中国科学院大学
University of Chinese Academy of Sciences

数据结构与算法分析实验报告

最小成本搭建通信网络

姓名 徐昕妍

学号 2023K8009970008

班级 2311 班

2025 年 5 月 31 日

目录

1 实验内容

1.1 实验内容

设计一个完整的求解程序，包括程序输入、打印无向连通图、输出等功能。给定 n 个城市，以及城市间的网络连接代价，求解连接所有城市的最小费用。

1. 构建一个模拟城市网络连接的带权无向图，顶点表示城市，边表示两个城市之间的网络线路，边的权值表示城市间网络连接成本。
2. 基于最小生成树实现城市间最小成本网络连接。
3. 详细阐述算法的每一步执行过程，分析算法的适用条件和局限性。

1.2 实验目的

1. 理解带权无向图的构建及表示方法。
2. 掌握最小生成树（Minimum Spanning Tree, MST）算法的设计和实现，重点是 Kruskal 或 Prim 算法。
3. 学会如何求解连接所有城市的最小费用网络问题。
4. 分析最小生成树算法的步骤、适用条件和局限性。

2 实验过程

2.1 最小生成树算法

生成树

一个无向连通图的生成树是包含图中所有顶点的极小连通子图（删去任何一条边都会使其不连通）。若图有 n 个顶点，生成树包含在这 n 个顶点并恰好有 n - 1 条边。[1]

最小生成树

最小生成树就是所有生成树中边权之和最小的那一棵（或几棵）。

最小生成树算法主要有两种：**Prim 算法**和 **Kruskal 算法**。

2.1.1 Prim 算法

此算法可以称为“加点法”，每次迭代选择代价最小的边对应的点，加入到最小生成树中。算法从某一个顶点 s 开始，逐渐长大覆盖整个连通网的所有顶点。

算法步骤如下：

1. 初始化：

   任选一个起始节点（通常选节点 0），加入集合 MST。

   初始化优先队列（最小堆），存储所有连接 MST 与非 MST 节点的边。

2. 循环扩展：

   从堆顶取出当前最小权重边 (u, v, w)，其中 $u \in MST, v \notin MST$。

   将 v 加入 MST，边 (u, v) 加入结果集。

   将 v 的所有邻接边（连接未加入节点）加入堆。

3. 终止条件：

   当所有节点均加入 MST 时结束（共选 V-1 条边）。

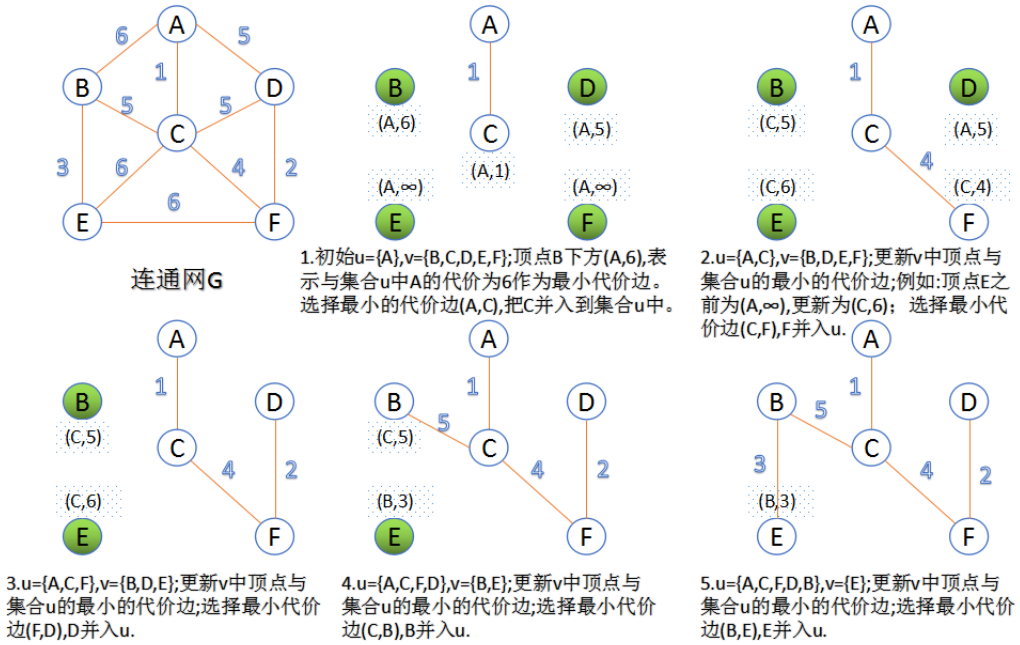


图 1: Prim 算法步骤图 [2]

伪代码如下所示：

```
函数 PRIM(G, w, s):
    初始化:
        for each vertex v ∈ V:
            key[v] ← ∞          // 记录连接到v的最小边权
            π[v] ← NIL          // 记录v在MST中的父节点
        key[s] ← 0              // 起始节点权值为0

        Q ← V                   // 优先队列（最小堆），按key[v]排序

    while Q ≠ ∅:
        u ← EXTRACT-MIN(Q)    // 取出key最小的节点u
        for each v ∈ G.Adj[u]: // 遍历u的所有邻接节点
            if v ∈ Q 且 w(u,v) < key[v]:
                π[v] ← u     // 更新父节点
                key[v] ← w(u,v) // 更新最小边权
                DECREASE-KEY(Q, v) // 调整堆

    // 构建MST边集合
    MST ← ∅
    for each v ∈ V - {s}:
        MST ← MST ∪ {(π[v], v, w(π[v],v))}

    return MST
```

图 2: Prim 算法伪代码

2.1.2 Kruskal 算法

此算法可以称为 “加边法”，初始最小生成树边数为 0，每迭代一次就选择一条满足条件的最小代价边，加入到最小生成树的边集合里。

算法步骤如下：

1. 初始化：

   将每个顶点视为独立的连通分量（使用并查集实现）。

   将所有边按权重升序排序。

2. 遍历边并构建 MST：

   对于每条边 (u, v, w)，检查 u 和 v 是否属于同一连通分量：

   若不属于：将边加入 MST，并合并 u 和 v 的连通分量。

   若属于：跳过（避免形成环）。

3. 终止条件：

   已选中 V-1 条边，或所有边处理完毕

原理：

因为边是由小到大进行排序的，所以先考虑的边一定比后考虑的边的边权要小，所以如果两个点有两种方式相连，先枚举到的方式一定比后枚举到的方式要优，所以不需要考虑链接两个点的多种方式中到底哪个是最优解。[3]

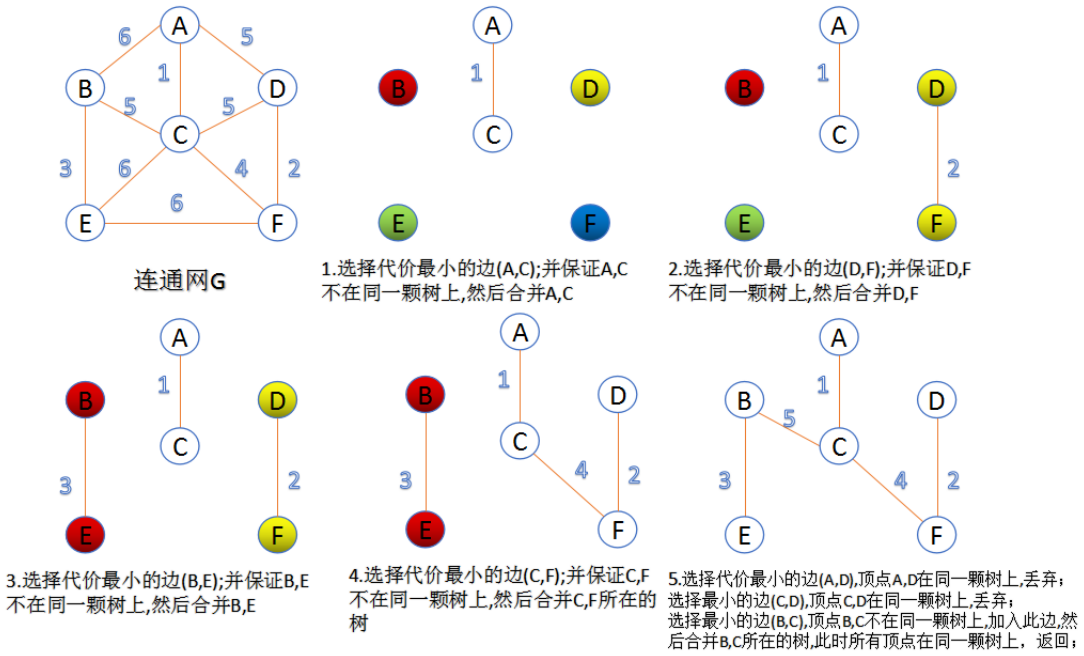


图 3: Kruskal 算法步骤图

伪代码如下：

```
KRUSKAL(G, w):
    MST ← ∅
    for each vertex v ∈ G.V:
        MAKE-SET(v)                 // 初始化并查集
    sort G.E by w(u,v)              // 按边权升序排序
    for each (u, v) in G.E:         // 遍历所有边
        if FIND-SET(u) ≠ FIND-SET(v):
            MST ← MST ∪ {(u, v)}
            UNION(u, v)             // 合并连通分量
    return MST
```

图 4: Kruskal 算法伪代码

2.2 实验代码

本次实验采用了 Kruskal 算法构造最小生成树。

2.2.1 数据存储结构

```
struct Edge {
    int city1, city2;//两个顶点
    float cost;//权重
};
```

主要存储了边的两个顶点（两个城市编号）和边的权重。

2.2.2 并查集查找

```
//并查集查找父亲结点
int find(int parent[], int i) {
    if (parent[i] == i)
        return i;
    return find(parent, parent[i]);
}
```

find 函数用于查找节点 i 的根节点，是并查集的核心操作，主要解决以下问题：

- 判断两个节点是否属于同一集合：通过比较它们的根节点。
- 路径压缩优化：在查找过程中扁平化树结构，加速后续查询。

```
if (parent[i] == i)
```

作用是检查当前节点 i 是否为根节点（即集合的代表元）。如果是根节点，直接返回 i；否则继续递归查找 i 的父节点的根节点。

2.2.3 Kruskal 算法实现

以下为核心代码：

```
void kruskal(struct Edge graph[], int n, int edge_count) {
    //按边权排序
    for (int i = 0; i < edge_count - 1; i++) {
        for (int j = 0; j < edge_count - i - 1; j++) {
            if (graph[j].cost > graph[j + 1].cost) {
                struct Edge temp = graph[j];
                graph[j] = graph[j + 1];
                graph[j + 1] = temp;
            }
        }
    }

    int *parent = (int *)malloc(n * sizeof(int));
    for (int i = 0; i < n; i++)
        parent[i] = i; //事先把每个点的父亲初始化为它自己

    printf("\nMinimum Cost Connection Plan:\n");
    float total_cost = 0;

```

```
    for (int i = 0; i < edge_count; i++) {
        int root1 = find(parent, graph[i].city1);
        int root2 = find(parent, graph[i].city2);

        if (root1 != root2) {
            printf("City %d -- City %d : Cost %.2f\n",
                   graph[i].city1, graph[i].city2, graph[i].cost);
            total_cost += graph[i].cost;
            parent[root1] = root2;
        }
    }
    printf("Total Cost: %.2f\n", total_cost);
    free(parent);
}
```

首先采用冒泡排序将边按权重排序。

```
//按边权排序
    for (int i = 0; i < edge_count - 1; i++) {
        for (int j = 0; j < edge_count - i - 1; j++) {
            if (graph[j].cost > graph[j + 1].cost) {
                struct Edge temp = graph[j];
                graph[j] = graph[j + 1];
                graph[j + 1] = temp;
            }
        }
    }
```

再对各顶点的父亲顶点进行初始化。必须将每个点的父亲顶点初始化为它自己，否则默认每个父亲顶点都是 0，会出现错误。

```
int *parent = (int *)malloc(n * sizeof(int));
    for (int i = 0; i < n; i++)
        parent[i] = i; //事先把每个点的父亲初始化为它自己
```

这一部分代码主要对应 MST 的构建。我们需要引入并查集来判断两个点是否已经联通。

```
for (int i = 0; i < edge_count; i++) {
        int root1 = find(parent, graph[i].city1);
        int root2 = find(parent, graph[i].city2);

        if (root1 != root2) {
```

```
            printf("City %d -- City %d : Cost %.2f\n",
                    graph[i].city1, graph[i].city2, graph[i].cost);
            total_cost += graph[i].cost;
            parent[root1] = root2;
        }
    }
```

并查集是一种用于处理动态连通性问题的数据结构，能够高效地支持查找和合并两种操作。

- 查找：判断某个元素属于哪个集合（找到该元素所属集合的代表元，通常称为“根”或“父节点”）。
- 合并：将两个元素所在的集合合并成一个集合。

并查集的核心思想是**将每个集合用一棵树表示，树的根节点作为集合的代表元素**。

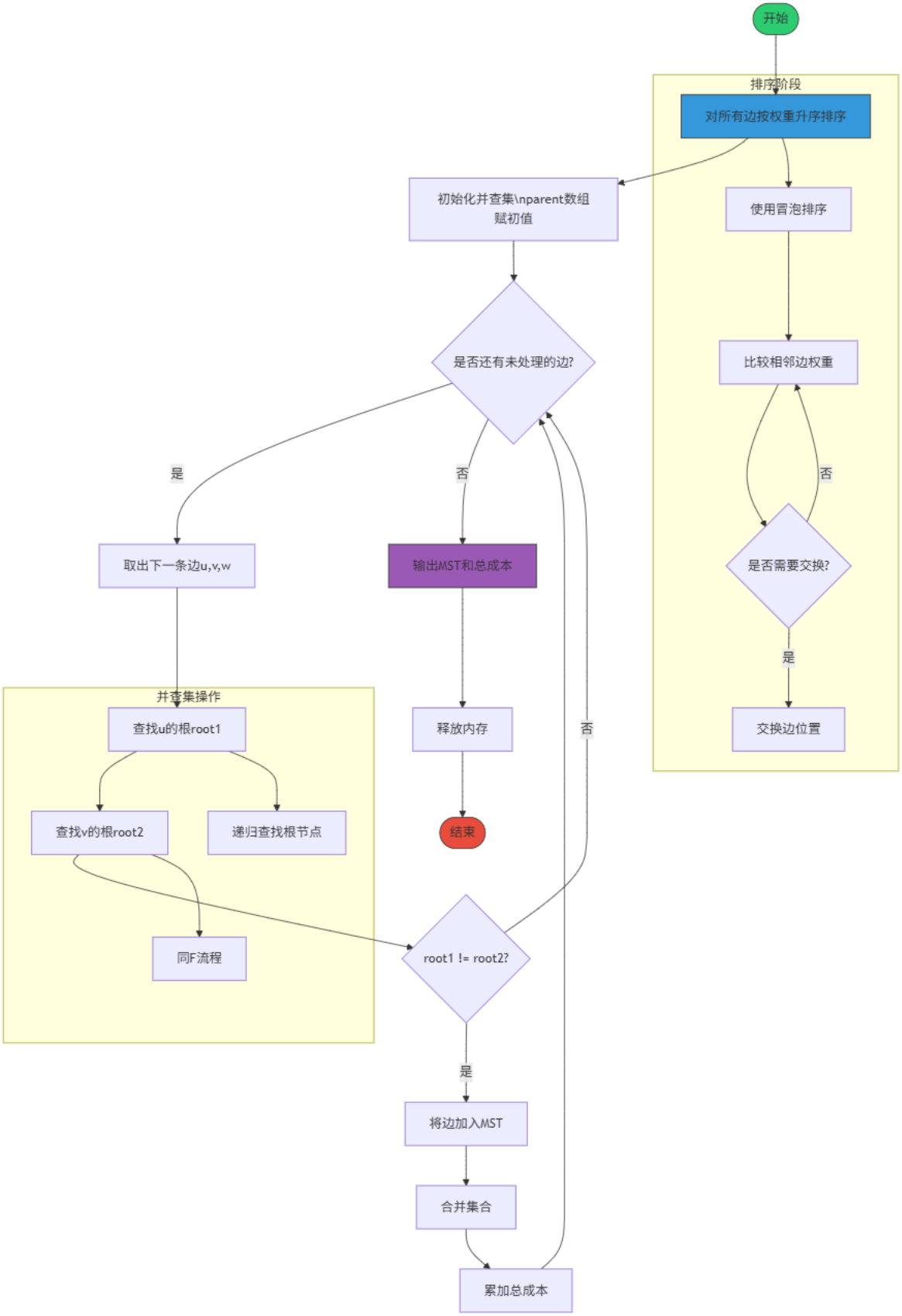
- 每个节点指向它的父节点，根节点的父节点指向自己。
- 查找操作就是不断沿父节点指针向上，直到找到根节点。
- 合并操作就是把一个集合的根节点指向另一个集合的根节点。

因此我们可以将联通的两个顶点合并入一个并查集里，它们将拥有相同的根节点；而两个顶点是否已经联通也可以通过它们是否拥有相同的根节点来判断。

如果两个顶点的根节点并不相同，证明它们并不联通，将这两个顶点加入最小生成树，计算目前的总权重，并将它们合并到同一个并查集，代表它们现在已经联通。不断循环操作，直至遍历所有边。

最终我们将得到所有顶点的最小生成树及总权重，即连接所有城市的最小费用。

Kruskal 算法流程图如下图所示：


开始
排序阶段
对所有边按权重升序排序
使用冒泡排序
比较相邻边权重
是否需要交换?
是
否
交换边位置
初始化并查集\nparent数组
赋初值
是否还有未处理的边?
是
否
取出下一条边u,v,w
并查集操作
查找u的根root1
查找v的根root2
递归查找根节点
同F流程
root1 != root2?
是
否
将边加入MST
合并集合
累加总成本
输出MST和总成本
释放内存
结束

2.2.4 实验结果

测试样例如下：

```
Enter number of cities: 8
Enter connection costs (format: city1 city2 cost), enter -1 -1 -1 to finish:
0 1 15
0 2 20
0 3 10
1 4 25
2 5 30
3 6 12
4 5 8
5 6 18
6 7 22
1 7 35
2 3 5
-1 -1 -1
```

最终输出结果如下所示：

```
Minimum Cost Connection Plan:
City 2 – City 3 : Cost 5.00
City 4 – City 5 : Cost 8.00
City 0 – City 3 : Cost 10.00
City 3 – City 6 : Cost 12.00
City 0 – City 1 : Cost 15.00
City 5 – City 6 : Cost 18.00
City 6 – City 7 : Cost 22.00
Total Cost: 90.00
```

为了方便表示最小生成树的图，我们也输出了它的邻接矩阵。如下所示：

```
Adjacency Matrix:
    0      1      2      3      4      5      6      7
0   0.00   15.00  20.00  10.00  0.00   0.00   0.00   0.00
1   15.00  0.00   0.00   0.00   25.00  0.00   0.00   35.00
2   20.00  0.00   0.00   5.00   0.00   30.00  0.00   0.00
3   10.00  0.00   5.00   0.00   0.00   0.00   12.00  0.00
4   0.00   25.00  0.00   0.00   0.00   8.00   0.00   0.00
5   0.00   0.00   30.00  0.00   8.00   0.00   18.00  0.00
6   0.00   0.00   0.00   12.00  0.00   18.00  0.00   22.00
7   0.00   35.00  0.00   0.00   0.00   0.00   22.00  0.00
```

图 6: 邻接矩阵

使用 Python 绘出原图和最小生成树图，如下图所示：

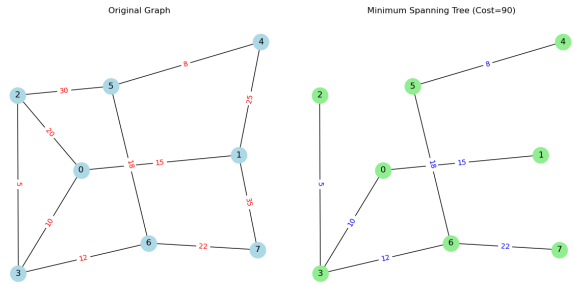


图 7: 结果对比图

2.3 讨论与分析

2.3.1 适用条件

1. 无向连通图。Kruskal 算法仅适用于无向图，因为最小生成树的定义基于无向边。图必须是连通的，否则无法生成包含所有顶点的树，而是会得到最小生成森林。

2. 非负边权重。边权重应为非负数（如距离、成本），负权边可能导致算法失效（但可通过偏移修正）。边权重可以相等。

3. 稀疏图优势：在边数 E 远小于顶点数 V^2 时，效率优于 Prim 算法。

2.3.2 局限性

1. 性能局限性。Kruskal 算法的时间复杂度主要取决于排序步骤。对于稠密图，时间复杂度接近 $O(V^2logV)$，不如 Prim 算法的 $O(V^2)$ 高效。

2. 功能局限性。该算法适合适合边已预先给定的场景，无法处理动态图：如果图需要频繁插入/删除边，需重新排序和计算，效率低下。

   同时也不支持有向图，无法直接处理有向图的最小树形图问题

3. 内存占用大。需要存储所有边，空间复杂度 O(E)，对于超大规模图可能受限。

2.3.3 与 Prim 算法的对比

表 1: Kruskal 算法与 Prim 算法对比

对比维度	Kruskal 算法	Prim 算法
核心策略	按边权排序 + 并查集合并	按顶点扩展 + 优先队列
时间复杂度	$O(E \log E)$	$O(E \log V)$
空间复杂度	$O(E)$	$O(V)$
适用图类型	无向连通图	无向连通图
稀疏图性能	优（$E \sim O(V)$ 时）	中（堆操作开销）
稠密图性能	劣（排序代价高）	优（$O(V^2)$ 可优化）
动态图支持	不适用	支持（Fibonacci 堆优化）
负权边处理	支持非负权	支持非负权
实现复杂度	中等（需并查集）	简单（仅需优先队列）
典型应用场景	通信网络布线、离线计算	实时路径规划、图像分割

3 实验总结

本次实验表面上是建立城市间网络连接的最小费用问题，实际上是最小生成树的构建问题。在本次实验中我通过查阅资料，了解了两种最小生成树的算法：Kruskal 算法和 Prim 算法，并且动手实现了 Kruskal 算法，并通过该算法解决了建立城市间网络连接的最小费用问题。

在实现 Kruskal 算法的过程中，我接触到了一个新的概念：并查集。并查集可以很轻松地处理网络连接问题。引入并查集可以方便对顶点连通的判断。通过实现并查集数据结构及其查找、合并操作，我们高效地解决了边的连通性判断问题，避免了环路的产生，这正是 Kruskal 算法巧妙之处。

总这次实验加强了我对最小生成树的理解。最小生成树算法作为图论中的经典问题，有着广泛的实际应用。希望在未来的实验中我可以逐步实现以可视化的方式展示城市网络连接并自动计算展示最佳规划。

参考文献

[1] https://blog.csdn.net/2301_80361697/article/details/148255400. Accessed: 2025-5-31.

[2] https://blog.csdn.net/a2392008643/article/details/81781766. Accessed: 2025-5-31.

[3] https://blog.csdn.net/Floatiy/article/details/79424763. Accessed: 2025-5-31.