %c %d\n", s2, s1, g[t][j]); // 输出边的信息
                    break;
                }
            }
        }
        st[t] = true; // 将节点加入最小生成树
        for (int j = 1; j <= n; j++) {
            dist[j] = (dist[j] < g[t][j]) ? dist[j] : g[t][j]; // 更新最短距离
        }
    }
}
```

```
int main() {
	int a, b, c;
	char s1, s2;
	scanf("%d%d", &n, &m); // 读取节点数和边数
	memset(g, 0x3f, sizeof g); // 初始化邻接矩阵，表示无穷大距离
	while (m--) {
		scanf(" %c %c %d", &s1, &s2, &c); // 读取边的信息
		a = s1 - 'A' + 1; // 将节点转换为对应的索引
		b = s2 - 'A' + 1;
		g[a][b] = g[b][a] = (g[a][b] < c) ? g[a][b] : c; // 更新边的权重
	}

	prim(); // 执行 Prim 算法生成最小生成树

	return 0;
}
//该算法的时间复杂度为 O(n^2)
//空间复杂度为 O(n^2)
```

3. 分析

这段代码使用的是 Prim 算法来解决最小生成树问题。

Prim 算法是一种用于求解无向连通图的最小生成树（Minimum Spanning Tree）的算法。最小生成树是一个包含图中所有节点的树，且树的所有边的权重之和最小。

这段代码中的 Prim 算法的流程如下：

1、初始化阶段：

读取输入：从标准输入中读取节点数量 n 和边数量 m。

初始化邻接矩阵 g：将所有边的权重初始化为无穷大，表示没有直接连接的边。

初始化距离数组 dist：将所有节点到最小生成树的距离初始化为无穷大。

初始化标记数组 st：标记节点是否已经加入最小生成树，初始化为 false。

Prim 算法主循环：

外层循环：重复执行 n 次，每次选择一个节点加入最小生成树。

在未加入最小生成树的节点中，寻找距离最小生成树最近的节点 t。

内层循环：遍历所有节点，更新节点到最小生成树的距离。

如果节点未加入最小生成树且与节点 t 之间的边的权重小于当前距离数组中的值，则更新距离数组。

如果存在无法连通的节点，即最小生成树无法构建，则算法结束。

2、输出结果阶段：

遍历最小生成树中的节点，并输出节点与其父节点的边的信息。

Prim 算法通过每次选择当前与最小生成树距离最近的节点来逐步构建最小生成树，直到所有节点都加入到最小生成树中或者无法连通为止。代码中使用邻接矩阵来表示图，并使用距离数组和标记数组来辅助计算和记录节点的状态。输出阶段将最小生成树中的边信息打印出来。

算法复杂度分析：

初始化：O(n^2)

初始化邻接矩阵 g：O(n^2)
初始化距离数组 dist：O(n)
初始化标记数组 st：O(n)
Prim 算法主循环：O(n^2)

外层循环：执行 n 次，每次找到一个节点加入最小生成树，时间复杂度为 O(n)

内层循环：对于每个节点，寻找与之相连的节点的最小距离，时间复杂度为 O(n)

输出结果：O(n)

输出边的信息：每个节点最多与 n-1 条边相连，所以时间复杂度为 O(n)。

综上所述，该代码的总时间复杂度为 O(n^2)。

补充：代码中的邻接矩阵 g 存储了节点之间的边的权重，这里假设节点数 n 较小，因此矩阵的大小为 N x N，N 为一个较大的常数（例如 510）。在实际应用中，如果节点数较大，邻接矩阵的大小将会很大，会导致算法的空间复杂度增加。可以考虑使用邻接表等其他数据结构来优化空间占用。

4. 测试

图 1：按样例输入

图 2：随机测试

二 Kruskal

1. 问题

给定一个无向图，自选一种数据存储结构，实现最小生成树计算的 Prim 算法和 Kruskal 算法，分析基于所选数据存储结构得到的算法复杂度。（100 分）

输入：
第一行输入两个数 m, n，表图示有 m 个顶点（所有顶点的字母各不相同），n 条边；
接下来 n 行每行输入两个顶点，一个值 w，表示这两个顶点之间有边连接，且边的权重为 w；

输出:
分别输出 Prim 算法和 Kruskal 算法得到的最小生成树（若某个节点存在最小生成树的多个选择，则按照字母表顺序进行优先选择，即输出只有一种结果）

例子：

输入：
3,3
A,B,2
A,C,5
B,C,4
输出：
Prim：
A,B,2
B,C,4
Kruskal:
A,B,2
B,C,4

得分标准：
（4） 按照要求实现 Prim 算法获得 40 分
（5） 按照要求实现 Kruskal 算法获得 40 分
（6） 实验报告和分析占 20 分

2. 代码

```
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
#define N 100010
#define M 200010
#define INF 0x3f3f3f3f

int p[N]; // 并查集数组，用于存储每个顶点的父节点
int n, m; // 顶点数和边数

struct Edge{
    int a, b, w; // 边的起始顶点、结束顶点和权重
};

Edge e[M]; // 边的数组，用于存储边的信息

// 比较函数，用于边的排序
int cmp(const void *a, const void *b){
    struct Edge* x = (struct Edge*)a;
    struct Edge* y = (struct Edge*)b;
    return x->w - y->w;
}

// 并查集的查找操作
int find(int x){
    if(x != p[x])
        p[x] = find(p[x]); // 路径压缩
    return p[x];
}

// Kruskal 算法实现
void kru(){
    char s1, s2;
    qsort(e, m, sizeof(Edge), cmp); // 对边按权重进行排序

    for(int i = 1; i <= n; i++)
        p[i] = i; // 初始化并查集数组，每个顶点自成一个连通分量

    int cnt = 0; // 计数器，记录已添加到最小生成树的边数

    for(int i = 0; i < m; i++){
```

```
            int a = e[i].a, b = e[i].b, w = e[i].w; // 当前遍历的边的起始顶点、结束顶点和权重

            a = find(a); // 查找顶点 a 的父节点
            b = find(b); // 查找顶点 b 的父节点

            if(a != b){ // 顶点 a 和 b 不属于同一个连通分量，不会形成环
                p[a] = b; // 将顶点 a 的父节点更新为顶点 b 的父节点，表示合并连通分量

                s1 = 'A' + a - 1; // 起始顶点转换为对应的字母表示
                s2 = 'A' + b - 1; // 结束顶点转换为对应的字母表示
                printf("%c %c %d\n", s1, s2, w); // 打印边的信息
                cnt++; // 边数加 1
            }
        }
}

int main(){
    int a, b, c;
    char s1, s2;
    scanf("%d %d", &n, &m); // 读取顶点数和边数

    for(int i = 0; i < m; i++){
        scanf(" %c %c %d", &s1, &s2, &c); // 读取每条边的起始顶点、结束顶点和权重
        a = s1 - 'A' + 1; // 起始顶点转换为对应的整数值
        b = s2 - 'A' + 1; // 结束顶点转换为对应的整数值
        e[i].a = a;
        e[i].b = b;
        e[i].w = c;
    }

    kru(); // 执行 Kruskal 算法找到最小生成树

    return 0;
}
//对边按照权值从小到大排序的时间复杂度为 O(mlogm)，其中 m 为边的数量；
//初始化并查集的时间复杂度为 O(n)，其中 n 为点的数量；
//遍历每条边的时间复杂度为 O(m)，在并查集中查找每个点的父节点的时间复杂度为 O(logn)，因此总时间复杂度为 O(mlogn)；
//当输出的边数达到 n-1 时，停止遍历，因此最多需要遍历 m 条边；
//空间复杂度主要取决于数组 e 和并查集 p 的大小，因此为 O(m+n)。
```

3. 分析

这段代码使用了 Kruskal 算法来找到图的最小生成树，具体实现如下：

1.读取输入的顶点数 n 和边数 m。
2.创建一个结构体数组 e 来存储边的信息，数组大小为 M，即 200010。每条边包括起始顶点 a、结束顶点 b 和权重 w。
3.通过循环读取输入，将每条边的起始顶点、结束顶点和权重存储到数组 e 中。
4.调用 kru 函数，该函数实现了 Kruskal 算法。
5.在 kru 函数中，首先对边数组 e 进行排序，使用 qsort 函数和自定义的比较函数 cmp，按照边的权重从小到大进行排序。
6.创建一个大小为 N 的并查集数组 p，用于存储每个顶点的父节点。初始时，将每个顶点自身作为父节点。
7.使用循环遍历每条边，从权重最小的边开始。
8.对于当前遍历的边，获取其起始顶点 a 和结束顶点 b。
9.使用 find 函数查找顶点 a 和 b 的父节点，并进行路径压缩。
10.如果顶点 a 和 b 的父节点不同，说明它们属于不同的连通分量，不会形成环。
11.将顶点 a 的父节点更新为顶点 b 的父节点，表示将两个连通分量合并。
12.打印边的信息，包括起始顶点、结束顶点和权重。
13.增加一个计数器 cnt，表示已添加到最小生成树中的边的数量。
14.循环继续，直到遍历完所有的边或者找到了 n-1 条边（最小生成树的边数为 n-1）。
15.最后，kru 函数结束，程序打印出最小生成树的边。

整体上，该程序首先对边进行排序，然后使用并查集数据结构进行判断和合并操作，最终找到最小生成树。

复杂度分析：

1：读取输入的顶点数 n 和边数 m 的时间复杂度为 O(1)。
2：创建并初始化边的结构体数组 e 的时间复杂度为 O(m)。
3：使用 qsort 函数对边进行排序的时间复杂度为 O(m log m)。
4：初始化并查集数组 p 的时间复杂度为 O(n)。
5：在 kru 函数中的循环中，遍历边的数量 m，时间复杂度为 O(m)。
6：在循环中，使用 find 函数进行并查集的查找和路径压缩，时间复杂度近似为 O(1)。
7：在循环中进行判断和合并操作的时间复杂度也近似为 O(1)。

打印边的信息的时间复杂度可以忽略不计。综上所述，代码的时间复杂度主要由排序边的步骤决定，为 O(m log m)，其中 m 是边的数量。并查集的操作时间复杂度可以近似看作 O(1)。空间复杂度方面，除了存储图的数据结构外，代码额外使用了一个大小为 N 的数组来表示并查集，因此空间复杂度为 O(N)。

需要注意的是，这个时间复杂度分析是基于对边进行排序的情况下。如果输入的边已经是预先排好序的，或者图是一个稀疏图（边的数量相对较少），那么算法的时间复杂度可能会更好。

4. 测试

图 3：按样例测试

图 4：随机进行测试