同济大学

TONGJI UNIVERSITY

离散数学课程设计

项目名称	最小生成树
学　　院	计算机科学与技术学院
专　　业	软件工程
学生姓名	杨瑞晨
学　　号	2351050
指导教师	唐剑锋
日　　期	2024 年 12 月 1 日

目 录

1 项目分析

1.1 项目背景

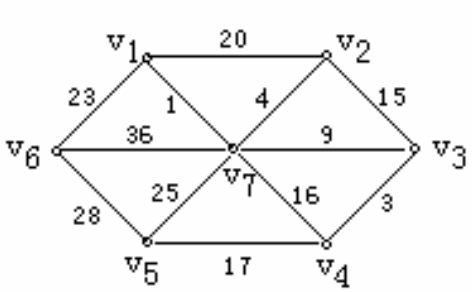
在网络设计和连接问题中，最小生成树（Minimum Spanning Tree, MST）是一个关键概念，它涉及到如何在加权连通图中找到一棵连接所有顶点的树，且这棵树的总权重（即所有边的权重之和）最小。这样的树可以被视为网络布线成本的最小化，其中顶点代表网络节点（如城市、计算机等），边的权重代表连接这些节点的成本（如距离、电缆长度等）。最小生成树问题在多个领域都有广泛的应用，包括网络设计、电路设计、供水系统规划等。

具体来说，最小生成树问题可以描述为：给定一个连通的加权图，其包含若干顶点和边，以及每条边的非负权重，求出一棵包含所有顶点的树，使得这棵树的边权重之和最小，同时确保任意两个顶点之间都连通。这个问题的解决方案对于优化资源分配、降低成本和提高网络效率至关重要。

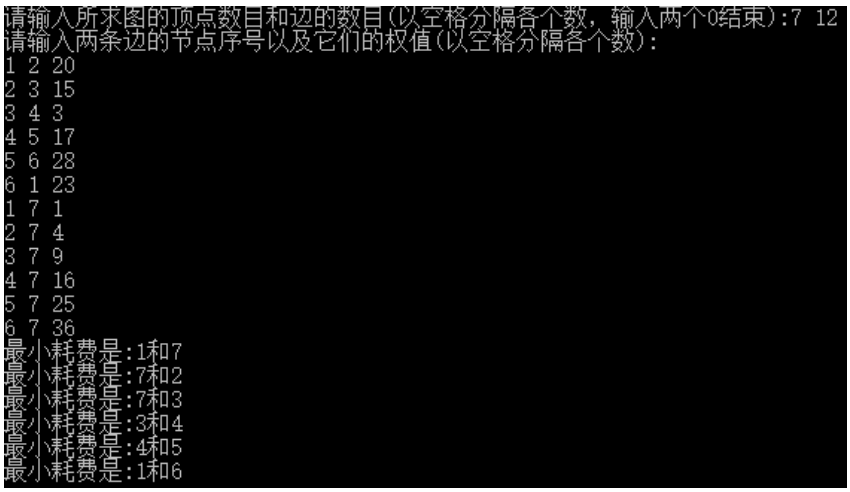
1.2 项目要求

本项目的目标是实现两种经典的最小生成树算法：Prim 算法和 Kruskal 算法。给定一个赋权图，表示为若干城市之间的直接通信道路造价，目标是提供一个设计方案，使得所有城市能够保持通信，同时使得总造价最小，并计算出这一最小值。

如下图所示的赋权图表示某七个城市，预先计算出它们之间的一些直接通信道路造价（单位：万元），试给出一个设计方案，使得各城市之间既能够保持通信，又使得总造价最小，并计算其最小值。



1.3 项目示例



1.4 项目环境

使用 C++ 语言实现，开发环境为 Linux 下的 gcc 编译器。

2 项目设计

2.1 数据结构应用

在最小生成树项目中，选择合适的数据结构对于算法的性能至关重要。以下是项目中使用的关键数据结构及其分析：

2.1.1 边 (Edge) 类

在最小生成树问题中，图由顶点和连接顶点的边组成。每条边都有一个与之相关的权重，表示连接的成本或距离。边的表示需要能够快速比较两条边的权重，以便在 Kruskal 算法中对边进行排序。

Edge 类用于表示图中的边，包含两个顶点的索引（u 和 v）和一个权重（weight）。提供了一个小于运算符的重载，使得边可以根据权重进行排序，这对于 Kruskal 算法中的边排序至关重要。

2.1.2 并查集 (UnionFind) 类

Kruskal 算法需要检测图中是否存在环，这可以通过并查集数据结构高效实现。并查集支持两个主要操作：查找（确定元素所属的集合）和合并（合并两个集合）。

UnionFind 类用于管理集合的合并和查找操作，内部使用一个数组 parent 来存储每个元素的父节点。

- find 方法用于找到元素的根节点，通过路径压缩优化，使得后续的查找操作更加高效。
- unite 方法用于合并两个元素所属的集合，通过将一个集合的根节点指向另一个集合的根节点实现。
- connected 方法用于检查两个元素是否属于同一个集合，用于 Kruskal 算法中检测环。

2.1.3 图 (Graph) 类

图的表示需要能够存储所有顶点和边的信息，并支持对边的增删查改操作。

对于 Prim 算法，需要快速访问每个顶点的邻接顶点和边的权重，因此使用邻接表表示图；

对于 Kruskal 算法，需要快速访问所有边的权重，因此使用边的列表表示图。

Graph 类是项目的中心数据结构，包含图中顶点的数量（V），边的列表（edges）和邻接表（adjList）。

- edges 用于 Kruskal 算法，存储图中所有边的对象，可以快速访问每条边的权重和顶点。
- adjList 用于 Prim 算法，是一个二维向量，每个顶点对应一个包含其所有邻接顶点和边权重的列表。
- addEdge 方法允许向图中添加边，同时更新 edges 和 adjList。
- kruskalMST 和 primMST 方法分别实现了 Kruskal 和 Prim 算法，用于计算最小生成树。

2.1.4 std::priority_queue

在 Prim 算法中，需要一个能够自动排序的队列，用于存储每个顶点的邻接边，并选择权重最小的边。

优先队列是一个能够自动排序的队列，它能够根据元素的优先级自动调整元素的顺序。在霍夫曼编码中，我们需要一个能够快速找到频率最小的节点的数据结构，我们可以将霍夫曼树的节点放入优先队列中，队列会自动根据节点的频率排序，从而保证每次取出的节点都是频率最小的。以下是优先队列的一些关键特性：

（1）元素优先级：每个元素都有一个优先级，通常使用数值表示，数值越小优先级越高。

（2）出队顺序：元素出队时，总是优先级最高的元素先出队。

（3）入队操作：元素可以按照任意顺序入队。

（4）动态性：优先队列可以在运行时动态地添加和移除元素。

（5）无序性：优先队列内部元素的存储通常是无序的，即元素的存储顺序并不反映它们的优先级顺序。

2.2 算法设计

2.2.1 算法思路

Kruskal **算法**： Kruskal 算法是一种贪心算法，用于寻找加权连通图的最小生成树。该算法通过将边按权重排序，然后逐个考虑这些边，如果加入边不会形成环，则将其加入到生成树中。

算法 1 Kruskal 算法求最小生成树

输入：加权连通图 $G = (V, E)$，其中 V 是顶点集合，E 是边集合

输出：最小生成树 MST

初始化 MST 为空

将所有边按权重 $w(e)$ 从小到大排序

初始化并查集 UF，每个顶点初始时为独立的集合

for 每条边 $e = (u, v)$ 按权重递增顺序 **do**

  if $UF.find(u) \neq UF.find(v)$ **then**

    将边 e 加入 MST

    $UF.union(u, v)$

  end if

end for

return MST

Prim **算法**： Prim 算法是一种贪心算法，用于寻找加权连通图的最小生成树。该算法从一个任意顶点开始，逐步增加新的顶点到生成树中，直到所有顶点都被包含。

算法 2 Prim 算法求最小生成树

输入：加权连通图 $G = (V, E)$，其中 V 是顶点集合，E 是边集合
输出：最小生成树 MST
初始化 MST 为空
初始化 $key[V]$ 为 ∞，用于保存顶点到 MST 的最小权重边
初始化 $parent[V]$ 为 -1，用于保存 MST 中每个顶点的父节点
$key[0] \leftarrow 0$，选择任意顶点作为起始点，这里选择顶点 0
创建一个优先队列 Q，将所有顶点及其 key 值加入 Q
while MST 中顶点数小于 V **do**
    从 Q 中取出 key 最小的顶点 u
    if u 已经在 MST 中 **then**
        继续下一次循环
    end if
    将 u 加入 MST
    for u 的每个邻接顶点 v **do**
        if v 不在 MST 中且 $w(u, v) < key[v]$ **then**
            $key[v] \leftarrow w(u, v)$
            $parent[v] \leftarrow u$
            更新 Q 中 v 的 key 值
        end if
    end for
end while
return MST

2.2.2 性能评估

Kruskal **算法**：

- **时间复杂度**：
  - 边的排序：$O(E \log E)$，其中 E 是边的数量。
  - 并查集操作：$O(\alpha(V))$，其中 α 是阿克曼函数的反函数，对于所有实际的图，它非常接近于 1。
  - 总时间复杂度：$O(E \log E + V \log V)$。

- **空间复杂度**：$O(V+E)$，需要存储边、顶点和并查集结构。

Prim **算法**：

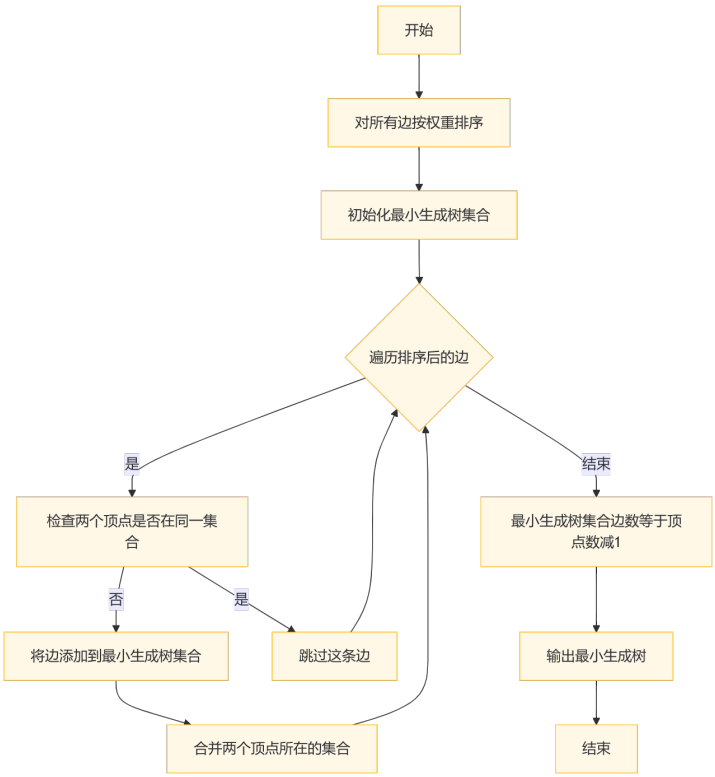
- **时间复杂度**：
  - 使用邻接矩阵：$O(V^2)$，其中 V 是顶点的数量。
  - 使用优先队列（二叉堆）和邻接表：$O((V+E)\log V)$，其中 E 是边的数量。
  - 使用斐波那契堆：$O(E+V\log V)$。
- **空间复杂度**：$O(V)$，需要存储顶点、边和辅助数组。

对于稠密图，Prim 算法（使用邻接矩阵）可能比 Kruskal 算法更高效。对于稀疏图，Kruskal 算法通常更优，因为它涉及的边的数量较少，且并查集操作非常快速。

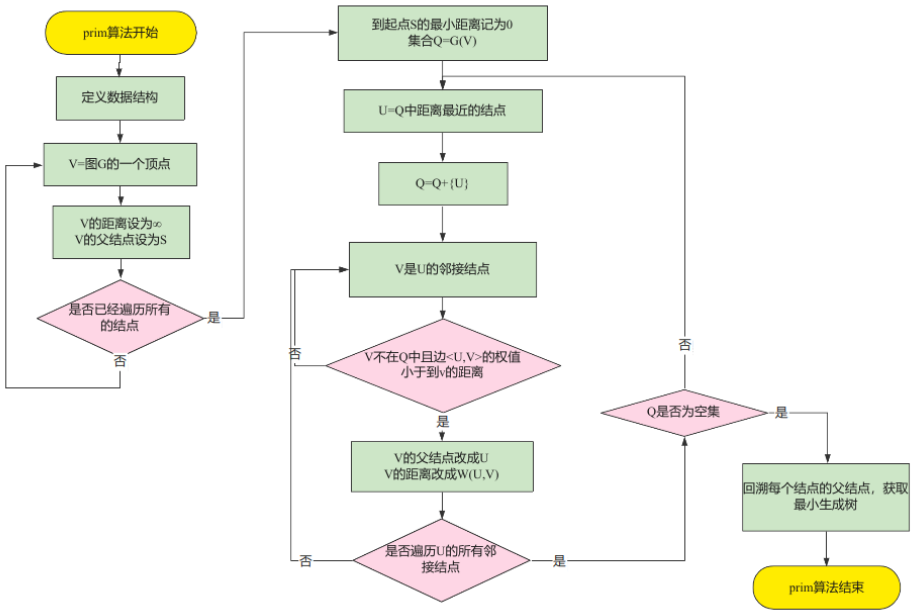
在最坏情况下，两种算法的时间复杂度都不超过 $O(E\log E+V\log V)$。空间复杂度对于两种算法都是线性的，与顶点和边的数量成比例。

2.2.3 流程图表示

Kruskal **算法**：



Prim 算法：



2.2.4 代码实现

Kruskal 算法：

```
void kruskalMST()
{
    // 将边按权值排序
    sort(edges.begin(), edges.end());
    UnionFind uf(V);        // 初始化并查集
    int mst_weight = 0;     // 最小生成树的总权值
    vector<Edge> mst_edges; // 存储最小生成树的边

    for (const auto &edge : edges)
    {// 如果边的两个端点不连通，则加入最小生成树
        if (!uf.connected(edge.u, edge.v))
        {
            uf.unite(edge.u, edge.v); // 合并
            mst_weight += edge.weight;
            mst_edges.push_back(edge);
        }
    }

    cout << "\nKruskal 最小生成树的总权值 (最小成本): " << mst_weight << endl;
    cout << " 构成最小生成树的边: " << endl;
    for (const auto &edge : mst_edges) {
        cout << edge.u + 1 << " - " << edge.v + 1 << " : " << edge.weight << endl;
    }
    cout << endl;
}
```

Prim 算法：

```cpp
void primMST()
{
    vector<int> key(V, numeric_limits<int>::max());  // 保存连接每个顶点的最小权值
    vector<int> parent(V, -1);    // 保存每个顶点的父节点
    vector<bool> inMST(V, false); // 标记顶点是否在最小生成树中

    // 优先队列，按权值从小到大选择边
    priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> pq;

    key[0] = 0;      // 从顶点 0 开始
    pq.push({0, 0}); // {权值，顶点}

    while (!pq.empty())
    {
        int u = pq.top().second; // 选择权值最小的顶点
        pq.pop();

        if (inMST[u])
            continue;    // 如果已在最小生成树中，跳过
        inMST[u] = true; // 标记为已访问

        // 遍历 u 的邻接边
        for (auto &adj : adjList[u])
        {
            int v = adj.first;
            int weight = adj.second;
            // 如果 v 不在最小生成树中且权值小于 key[v]，则更新 key[v] 和 parent[v]
            if (!inMST[v] && weight < key[v])
            {
                key[v] = weight;
                parent[v] = u;
                pq.push({key[v], v});
            }
        }
    }

    int mst_weight = 0; // 最小生成树的总权值
    for (int i = 0; i < V; ++i)
    {
        if (parent[i] != -1) mst_weight += key[i];
    }

    cout << "\nPrim 最小生成树的总权值（最小成本）: " << mst_weight << endl;
    cout << " 构成最小生成树的边: " << endl;
    for (int i = 1; i < V; ++i)
    {
        if (parent[i] != -1) {
            cout << parent[i] + 1 << " - " << i + 1 << " : " << key[i] << endl;
        }
    }
    cout << endl;
}
```

其他代码:

```cpp
#include <iostream>
#include <algorithm>
#include <vector>
#include <queue>
#include <limits>

using namespace std;

// 边
class Edge
{
public:
    int u, v;   // 顶点
    int weight; // 权重
    Edge(int u, int v, int weight) : u(u), v(v), weight(weight) {}
    bool operator<(const Edge &other) const { return weight < other.weight; }
};

// 并查集，用于 Kruskal 算法检测环
class UnionFind
{
private:
    vector<int> parent;
public:
    // 初始化，每个节点的父节点指向自己
    UnionFind(int n)
    {
        parent.resize(n);
        for (int i = 0; i < n; ++i) { parent[i] = i; }
    }
    // 查找，带路径压缩
    int find(int x)
    {
        if (parent[x] != x) { parent[x] = find(parent[x]); }
        return parent[x];
    }
    // 合并
    void unite(int x, int y) { parent[find(x)] = find(y); }
    // 检查两个节点是否联通
    bool connected(int x, int y) { return find(x) == find(y); }
};

// 图
class Graph
{
private:
    int V;                                  // 顶点数
    vector<Edge> edges;                     // 边，用于 Kruskal 算法
    vector<vector<pair<int, int>>> adjList; // 邻接表，用于 Prim 算法
public:
    Graph(int V) : V(V) { adjList.resize(V); }

    // 添加边，u 和 v 为定点编号，weight 为权值
    void addEdge(int u, int v, int weight)
    {
        edges.push_back(Edge(u, v, weight));
```

```
        adjList[u].push_back({v, weight});
        adjList[v].push_back({u, weight});
    }

    void kruskalMST();
    void primMST();
};

int main()
{
    int V, E;

    while (true)
    {
        cout << " 请输入顶点数和边数：(输入 0 0 退出)" << endl;
        cin >> V >> E;
        if (V == 0 && E == 0) return 0;
        else if (cin.fail() || V < 2 || E < V - 1 || E > V * (V - 1) / 2)
        {
            cerr << " 输入错误，请重新输入" << endl;
            cin.clear();
            cin.ignore(numeric_limits<streamsize>::max(), '\n');
            continue;
        }
        else
        {
            break;
        }
    }

    Graph graph(V);
    cout << " 请一次输入每条边的起点、终点和权值（以空格分割）：" << endl;
    for (int i = 0; i < E; ++i)
    {
        cout << " 第 " << i + 1 << " 条边：";
        int u, v, weight;
        cin >> u >> v >> weight;

        if (cin.fail() || u < 1 || u > V || v < 1 || v > V || weight < 0)
        {
            cerr << " 输入错误，请重试!" << endl;
            cin.clear();
            cin.ignore(numeric_limits<streamsize>::max(), '\n');
            --i; // 重复此次输入
            continue;
        }

        --u, --v;
        graph.addEdge(u, v, weight);
    }

    graph.kruskalMST();
    graph.primMST();

    return 0;
}
```

3 项目测试

测试的目的是验证两种算法能否正确地为给定的加权图找到最小生成树，并计算出正确的总权重。同时，测试也旨在检查算法在处理特殊情形，如稀疏图、密集图、只有两个顶点的图以及平凡图时的性能。

3.1 测试用例

3.1.1 一般情况

```
请输入顶点数和边数：（输入 0 0 退出）
7 12
请一次输入每条边的起点、终点和权值（以空格分割）：
第 1 条边：1 2 20
第 2 条边：1 6 23
第 3 条边：1 7 1
第 4 条边：2 7 4
第 5 条边：2 3 15
第 6 条边：6 7 36
第 7 条边：3 7 9
第 8 条边：5 6 28
第 9 条边：5 7 25
第 10 条边：5 4 17
第 11 条边：3 4 3
第 12 条边：4 7 16

Kruskal 最小生成树的总权值(最小成本)：57
构成最小生成树的边：
1 - 7 : 1
3 - 4 : 3
2 - 7 : 4
3 - 7 : 9
5 - 4 : 17
1 - 6 : 23


Prim 最小生成树的总权值(最小成本)：57
构成最小生成树的边：
7 - 2 : 4
7 - 3 : 9
3 - 4 : 3
4 - 5 : 17
1 - 6 : 23
1 - 7 : 1
```

3.1.2 只有两个顶点的图

```
请输入顶点数和边数：(输入 0 0 退出)
2 1
请一次输入每条边的起点、终点和权值（以空格分割）：
第 1 条边：1 2 8

Kruskal 最小生成树的总权值(最小成本)：8
构成最小生成树的边：
1 - 2 : 8


Prim 最小生成树的总权值(最小成本)：8
构成最小生成树的边：
1 - 2 : 8
```

3.1.3 平凡图

```
请输入顶点数和边数：(输入 0 0 退出)
1 0
输入错误，请重新输入
请输入顶点数和边数：(输入 0 0 退出)
```

3.2 健壮性测试

程序具有良好的健壮性，对于输入的不合法数据，程序能够给出合理的提示，如下图所示：

```
请输入顶点数和边数：(输入 0 0 退出)
1 0
输入错误，请重新输入
请输入顶点数和边数：(输入 0 0 退出)
3 9
输入错误，请重新输入
请输入顶点数和边数：(输入 0 0 退出)
8 5
输入错误，请重新输入
请输入顶点数和边数：(输入 0 0 退出)
4 6
请一次输入每条边的起点、终点和权值（以空格分割）：
第 1 条边：1 2 q
输入错误，请重试！
第 1 条边：1 2 3
第 2 条边：fjkn rwk
输入错误，请重试！
第 2 条边：2 4 18
```

4 心得体会

在完成最小生成树项目的实现过程中，我获得了宝贵的学习和实践机会，这些经历不仅加深了我对图论和算法的理解，还提升了我的编程技能和问题解决能力。

通过实现 Prim 算法和 Kruskal 算法，我对最小生成树的概念有了更深刻的认识。我学习了如何通过不同的策略来寻找最小生成树，以及如何评估一个算法的效率和适用性。这些知识不仅局限于理论，还通过实际编码得到了应用和验证；在编写和调试代码的过程中，我提升了自己的编程技能。我学会了如何使用 C++ 标准库中的容器和算法，例如优先队列和向量，以及如何高效地实现并查集数据结构。这些技能对于解决复杂的编程问题至关重要；我学会了如何分析和比较不同算法的时间复杂度和空间复杂度。通过对比 Prim 算法和 Kruskal 算法的性能，我理解了它们各自的优势和局限性，以及如何根据具体问题选择合适的算法；在实现最小生成树的过程中，我深入理解了贪心策略。我学会了如何设计贪心算法来解决优化问题，并理解了贪心选择的性质以及它与全局最优解之间的关系。

总的来说，这个项目不仅加深了我对图论和算法的理解，还提升了我的编程实践能力。这些经验将为我未来的学习和工作奠定坚实的基础。