# 暨南大学本科上机课实验报告

学生姓名：肖健成　　学号：2021103268

学院：网络空间安全　　专业：网络空间安全

课程名称：《算法分析与设计》　　实验名称：用贪心算法求解最小生成树：Prim 算法和 Kruskal 算法

实验类型：验证-设计-综合　　实验时间：2023 年 11 月 22 日晚　　地点：513

---

上机内容：2.5 利用贪心策略设计具有数据输入、处理和输出功能的最小生成树算法，包括两种不同方式：Prim 算法和 Kruskal 算法，实现算法的编译和运行，记录实验过程并整理实验结果。

算法描述：

Prim 算法是一种用于解决最小生成树问题的贪心算法。最小生成树是一棵包含图中所有节点的子树，并且具有最小总权重。Prim 算法的目标是找到这棵最小生成树。下面是对 Prim 算法的详细分析：

输入：

一个无向加权图（通常用邻接矩阵或邻接列表表示）。

一个起始节点（可以是任何图中的节点）。

输出：

最小生成树，通常表示为一组边或节点的集合。

算法步骤：

1.创建一个空的最小生成树（初始时只包含起始节点）。

2.初始化两个数组：

key[]：用于保存每个节点到最小生成树的最小权重边的权重。

mstSet[]：用于标记节点是否已经包含在最小生成树中。

3.将 key 数组中的所有值初始化为正无穷大（表示不可达），将 mstSet 数组中的所有值初始化为 False（表示尚未包含在最小生成树中）。

4.从起始节点开始，设置 key[start_vertex]为 0。

5.循环直到最小生成树包含了图中的所有节点：

选择 key 数组中的最小值所对应的节点 u，该节点不在 mstSet 中。

将节点 u 标记为已包含在最小生成树中，即 mstSet[u] = True。

对于每个与节点 u 相邻的节点 v，如果 v 不在 mstSet 中且与 u 的边的权重小于 key[v]，则更新 key[v]为边 u-v 的权重。

6.最终，最小生成树包含了所有节点，并且包含的边就构成了最小生成树。

Prim 算法使用了贪心策略，每次选择一个节点，它连接到最小生成树中具有最小权重的边。这确保了最终生成的树是最小的。

Kruskal 算法是一种贪心算法，用于解决最小生成树问题。最小生成树是一个包含了图中所有节点的子图，使得该子图中的边权重之和最小。Kruskal 算法的主要目标是找到图中的最小生成树，以下是对 Kruskal 算法的详细分析：

输入：

一个无向加权图（通常用边的列表或邻接列表表示）。

输出：

最小生成树，通常表示为一组边的集合。

算法步骤：

1.创建一个空的最小生成树（初始时没有边）。

2.将图中的所有边按照权重从小到大进行排序。

3.初始化一个空的并查集（通常用来检测是否形成环）。

4.从最小权重的边开始，依次考虑每条边，如果该边不会形成环，就将其添加到最小生成树中。

5.继续考虑下一条边，直到最小生成树中包含了所有的节点为止。

并查集：

并查集是一种数据结构，用于管理元素的集合。它支持两种操作：查找（Find）和合并（Union）。

在 Kruskal 算法中，使用并查集来检测是否形成环。当考虑一条边时，如果边的两个端点属于同一个集合（即它们已经在最小生成树中），那么添加这条边将导致形成环，应该跳过。

如果边的两个端点不属于同一个集合，那么可以将它们合并成一个集合，然后将这条边添加到最小生成树中。

Kruskal 算法使用贪心策略，从权重最小的边开始，逐步选择边，以确保生成的树总权重最小。

Prim 算法伪代码

```
Prim(G, start_vertex):
    num_vertices = G.number_of_vertices()  # 获取图中的节点数
    key = [INFINITY] * num_vertices  # 初始化一个列表，用于保存到最小生成树的最小权重边的权重
    parent = [None] * num_vertices  # 初始化一个列表，用于保存到最小生成树中每个节点的父节点
    key[start_vertex] = 0  # 将起始节点的权重设置为 0
    mst_set = [False] * num_vertices  # 初始化一个列表，用于标记节点是否已经包含在最小生成树中

    for i in range(num_vertices):
        min_key = INFINITY
        min_index = -1
        for v in range(num_vertices):
            if key[v] < min_key and not mst_set[v]:
                min_key = key[v]
                min_index = v
```

```
            mst_set[min_index] = True

            for v in range(num_vertices):
                if G.has_edge(min_index, v) and not mst_set[v] and
G.get_weight(min_index, v) < key[v]:
                    parent[v] = min_index
                    key[v] = G.get_weight(min_index, v)

    result = []  # 用于存储最小生成树的边
    for i in range(num_vertices):
        if i != start_vertex:
            result.append((parent[i], i))

    return result  # 返回最小生成树的边集合
```

Kruskal 算法伪代码

```
Kruskal(G):
    edges = G.get_edges()  # 获取图中的所有边
    edges = sort(edges)  # 按权重升序排序边

    num_vertices = G.number_of_vertices()  # 获取图中的节点数
    minimum_spanning_tree = []  # 用于存储最小生成树的边
    union_find = InitializeUnionFind(num_vertices)  # 初始化并查集，每个节点独立
成集合

    for edge in edges:
        u, v, weight = edge  # 获取边的起点、终点和权重
        if union_find.find(u) != union_find.find(v):
            minimum_spanning_tree.append((u, v, weight))  # 将边添加到最小生
成树
            union_find.union(u, v)  # 合并 u 和 v 所在的集合

    return minimum_spanning_tree  # 返回最小生成树的边集合
```

源程序：

Prim 算法

```python
import sys

# Prim 算法实现
def prim(graph, start_vertex):
    num_vertices = len(graph)
    key = [sys.maxsize] * num_vertices  # 初始化节点到最小生成树的
```

```
最小权重边的权重
    parent = [None] * num_vertices  # 初始化节点的父节点
    key[start_vertex] = 0  # 将起始节点的权重设置为 0
    mst_set = [False] * num_vertices  # 初始化节点是否已包含在最小
生成树中

    for _ in range(num_vertices):
        min_key = sys.maxsize
        min_index = -1
        # 选择最小的 key 值
        for v in range(num_vertices):
            if key[v] < min_key and not mst_set[v]:
                min_key = key[v]
                min_index = v

        mst_set[min_index] = True

        # 更新 key 值和父节点
        for v in range(num_vertices):
            if graph[min_index][v] and not mst_set[v] and
graph[min_index][v] < key[v]:
                parent[v] = min_index
                key[v] = graph[min_index][v]

    # 输出最小生成树的边和权重
    for i in range(num_vertices):
        if i != start_vertex:
            print(f"Edge: {parent[i]} - {i}, Weight:
{graph[i][parent[i]]}")

# 主函数
def main():
    num_vertices = int(input("请输入图的节点数: "))  # 输入节点数
    num_edges = int(input("请输入图的边数: "))  # 输入边数
    graph = [[0] * num_vertices for _ in range(num_vertices)]  # 初
始化图的邻接矩阵

    for _ in range(num_edges):
        start, end, weight = map(int, input("请输入边的起点、终点和
权重，以空格分隔: ").split())
        graph[start][end] = weight
        graph[end][start] = weight  # 由于是无向图，所以两个方向都
要设置权重
```

```
    start_vertex = int(input("请选择起始点: "))  # 输入起始点
    prim(graph, start_vertex)  # 运行 Prim 算法

if __name__ == "__main__":
    main()
```

**Kruskal 算法**

```
# 定义一个图类
class Graph:
    # 初始化函数，接受节点数量作为参数
    def __init__(self, vertices):
        self.V = vertices  # 存储节点数量
        self.graph = []  # 存储图的边

    # 添加一条边的方法，接受起点、终点和权重作为参数
    def add_edge(self, u, v, w):
        self.graph.append([u, v, w])

    # 查找操作，用于寻找某个节点的根节点
    def find(self, parent, i):
        if parent[i] == i:  # 如果该节点是根节点
            return i  # 返回节点本身
        return self.find(parent, parent[i])  # 否则递归查找根节点

    # 合并操作，用于将两个节点所在的集合合并
    def union(self, parent, rank, x, y):
        x_root = self.find(parent, x)  # 找到 x 所在集合的根节点
        y_root = self.find(parent, y)  # 找到 y 所在集合的根节点

        if rank[x_root] < rank[y_root]:  # 如果 x 所在集合的高度小于
y 所在集合
            parent[x_root] = y_root  # 将 x 的根节点设为 y 的根节点
        elif rank[x_root] > rank[y_root]:  # 如果 x 所在集合的高度大
于 y 所在集合
            parent[y_root] = x_root  # 将 y 的根节点设为 x 的根节点
        else:  # 如果两个集合高度相同
            parent[y_root] = x_root  # 将 y 的根节点设为 x 的根节点
            rank[x_root] += 1  # 增加 x 所在集合的高度

    # Kruskal 算法实现
    def kruskal(self):
        result = []  # 用于存储最小生成树的边
        i = 0  # 用于迭代 graph 列表中的边
        e = 0  # 计数最小生成树的边数
        self.graph = sorted(self.graph, key=lambda item: item[2])
```

```
# 按边的权重升序排序

        parent = []  # 存储每个节点的父节点
        rank = []  # 存储每个节点所在集合的高度

        for node in range(self.V):  # 初始化 parent 和 rank
            parent.append(node)  # 每个节点初始时都是自己的根节点
            rank.append(0)  # 初始高度为 0

        while e < self.V - 1:  # 循环直到生成树包含 V-1 条边
            u, v, w = self.graph[i]  # 获取边的起点、终点和权重
            i = i + 1  # 移动到下一条边
            x = self.find(parent, u)  # 查找 u 的根节点
            y = self.find(parent, v)  # 查找 v 的根节点

            if x != y:  # 如果 u 和 v 不在同一集合
                e = e + 1  # 增加生成树的边数
                result.append([u, v, w])  # 将边添加到生成树
                self.union(parent, rank, x, y)  # 合并 u 和 v 所在的
集合

        for u, v, w in result:  # 遍历最小生成树的边
            print(f"Edge: {u} - {v}, Weight: {w}")  # 输出边的起
点、终点和权重

# 主函数
def main():
    num_vertices = int(input("请输入图的节点数: "))  # 输入节点数
    num_edges = int(input("请输入图的边数: "))  # 输入边数
    g = Graph(num_vertices)  # 创建 Graph 对象

    for _ in range(num_edges):
        start, end, weight = map(int, input("请输入边的起点、终点和
权重，以空格分隔: ").split())
        g.add_edge(start, end, weight)  # 添加边到图中

    g.kruskal()  # 运行 Kruskal 算法

if __name__ == "__main__":
    main()
```

**复杂度分析：**

**Prim 算法：**

**1.时间复杂度：**

初始化: 初始化数据结构的时间复杂度为 O(V)，其中 V 是节点的数量。

主循环: Prim 算法的主要工作是在每一步中选择一个节点并查找连接到该节点的最小权重边。对于每个节点，需要查找其未包含在最小生成树中的最小权重边。这通常需要 O(V)时间。由于算法会执行 V 次这样的操作，主循环的时间复杂度为 O(V^2)。

总体时间复杂度：O(V) + O(V^2) = O(V^2)

Prim 算法的总体时间复杂度是 O(V^2)，其中 V 是节点的数量。

2.空间复杂度：

数据结构: 存储最小生成树和其他信息所需的空间取决于数据结构的选择。通常，需要 O(V)的空间来存储 key、parent、mst_set 等数组。

邻接矩阵: 如果使用邻接矩阵来表示图，空间复杂度为 O(V^2)。如果使用邻接列表，则取决于边的数量和节点的数量。

总体空间复杂度：O(V) + O(V^2) = O(V^2)

Prim 算法的总体空间复杂度主要取决于数据结构和图的表示方式,通常为 O(V^2)。

Kruskal 算法：

1.时间复杂度：

排序边: Kruskal 算法首先需要对所有边进行排序，通常使用快速排序或其他排序算法，这需要 O(E log E)的时间，其中 E 是边的数量。

初始化数据结构: 初始化数据结构，如并查集（用于判断边是否形成环），需要 O(V)的时间，其中 V 是节点的数量。

主循环: Kruskal 算法的主要工作是在每一步中选择一个边并检查是否可以将其添加到最小生成树中。对于每条边，需要执行两次查找操作和一次合并操作，这通常需要 O(log V)的时间。由于算法会执行 V-1 次这样的操作，主循环的时间复杂度为 O((V-1) * log V)。

总体时间复杂度：O(E log E) + O(V) + O((V-1) * log V) = O(E log E)

Kruskal 算法的总体时间复杂度是 O(E log E)，其中 E 是边的数量。

2.空间复杂度：

排序边: 对所有边进行排序通常需要 O(E)的额外空间。

存储父亲节点数组：O(V)

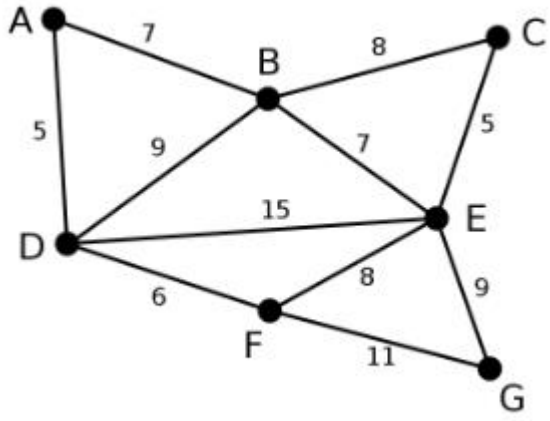
总体空间复杂度： O(E+V)

Kruskal 算法的总体空间复杂度主要取决于排序边的空间需求，通常为 O(E)。

| Prim 算法时间复杂度 | Prim 算法空间复杂度 | Kruskal算法时间复杂度 | Kruskal算法空间复杂度 |
|---|---|---|---|
| O(V^2) | O(V^2) | O(E log E) | O(E+V) |

程序的运行结果及截图：

这是我们的测试用例




下面是使用 prim 算法的实验结果

```
"C:\Program Files\Python39\python.exe" D:
请输入图的节点数: 7
请输入图的边数: 11
请输入边的起点、终点和权重，以空格分隔: 0 1 7
请输入边的起点、终点和权重，以空格分隔: 0 3 5
请输入边的起点、终点和权重，以空格分隔: 1 3 9
请输入边的起点、终点和权重，以空格分隔: 1 2 8
请输入边的起点、终点和权重，以空格分隔: 1 4 7
请输入边的起点、终点和权重，以空格分隔: 2 4 5
请输入边的起点、终点和权重，以空格分隔: 3 4 15
请输入边的起点、终点和权重，以空格分隔: 3 5 6
请输入边的起点、终点和权重，以空格分隔: 4 5 8
请输入边的起点、终点和权重，以空格分隔: 4 6 9
请输入边的起点、终点和权重，以空格分隔: 5 6 11
请选择起始点: 0
Edge: 0 - 1, Weight: 7
Edge: 4 - 2, Weight: 5
Edge: 0 - 3, Weight: 5
Edge: 1 - 4, Weight: 7
Edge: 3 - 5, Weight: 6
Edge: 4 - 6, Weight: 9

进程已结束,退出代码0
```

下面是使用 Kruskal 算法测试的结果

```
"C:\Program Files\Python39\python.exe" E:\pythonProject1\tree2.py
请输入图的节点数: 7
请输入图的边数: 11
请输入边的起点、终点和权重，以空格分隔: 0 1 7
请输入边的起点、终点和权重，以空格分隔: 0 3 5
请输入边的起点、终点和权重，以空格分隔: 1 3 9
请输入边的起点、终点和权重，以空格分隔: 1 2 8
请输入边的起点、终点和权重，以空格分隔: 1 4 7
请输入边的起点、终点和权重，以空格分隔: 2 4 5
请输入边的起点、终点和权重，以空格分隔: 3 4 15
请输入边的起点、终点和权重，以空格分隔: 3 5 6
请输入边的起点、终点和权重，以空格分隔: 4 5 8
请输入边的起点、终点和权重，以空格分隔: 4 6 9
请输入边的起点、终点和权重，以空格分隔: 5 6 11
Edge: 0 - 3, Weight: 5
Edge: 2 - 4, Weight: 5
Edge: 3 - 5, Weight: 6
Edge: 0 - 1, Weight: 7
Edge: 1 - 4, Weight: 7
Edge: 4 - 6, Weight: 9

进程已结束,退出代码0
```

**个人总结：**

**Prim 算法的核心思想：**

Prim 算法是一种贪心算法，其核心思想是从一个起始节点开始，逐步扩展最小生成树，每次选择与当前最小生成树连接最短的边，并将新的节点添加到最小生成树中。

**Kruskal 算法的核心思想：**

Kruskal 算法也是一种贪心算法，其核心思想是从边的集合中逐步选择边，保证不形成环，直到构建完整的最小生成树。

在本次实验中，我经历了从理论到实践的过程，将 Prim 算法与 Kruskal 算法进行了实践，虽然经历了一些实践上的困难，但是最终成功将代码实现。

**收获：**

提高了自己的代码能力与对贪心算法的体会。