

## 《离散数学课程项目文档》

### ——最小生成树

作者姓名：\_\_\_\_\_胡峻玮\_\_\_\_\_

学        号：\_\_\_\_\_2153393\_\_\_\_\_

指导教师：\_\_\_\_\_唐剑锋\_\_\_\_\_

学院、专业：\_\_\_\_\_软件学院  软件工程\_\_\_\_\_



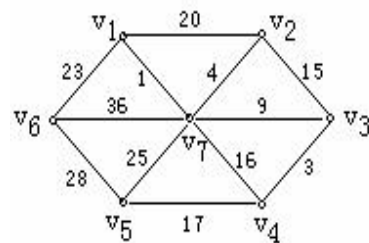
## 目 录

1 项目分析.....	1
1.1 项目要求.....	1
2 项目设计.....	2
2.1 数据结构设计.....	2
2.2 类设计.....	2
2.3 算法设计.....	3
3 主要功能实现.....	4
3.1 程序流程.....	4
3.2 GetVertexAndEdgeCounts 函数实现.....	5
3.3 Graph 类的实现.....	6
3.4 主函数的实现.....	10
4 项目测试.....	11
4.1 项目示例情况.....	11
4.2 仅有两个顶点一条边的情况.....	12
4.3 平凡图的情况.....	12
5 心得体会.....	12

## 1. 项目分析

### 1.1 项目要求

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。



```
请输入所求图的顶点数目和边的数目(以空格分隔各个数,输入两个0结束):7 12
请输入两条边的节点序号以及它们的权值(以空格分隔各个数):
1 2 20
2 3 15
3 4 3
4 5 17
5 6 28
6 1 23
1 7 1
2 7 4
3 7 9
4 7 16
5 7 25
6 7 36
最小耗费是:1和7
最小耗费是:7和2
最小耗费是:7和3
最小耗费是:3和4
最小耗费是:4和5
最小耗费是:1和6
```

## 2. 项目设计

### 2.1 数据结构设计

#### 1. 顶点和边的表示:

- 顶点数组 (vertices): 数组, 每个元素代表一个顶点。顶点用整数索引表示。

- 边数组 (edges): 二维数组, 每行代表一条边, 包含三个整数: 起点索引、终点索引和边的权重。

- 邻接矩阵 (adjMatrix): 二维数组, 存储顶点间的连接关系和边权重。

adjMatrix[i][j] 表示顶点 i 和顶点 j 之间的边权重, 如果没有边则为 INT\_MAX。

#### 2. 图类 (Graph):

- 属性:

- vertexCount: 图中顶点的数量。

- edgeCount: 图中边的数量。

- vertices: 顶点数组。

- edges: 边数组。

- adjMatrix: 邻接矩阵。

- minTotalCost: 最小生成树的总成本。

- 方法:

- 构造函数和析构函数: 负责初始化和清理资源。

- Prim 和 Kruskal: 实现最小生成树算法。

- FindSet 和 UnionSets: 实现并查集操作。

- GetMinTotalCost: 返回最小生成树的总成本。

### 2.2 类设计

#### 1. 构造函数 (Graph):

- 初始化顶点数和边数。
  - 分配并初始化顶点数组、边数组和邻接矩阵。
  - 读取用户输入来填充边数组和邻接矩阵。
2. 析构函数 (~Graph):
- 释放顶点数组、边数组和邻接矩阵的内存。
3. Prim 算法 (Prim):
- 基于邻接矩阵实现。
  - 使用贪心策略逐步构建最小生成树。
4. Kruskal 算法 (Kruskal):
- 基于边数组和并查集实现。
  - 将边按权重排序并逐一尝试加入生成树。
5. 并查集操作 (FindSet, UnionSets):
- FindSet: 递归查找顶点的根节点。
  - UnionSets: 合并两个顶点所在的集合。

## 2.3 算法设计

1. Prim 算法 (Prim) 实现细节:
- 从任意顶点开始构建生成树。
  - 使用一个数组 `visited` 来跟踪已经添加到生成树的顶点。
  - 在每一步中, 查找连接已添加和未添加顶点的最小权重边, 并将其添加到生成树中。
  - 更新总成本和 `visited` 数组。
  - 重复此过程直到所有顶点都被添加。
2. Kruskal 算法 (Kruskal) 实现细节:
- 使用并查集来维护顶点的连接信息。
  - 将所有边按权重排序。
  - 依次遍历排序后的边数组, 对于每条边, 使用 `FindSet` 检查其两个顶点是否已经在同一集合中。
  - 如果不在同一集合中, 使用 `UnionSets` 将这两个顶点合并到同一集合, 并将边的权重添加到总成本中。

- 继续处理直到所有边都被检查。

### 3. 并查集操作:

- FindSet: 递归地查找给定顶点的根顶点, 并在递归过程中实施路径压缩。
- UnionSets: 将一个顶点集合的根顶点指向另一个集合的根顶点, 从而合并两个集合。

## 3. 主要功能实现

### 3.1 程序总流程

#### 1. 程序启动和图的初始化

- 当程序启动时, 首先执行 `main` 函数。
- 在 `main` 函数中, 创建 `Graph` 类的实例 `g`。这触发了 `Graph` 的构造函数。

#### 2. `Graph` 构造函数

- 输入顶点和边数: 首先, 程序要求用户输入顶点数和边数。这是通过调用 `GetVertexAndEdgeCounts` 函数完成的。

- 初始化顶点和边: 接下来, 为顶点数组、边数组和邻接矩阵分配内存。顶点数组简单地存储每个顶点的索引; 边数组用于存储每条边的起始顶点、结束顶点和权重; 邻接矩阵则初始化为最大整数值 `INT_MAX`, 表示顶点间无连接。

- 读取边的信息: 之后, 程序进一步要求用户为每条边输入起始和结束顶点的索引以及权重, 并更新边数组和邻接矩阵。

#### 3. 执行最小生成树算法

在图初始化完成后, 程序继续在 `main` 函数中执行最小生成树算法。

##### 3.1 Prim算法

- 调用 `Prim` 方法来计算最小生成树。
- `Prim` 方法通过贪心算法逐步构建最小生成树, 每次选择连接已选顶点和未

选顶点且权重最小的边。

- 方法维护已访问顶点的集合，并更新最小总成本。
- 完成后，打印出最小生成树的总成本。

### 3.2 Kruskal算法

- 接着，程序调用 `Kruskal` 方法。
- `Kruskal` 方法首先将所有边按权重进行排序。
- 然后，遍历每条边，检查它的两个顶点是否属于不同的集合（使用并查集）。如果是，将这两个顶点合并，并将边的权重加入总成本中。
- 最后，打印出最小生成树的总成本。

## 4. 资源清理

- 在 `main` 函数的末尾，当 `Graph` 类的对象 `g` 离开作用域时，会自动调用 `Graph` 的析构函数。
- 析构函数释放了为顶点数组、边数组和邻接矩阵分配的内存，从而避免了内存泄漏。

这就是整个程序的流程。它首先初始化图，然后应用 `Prim` 和 `Kruskal` 算法来寻找最小生成树，并在最后清理所有动态分配的资源。

## 3.2 GetVertexAndEdgeCounts 函数实现

```
void GetVertexAndEdgeCounts(int& vertexCount, int& edgeCount)
{
    // 循环直到接收到有效的顶点数和边数
    while (true)
    {
        cin >> vertexCount >> edgeCount;
        cin.ignore(INT_MAX, '\n'); // 忽略行末的换行符
        if (vertexCount == 0 && edgeCount == 0) // 检查是否为退出条件
        {
            exit(1);
        }
        else if (!cin || vertexCount < 2 || edgeCount < vertexCount - 1 || edgeCount >
vertexCount * (vertexCount - 1) / 2)
        {
            cout << "输入有误，请重试!" << endl; // 检查输入有效性
        }
    }
}
```

```
        cin.clear();
    }
    else
    {
        break;
    }
}
}
```

## 3.3 Graph 类的实现

```
class Graph
{
private:
    int vertexCount;    // 顶点数
    int edgeCount;      // 边数
    int* vertices;      // 顶点数组
    int** edges;        // 边数组
    int** adjMatrix;    // 邻接矩阵

    int minTotalCost;   // 最小总成本

public:
    Graph();
    ~Graph();

    void Prim();
    void Kruskal();

    int FindSet(int* parent, int vertex);
    void UnionSets(int* parent, int vertex1, int vertex2);

    int GetMinTotalCost() const { return minTotalCost; }
};

Graph::Graph()
{
    cout << "请输入顶点数和边数: ";
    GetVertexAndEdgeCounts(vertexCount, edgeCount);

    vertices = new int[vertexCount];
    for (int i = 0; i < vertexCount; i++)
    {
        vertices[i] = i;
    }
}
```



```
}

edges = new int* [edgeCount];
for (int i = 0; i < edgeCount; i++)
{
    edges[i] = new int[3]; // 存储起点、终点、边权重
}

adjMatrix = new int* [vertexCount];
for (int i = 0; i < vertexCount; i++)
{
    adjMatrix[i] = new int[vertexCount];
    for (int j = 0; j < vertexCount; j++)
    {
        adjMatrix[i][j] = INT_MAX; // 初始化为最大值
    }
}

cout << "请依次输入每条边的起点、终点和权重（用空格隔开）" << endl;
for (int i = 0; i < edgeCount; i++)
{
    int startVertex, endVertex, weight;
    cout << "请输入第" << i + 1 << "条边的信息：";
    cin >> startVertex >> endVertex >> weight;
    cin.ignore(INT_MAX, '\n');

    if (!cin || startVertex < 1 || startVertex > vertexCount || endVertex < 1 ||
endVertex > vertexCount || weight < 0)
    {
        cout << "输入错误，请重试!" << endl;
        cin.clear();
        i--; // 重复此次输入
        continue;
    }

    edges[i][0] = startVertex - 1; // 调整为从 0 开始的索引
    edges[i][1] = endVertex - 1;
    edges[i][2] = weight;

    // 更新邻接矩阵
    adjMatrix[startVertex - 1][endVertex - 1] = weight;
    adjMatrix[endVertex - 1][startVertex - 1] = weight;
}
```

```
}
```

```
Graph::~Graph()
```

```
{
    delete[] vertices;
    for (int i = 0; i < edgeCount; i++)
    {
        delete[] edges[i];
    }
    delete[] edges;

    for (int i = 0; i < vertexCount; i++)
    {
        delete[] adjMatrix[i];
    }
    delete[] adjMatrix;
}
```

```
void Graph::Prim()
```

```
{
    cout << "执行 Prim 算法求最小生成树:" << endl;
    minTotalCost = 0;

    int* visited = new int[vertexCount](); // 已访问顶点数组
    visited[0] = 1; // 从第一个顶点开始

    for (int i = 0; i < vertexCount - 1; i++)
    {
        int minWeight = INT_MAX;
        int start = 0, end = 0;

        for (int j = 0; j < vertexCount; j++)
        {
            if (visited[j])
            {
                for (int k = 0; k < vertexCount; k++)
                {
                    if (!visited[k] && adjMatrix[j][k] < minWeight)
                    {
                        minWeight = adjMatrix[j][k];
                        start = j;
                        end = k;
                    }
                }
            }
        }
    }
}
```

```
    }
    }
}

visited[end] = 1;
minTotalCost += minWeight;
cout << "连接顶点 " << start + 1 << " 和 " << end + 1 << " 的边被选中, 权重: " << minWeight << endl;
}

delete[] visited;
}

void Graph::Kruskal()
{
    cout << "执行 Kruskal 算法求最小生成树:" << endl;
    minTotalCost = 0;

    // 初始化并查集
    int* parent = new int[vertexCount];
    for (int i = 0; i < vertexCount; i++)
    {
        parent[i] = i; // 每个顶点初始时代表自己的集合
    }

    // 对边按权重进行排序
    sort(edges, edges + edgeCount, [](const int* a, const int* b) { return a[2] < b[2]; });

    // 遍历边集合
    for (int i = 0; i < edgeCount; i++)
    {
        int startVertex = edges[i][0];
        int endVertex = edges[i][1];
        int weight = edges[i][2];

        // 检查边的两个顶点是否属于不同集合
        if (FindSet(parent, startVertex) != FindSet(parent, endVertex))
        {
            UnionSets(parent, startVertex, endVertex); // 合并集合
            minTotalCost += weight;
            cout << "连接顶点 " << startVertex + 1 << " 和 " << endVertex + 1 << " 的边被选中, 权重: " << weight << endl;
        }
    }
}
```

```
    }  
}  
  
    delete[] parent;  
}  
  
int Graph::FindSet(int* parent, int vertex)  
{  
    if (vertex != parent[vertex])  
    {  
        parent[vertex] = FindSet(parent, parent[vertex]); // 路径压缩  
    }  
    return parent[vertex];  
}  
  
void Graph::UnionSets(int* parent, int vertex1, int vertex2)  
{  
    int root1 = FindSet(parent, vertex1);  
    int root2 = FindSet(parent, vertex2);  
    if (root1 != root2)  
    {  
        parent[root2] = root1; // 将一个集合的根合并到另一个集合的根  
    }  
}
```

## 3.4 主函数的实现

```
int main()  
{  
    Graph g;  
  
    g.Prim();  
    cout << "Prim 算法计算的最小生成树总成本为: " << g.GetMinTotalCost() << endl;  
  
    cout << endl;  
  
    g.Kruskal();  
    cout << "Kruskal 算法计算的最小生成树总成本为: " << g.GetMinTotalCost() << endl;  
  
    return 0;  
}
```

## 4. 项目测试

### 4.1 项目示例情况

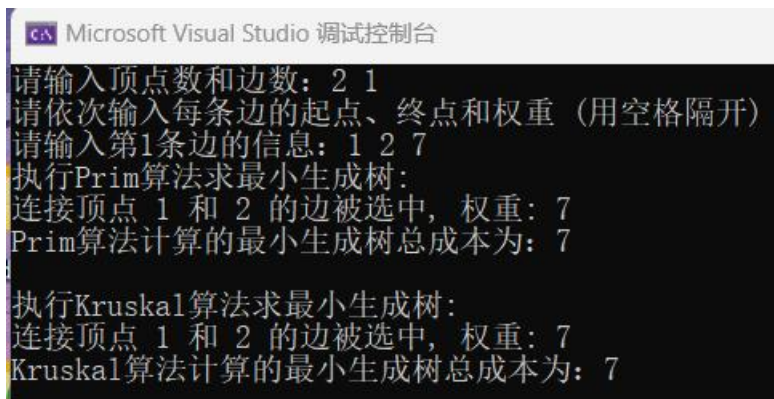
测试结果：

```
Microsoft Visual Studio 调试控制台
请输入顶点数和边数: 7 12
请依次输入每条边的起点、终点和权重（用空格隔开）
请输入第1条边的信息: 1 2 20
请输入第2条边的信息: 1 6 23
请输入第3条边的信息: 1 7 1
请输入第4条边的信息: 2 7 4
请输入第5条边的信息: 2 3 15
请输入第6条边的信息: 6 7 36
请输入第7条边的信息: 3 7 9
请输入第8条边的信息: 5 6 28
请输入第9条边的信息: 5 7 25
请输入第10条边的信息: 5 4 17
请输入第11条边的信息: 3 4 3
请输入第12条边的信息: 4 7 16
执行Prim算法求最小生成树:
连接顶点 1 和 7 的边被选中, 权重: 1
连接顶点 7 和 2 的边被选中, 权重: 4
连接顶点 7 和 3 的边被选中, 权重: 9
连接顶点 3 和 4 的边被选中, 权重: 3
连接顶点 4 和 5 的边被选中, 权重: 17
连接顶点 1 和 6 的边被选中, 权重: 23
Prim算法计算的最小生成树总成本为: 57

执行Kruskal算法求最小生成树:
连接顶点 1 和 7 的边被选中, 权重: 1
连接顶点 3 和 4 的边被选中, 权重: 3
连接顶点 2 和 7 的边被选中, 权重: 4
连接顶点 3 和 7 的边被选中, 权重: 9
连接顶点 5 和 4 的边被选中, 权重: 17
连接顶点 1 和 6 的边被选中, 权重: 23
Kruskal算法计算的最小生成树总成本为: 57
```

## 4.2 仅有两个顶点一条边的情况

测试结果：

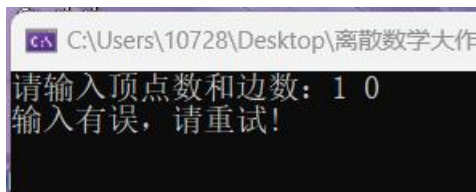


```
Microsoft Visual Studio 调试控制台
请输入顶点数和边数：2 1
请依次输入每条边的起点、终点和权重（用空格隔开）
请输入第1条边的信息：1 2 7
执行Prim算法求最小生成树：
连接顶点 1 和 2 的边被选中，权重：7
Prim算法计算的最小生成树总成本为：7

执行Kruskal算法求最小生成树：
连接顶点 1 和 2 的边被选中，权重：7
Kruskal算法计算的最小生成树总成本为：7
```

## 4.3 平凡图的情况

测试结果：



```
C:\Users\10728\Desktop\离散数学大作
请输入顶点数和边数：1 0
输入有误，请重试！
```

## 5. 心得体会

通过实践，我加深了对图的邻接矩阵表示和边表示的理解。这不仅提升了我对图理论的认识，还增强了我处理复杂数据结构的能力。实现 Prim 和 Kruskal 算法加深了我对最小生成树算法的理解，尤其是在比较两种算法的不同和适用场景时。通过编写并查集，我体会到了它在解决特定问题（如集合合并和环路检测）中的高效性和实用性。这个过程不仅锻炼了编程技巧，特别是在调试和优化代码方面，也提高了我解决实际问题的能力。在处理动态内存时，我意识到资源管理的重要性，学会了如何在程序中正确地分配和释放资源，避免内存泄露。