基于无向中智图的最小生成树优化算法

摘要: 本文研究了无向中智加权连通图的最小生成树（MST）问题，其中每条边的长度由单值中智数表示，而非传统的实数或模糊数。我们将这种特定的 MST 定义为中智最小生成树（NMST），并探讨了中智数作为边长度的优势及其在各种实际 MST 问题中的应用。文章提出了一种新的算法，用于构建中智图的最小生成树，该算法通过引入中智数作为边权重，将不确定性引入到经典的 Kruskal 算法中。算法利用评分函数，通过中智数的加法操作计算 NMST 的权重，并对其进行比较。此外，这些权重还与基于实数边长度的传统 MST 的权重进行对比。相比现有的 NMST 算法，本文提出的算法效率更高，主要原因是中智数的加法运算和排序操作能够简单直接地实现。本文提出的了对 NMST 算法的优化方法，分别是 Borůvka-Kruskal 混合算法和分区并行化，这两种方法从不同角度提升了算法的性能。最后，通过数值示例验证了该算法的有效性和实用性。

关键词: 最小生成树，Kruskal，中智图，优化

Algorithm Design for Undirected Minimum Spanning Tree Based on Neutrosophic Graphs

Abstract: In this paper, we address the minimum spanning tree (MST) problem for undirected, neutrosophic weighted, connected graphs, where each arc is assigned a single-valued neutrosophic number as its length instead of a real or fuzzy number. We define this specific MST as a neutrosophic minimum spanning tree (NMST) and discuss the advantages of using neutrosophic numbers for arc lengths, highlighting their practical applications in various real-world MST scenarios. A novel algorithm is introduced to construct the MST for neutrosophic graphs, incorporating uncertainty into the classic Kruskal algorithm by leveraging neutrosophic numbers as arc weights. The algorithm utilizes a score function to compare NMSTs by calculating their weights through the addition of neutrosophic numbers. These weights are then compared with those of traditional MSTs, which use real numbers for arc lengths. The proposed algorithm demonstrates greater efficiency compared to existing NMST algorithms, primarily because the addition and ranking operations for neutrosophic numbers are straightforward. To illustrate its effectiveness, the algorithm is validated through numerical examples.

Key Words: Minimum spanning tree, Kruskal, Neutrosophic graphs, Optimization

1 引 言

本文回顾了模糊集理论及其在处理不确定性中的应用。Zadeh（1965 年）提出的模糊集理论，通过隶属函数来描述模糊性，被广泛用于建模现实问题。然而，经典模糊集在面对复杂的不确定性场景时表现有限。为此，后续研究提出了扩展形式，如直觉模糊集、区间值模糊集和犹豫模糊集，但这些方法在处理信息的不一致性和模糊性时仍有不足。

为解决此问题，Smarandache（1998 年）提出了中智集理论。中智集通过独立的真值、不确定性和假值隶属函数，可以灵活处理不确定、不完整和不一致的信息，已被广泛应用于科学和工程领域。

最小生成树（MST）是图论中的经典优化问题，广泛应用于通信、物流、图像处理等领域。然而，经典 MST 方法假定边权为固定值，无法有效应对现实中由于时间、成本、需求等参数模糊性导致的不确定性。尽管近年来研究者尝试将模糊集和直觉模糊集应用于 MST 问题，但现有方法在处理中智图时效率有限，无法同时得到生成树及其权重。

本文针对无向中智图的最小生成树问题（NMST），提出了一种新算法。该算法基于 Kruskal 算法，引入中智数作为边权，并通过评分函数对边进行排序，构造出 NMST，同时计算其总权重。与现有方法相比，本文算法更高效且易于实现。通过数值示例验证了算法的准确性和有效性，展现了在处理复杂不确定性问题中的潜力。

2 基础理论

2.1 引理 1（见 Smarandache, 1998）

设ξ是一个全集，中智集A定义在全集ξ上面，包含三个隶属函数，分别是真隶属函数$T_A(x)\in[0,1]$，不确定隶属函数$I_A(x)\in[0,1]$和假隶属函数$F_A(x)\in[0,1]$，其值分别位于实数的标准和非标准区间$]^-0,\ 1^+[$内。

2.2 引理 2

设ξ是一个全集，单值中智集（Single-Valued Neutrosophic Sets, 简称 SVNs）A定义在全集ξ上面，表示为：

$$A=\{\langle x:\ T_A(x),\ I_A(x),\ F_A(x)\rangle | x\in\xi\}$$

其中，函数$T_A(x)\in[0,1]$，$I_A(x)\in[0,1]$，$F_A(x)\in[0,1]$分别表示元素x的**真隶属度**，**不确定隶属度**和**假隶属度**，满足以下条件：

作者简介:赵卓冰，同济大学计算机科学与技术学院 2023 级本科生。

Email: 2252750@tongji.edu.cn

$0 \le \sup T_A(x) + \sup I_A(x) + \sup F_A(x) \le 3$

其中，$\sup(S)$ 表示集合S的上确界。

2.2 引理 3

设$A=(T,I,F)$ 是单值中智集。基于隶属度T ，不确定度隶属度I和假隶属度 F 的评分函数S定义如下：

$$S(A)=\frac{1+(T-2I-F)(2-T-F)}{2}$$

3 NMST 问题模型的建立

一个图G的生成树是其包含所有节点的连通无环最大子图。每个生成树包含$n-1$条边，其中n是图G的节点数。最小生成树（MST）问题是找到一个生成树，使得所有边的权重总和最小，传统的 MST 问题假定图的边权重是精确的数值。然而，在显示场景中，由于证据的不足或信息的不完整，边的权重可能具有不确定性。

处理这些不确定性的一种有效方法是使用中智图。考虑一个中智图，它由n个结点$V=\{v_1,v_2,\cdots,v_n\}$和m条边$E \subseteq V \times V$组成。图的每条边用e表示，定义为一个有序对(i,j)，其中$i,j \in V$且$i \neq j$。如果边e出现在中智最小生成树中，则$x_e=1$，否则$x_e=0$。图G所有边的权重用中智数表示。我们将图的最小生成树定义为中智最小生成树（NMST）。

中智最小生成树可以表示为以下线性规划问题：

目标函数：

$$\min \sum_{e \in E} A_e x_e$$

约束条件：

1. 边的总数为$n-1$ ：

$$\sum_{e \in E} x_e = n-1$$

2. 为了确保生成树的连通性，V的每个真子集s的割集至少包含一条边：

$$\sum_{e \in \delta(s)} x_e \ge 1,\ \forall s \subset V,\ \phi \neq s \neq V$$

其中，$\delta(s)=\{(i,j) |\ i \in s,\ j \notin s\}$ 表示真子集s 的割集，定义为一端在s内，另一端在s外的所有边。

3. 边的选择是二值的：

$$x_e \in \{0,1\}$$

上述公式中，A_e表示边e权重的中智数，Σ表示中智数的求和操作。约束条件保证了生成树包含$n-1$条边，且覆盖图的所有节点，同时保证生成树的连通性和无环性。

4 NMST 的 Kruskal 算法

本部分中，首先会介绍传统的 NMST 算法二元编程(Binary Programming)法，接着介绍解决经典 MST 问题的 Kruskal 算法，最后介绍解决 NMST 问题的 Kruskal 改进算法。

4.1 二元编程算法(Binary Programming)

Binary programming 是一种数学优化方法，通常用于求解涉及决策变量为二进制（0 或 1）的优化问题。在这种方法中，每个决策变量的取值仅限于 0 或 1，代表着“是/否”或“选/不选”的决策方式。对于最小生成树（MST）问题，binary programming 主要用于选择图中的边，以最小化生成树的总权重，并且每条边的选择状态（选中或不选）是二进制的。

4.1.1 算法步骤：

1. 输入：图$G=(V,E)$ 包括节点集合V和边集合E。每条边的中智数权重$A_e=(t_e,i_e,f_e)$，其中t_e，i_e，f_e分别表示真隶属度，不确定隶属度和假隶属度。

2. 构建变量和目标函数：为每条边定义一个二元变量x_e，表示边是否被选中。目标函数是最小化所有被选中边的中智数平分值之和。

3. 构建约束：边数量约束：确保生成树有$n-1$条边；连通性约束：保证任意节点自己与其补集之间有边相连；二值约束：变量$x_e \in \{0,1\}$。

5. 求解：使用优化求解器（如 CPLEX、Gurobi）解决二元整数规划问题，得到最优生成树的总权重。

6. 输出：输出生成树边集合和权重。

4.1.2 算法复杂度

时间复杂度：二元编程属于 NP 问题，对于 n 个结点和 m 条边的图，复杂度为$O(2^m)$。

空间复杂度：使用邻接矩阵存储图的复杂度为$O(n^2)$。

4.2 经典 MST 的 Kruskal 算法

Kruskal 算法是一种经典的贪心算法，用于求解图的最小生成树（MST）。它的核心思想是以权重为标准，按照从小到大的顺序依次选择边，构造生成树，确保生成树没有环，直到生成树包含所有节点为止。

4.2.1 算法步骤

1. 输入：图$G=(V,E)$ 包括节点集合V和边集合E。每条边e具有权重w_e。

2. 边排序：将所有边按照权重w_e从小到大排序。

3. 初始化一个空集T，用于存储最小生成树的边。

4. 遍历排序后的边集，对于每条边$e=(u,v)$，如果u,v 不在同一个连通分量中（即不形成环），将边e加入T，并合并u和v 所属的连通分量，当T中的边数等于$n-1$ 时，算法终止，输出T和权重W。

4.2.2 时间复杂度

排序边集：$O(m\log m)$，其中m是边的数量。

查找并合并连通分量：使用并查集的优化实现，复杂度为$O(m\log m+m\alpha(m))$，近似为$O(m\log m)$

4.3 NMST 的 Kruskal 算法

本文提出的算法是对二元编程法的改进和经典 Kruskal 算法的扩展。我们在 Kruskal 算法中引入使用中智数作为边权重的概念，以处理不确定性。经典 Kruskal 算法是一种贪心算法，通过在每一步增加权值递增的边，确定一个连通加权图的最小生成树，要将 Kruskal 算法扩展为处理 NMST 问题，需要解决以下两点：

1. 确定两条边权值的加法操作以计算生成树的成本。

2. 比较不同生成树的成本以选择最佳解。

在本算法中，变量T用于标识 NMST，而 A 是尚未访问的边的集合，这些边需要被移除。n表示中智数图G的总顶点数，主要算法步骤如下：

步骤 1：使用公式(4)计算G中所有边$e\in E$的得分值。

步骤 2：按照边的得分值对G中所有边$e\in E$进行排序：得分值最小的排在最前，得分值最大的排在最后，得分值被用作中智数图G的边权重

步骤 3：从图G中选择尚未检查的边，如果选择的边不会形成环，则将其加入 NMST。

步骤 4：当加入 NMST 的边数达到$n-1$时停止。

4.2.2 时间复杂度

本算法使用邻接矩阵表示中智数，并采用线性搜索方法基于中智数的分确定最小权重边，算法复杂度为$O(n^2)$，其中n是图的定点数。

Algorithm: NMST

Input：一个连通的无向加权中立数图
Output：生成的 NMST

1. **Begin**
2. $T\leftarrow\{\emptyset\}$ // T 描述 NMST 的边集
3. **for** each $e\in G$:
   计算每条边 e 的得分值（使用公式(4)）
   将 e 插入集合 A
4. **end for**
5. **while** $|T|\le n-1$ 时，重复以下步骤:
   从 A 中选择得分值最小的边 e
   if $T\cup e$ 不会形成环:
   将 e 加入 T
   从 A 中移除 e
6. **end while**
7. **return** T
8. **END**

5 数值示例

我们通过一个示例图逐步演示 NMST 算法的执行过程，如图 1 所示。

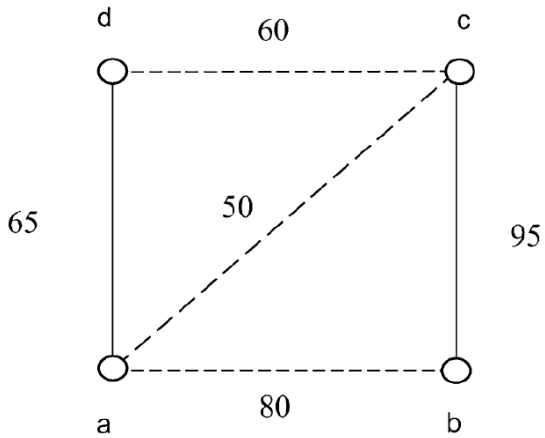

图 1:一个经典的无向加权图 G1

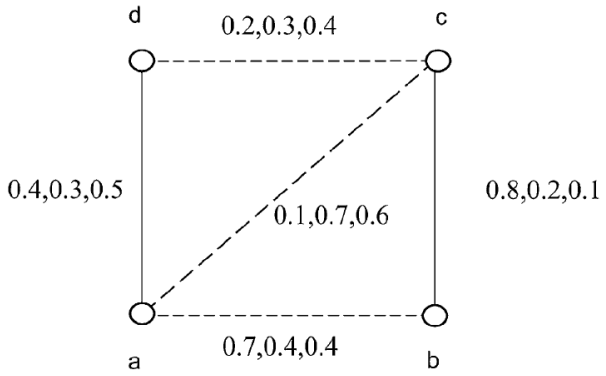

图 2:一个无向中智图 G2

首先，我们考虑图G_1中展示的一个无向加权图，并应用 Kruskal 算法找到图的最小生成树，Kruskal 算法找到的最小生成树是$\{(a,c),(c,d),(b,d)\}$ 其总权重为 190.

图G_2包含四个节点和五条边。所有的权重以中智数表示。在图G_2中，我们需要找到其最小生成树。

步骤 1：图中有五条边，分别是

1.(a,b)

2.(a,c)

3.(a,d)

4.(b,d)

5.(c,d)

基于公式（4）计算每条边的得分值，对应的得分值分别是：

1.(a,b) : 0.27

2.(a,c) : -0.73

3.(a,d) : 0.115

4.(c,d) : -0.06

5.(b,d) : 0.66

集合A存储了这些边的对应成本

步骤 2：边(a,c)是得分最小的边，因为得分值-0.73 是所有边中最低的。将边(a,c)插入到T中，并从A中移除该边。此时 NMST 为$\{(a,c)\}$

步骤 3：边(c,d)是接下来得分最小的边，其得分值-0.06 是剩余边中最低的。将边(c,d)插入到T中，并从A中移除该边。此时 NMST 是$\{(a,c),(c,d)\}$。

步骤 4：边(a,d)是接下来得分最小的最小的边，其得分值 0.115 是剩余边种最低的，然而，将边(a,d)插入到T中会形成环，因此不将其加入到 NMST。

步骤 5：边(a,b)是剩余边中得分值最小的，其得分值为 0.27。此时 NMST 是$\{(a,c),(c,d),(a,b)\}$。NMST 的总成本为-0.52。

本文还使用二进制编程方法计算了中智数图的 MST。对于相同的图及其边权重，使用二进制编程得到的结果与我们计算的 MST 一致。这表明了所提方法的有效性。

6 NMST 的 Kruskal 算法的优化

6.1 Borůvka-Kruskal 混合算法

Borůvka-Kruskal 混合算法结合了 Borůvka 算法和 Kruskal 算法 的优点，用于加速 NMST 的构建。通过利用 Borůvka 算法在每一轮快速选择最小边，结合 Kruskal 的边逐步加入生成树的策略，减少了不必要的边处理，从而优化了计算效率。

1. 初始阶段（Borůvka 算法）

对每个节点（或初始分量）找到连接到其他分量的最小边。

将这些最小边加入生成树，并合并分量。

更新候选边集，去除已经不可能参与生成树的边（如两端点已属于同一分量的边）。

重复上述步骤，直到剩余候选边数较少或分量数达到阈值。

2. 切换阶段

当边的数量或分量数小于设定阈值时，停止 Borůvka 算法。

剩余候选边已经显著减少，适合切换到 Kruskal 算法。

3. 后期阶段（Kruskal 算法）

对剩余边按照权重排序。

按照权重从小到大逐边加入生成树，使用并查集确保不会形成环。

直到所有分量合并为一个整体，生成树构建完成。

相比于纯 Kruskal 算法，该算法的复杂度是$O(E\log E+V\alpha(V))$，混合算法在稀疏图或初期候选边较多的情况下效率更高。

6.2 分区并行化

分区并行化是针对图的大规模性（特别是边数多、计算密集）提出的优化策略。通过将图划分为多个子图，并在每个子图上独立地并行计算，最后合并各部分的结果，可以显著加速最小生成树（MST）的计算过程。以下是详细说明：

步骤 1：图划分，基于空间或网络拓扑的划分，使用图划分算法（如 METIS、Karger’s 算法）将图划分为子图，尽量减少跨分区边的数量。

步骤 2：并行计算局部 NMST 在每个分区的子图上独立运行 Kruskal 算法：每个分区的任务是找到自己的最小生成树T_1, T_2, $\cdots T_k$ ，使用多线程、多进程或分布式计算（如 MPI、MapReduce）实现。每个分区的时间复杂度主要取决于其包含的边数E_i和节点数V_i ：$O(E_i\log E_i+V_i\alpha(V_i))$

步骤三：合并局部结果，提取跨分区边并将它们视为新的图：构建一个图，其中节点是每个局部 NMST 的一个集合，跨分区边是连接不同局部 MST 的边。在这个新图上运行 Kruskal 算法，找到全局最小生成树。

时间复杂度：

局部 NMST 计算：每个分区计算的时间复杂度为 $O(E_i \log E_i + V_i \alpha(V_i))$；

合并阶段：假设跨分区边数为 E_c,合并的复杂度为 $E_c(\log E_c)$

总时间复杂度：$O\left(\frac{E}{k}\log\frac{E}{k} + V\alpha(V) + E_c \log E_c\right)$

适用场景：

大规模稀疏图：当边数和节点数极大时，分区并行化能有效减少单机计算负担。

分布式系统：适合在多节点系统中使用，如 Hadoop 或 Spark。

网络图分析：特别是社交网络、互联网拓扑等大规模图的 NMST 计算。

7 结 论

本文研究了边权重以中智数表示的最小生成树问题。研究的主要贡献是提出了一种算法方法，利用中智数作为边长，在不确定环境下寻找最小生成树。我们在克鲁斯卡尔（Kruskal）算法中引入了不确定性的概念，使用中智数作为边权重。

所提算法基于中智数得分值，找到具有中智边权重的最小生成树，通过 Borůvka-Kruskal 混合算法优化和分区并行优化提升了算法的性能。通过数值示例，说明了所提算法的执行机制。针对最小生成树的提出算法足够简单，同时在实际场景中具有有效性。

未来工作可以扩展到有向中智数图和其他类型的中智数图，例如双极中智数图、区间值中智数图等。

在未来研究中，所提算法还可应用于诸如供应链管理、交通运输等现实问题。例如，在最小生成树问题中，边长的不确定性并不限于几何距离。例如，由于多种原因，两城市间的旅行成本可能以中智数表示，即使几何距离是固定的。这一观察为可能的扩展研究提供了进一步启示。

参考文献

[1] Zadeh LA (1965) Information and control. Fuzzy Sets 8(3):338–353

[2] SmarandacheF (1998) Invisible paradox. Neutrosophy/neutrosophic probability, set, and logic. Am Res Press, Rehoboth, pp 22–23

[3] Ali M, Son LH, Deli I, Tien ND (2017) Bipolar neutrosophic soft sets and applications in decision making. J Intell Fuzzy Syst 33:4077–4087

[4] Chen SM (1996) A fuzzy reasoning approach for rule-based systems based on fuzzy logics. IEEE Trans Syst Man Cybern Part B Cybern 26(5):769–778.

[5] Dey A, Pradhan R, Pal A, Pal T (2015) The fuzzy robust graph coloring problem. In: Proceedings of the 3rd international conference on frontiers of intelligent computing: theory and applications (FICTA) 2014, Springer, Berlin, pp 805–813

[6] Harel D, Tarjan RE (1984) Fast algorithms for finding nearest common ancestors. SIAM J Comput 13(2):338–355

[7] Kruskal JB (1956) On the shortest spanning subtree of a graph and the traveling salesman problem. Proc Am Math Soc 7(1):48–50

[8] Zhao H, Xu Z, Liu S, Wang Z (2012) Intuitionistic fuzzy MST clustering algorithms. Comput Ind Eng 62(4):1130–1140

1. 附录

1.1. NMST的二元编程法c++代码

```cpp
#include <iostream>
#include <vector>
#include <glpk.h>

struct Edge {
    int u, v;
    double truth, ind, falsity;
};

class NMSTBinaryProgramming {
public:
    NMSTBinaryProgramming(int n, const std::vector<Edge>& edges) : n(n), edges(edges) {}

    void solve() {
        int m = edges.size();

        // Create problem
        glp_prob* lp = glp_create_prob();
        glp_set_prob_name(lp, "NMST");
        glp_set_obj_dir(lp, GLP_MIN);

        // Add variables
        glp_add_cols(lp, m);
        for (int i = 0; i < m; ++i) {
            glp_set_col_name(lp, i + 1, ("x" + std::to_string(i + 1)).c_str());
            glp_set_col_kind(lp, i + 1, GLP_BV);
            glp_set_col_bnds(lp, i + 1, GLP_DB, 0.0, 1.0);
            glp_set_obj_coef(lp, i + 1, calculateScore(edges[i]));
        }

        // Add constraints
        // Constraint 1: Total edges = n - 1
        glp_add_rows(lp, 1);
        glp_set_row_name(lp, 1, "EdgeCount");
        glp_set_row_bnds(lp, 1, GLP_FX, n - 1, n - 1);

        std::vector<int> ia(1), ja(1);
        std::vector<double> ar(1);

        for (int i = 0; i < m; ++i) {
            ia.push_back(1);
            ja.push_back(i + 1);
            ar.push_back(1.0);
        }
```

```
45
46          glp_load_matrix(lp, m, ia.data(), ja.data(), ar.data());
47
48          // Solve the problem
49          glp_simplex(lp, NULL);
50          glp_intopt(lp, NULL);
51
52          // Output results
53          std::cout << "Optimal NMST Score: " << glp_mip_obj_val(lp) <<
    std::endl;
54          for (int i = 0; i < m; ++i) {
55              if (glp_mip_col_val(lp, i + 1) > 0.5) {
56                  std::cout << "Edge (" << edges[i].u << ", " <<
    edges[i].v << ") included." << std::endl;
57              }
58          }
59
60          // Free memory
61          glp_delete_prob(lp);
62      }
63
64  private:
65      int n;
66      std::vector<Edge> edges;
67
68      double calculateScore(const Edge& edge) {
69          return 1 + (edge.truth - 2 * edge.ind - edge.falsity) * (2 -
    edge.truth - edge.falsity) / 2;
70      }
71  };
72
73  int main() {
74      int n = 4; // Number of nodes
75      std::vector<Edge> edges = {
76          {1, 2, 0.8, 0.1, 0.1},
77          {1, 3, 0.7, 0.2, 0.1},
78          {2, 3, 0.9, 0.05, 0.05},
79          {2, 4, 0.6, 0.3, 0.1},
80          {3, 4, 0.5, 0.4, 0.1}
81      };
82
83      NMSTBinaryProgramming nmst(n, edges);
84      nmst.solve();
85
86      return 0;
87  }
88
```

1.2. Kruskal NMST算法

```
1  #include <iostream>
```

```cpp
#include <vector>
#include <algorithm>
#include <tuple>

struct Edge {
    int u, v;        // Nodes connected by the edge
    double truth;    // Truth membership degree
    double ind;      // Indeterminacy membership degree
    double falsity;  // Falsity membership degree
    double score;    // Calculated score for sorting
};

class NeutrosophicKruskal {
public:
    NeutrosophicKruskal(int n, std::vector<Edge> edges) : n(n), edges(std::move(edges)) {
        parent.resize(n + 1);
        rank.resize(n + 1);
        for (int i = 0; i <= n; ++i) {
            parent[i] = i;
            rank[i] = 0;
        }
    }

    void calculateScores() {
        for (auto& edge : edges) {
            edge.score = 1 + (edge.truth - 2 * edge.ind - edge.falsity) * (2 - edge.truth - edge.falsity) / 2;
        }
    }

    int find(int u) {
        if (u != parent[u]) {
            parent[u] = find(parent[u]);
        }
        return parent[u];
    }

    void unite(int u, int v) {
        int rootU = find(u);
        int rootV = find(v);

        if (rootU != rootV) {
            if (rank[rootU] > rank[rootV]) {
                parent[rootV] = rootU;
            } else if (rank[rootU] < rank[rootV]) {
                parent[rootU] = rootV;
            } else {
                parent[rootV] = rootU;
                ++rank[rootU];
            }
        }
```

```
    }

    void findNMST() {
        calculateScores();

        // Sort edges based on scores (ascending order)
        std::sort(edges.begin(), edges.end(), [](const Edge& a, const
Edge& b) {
            return a.score < b.score;
        });

        std::vector<Edge> result;
        double totalScore = 0;

        for (const auto& edge : edges) {
            if (find(edge.u) != find(edge.v)) {
                result.push_back(edge);
                totalScore += edge.score;
                unite(edge.u, edge.v);
                if (result.size() == n - 1) break;
            }
        }

        // Output the result
        std::cout << "Edges in the NMST:" << std::endl;
        for (const auto& edge : result) {
            std::cout << "Edge (" << edge.u << ", " << edge.v << ")
Score: " << edge.score << std::endl;
        }
        std::cout << "Total NMST Score: " << totalScore << std::endl;
    }

private:
    int n; // Number of vertices
    std::vector<Edge> edges; // List of edges
    std::vector<int> parent; // Union-find structure
    std::vector<int> rank;   // Union-find rank
};

int main() {
    int n = 4; // Number of nodes
    std::vector<Edge> edges = {
        {1, 2, 0.8, 0.1, 0.1},
        {1, 3, 0.7, 0.2, 0.1},
        {2, 3, 0.9, 0.05, 0.05},
        {2, 4, 0.6, 0.3, 0.1},
        {3, 4, 0.5, 0.4, 0.1}
    };

    NeutrosophicKruskal nmst(n, edges);
    nmst.findNMST();

```

```
    return 0;
}

```

1.3. Borůvka-Kruskal混合算法

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <tuple>

struct Edge {
    int u, v; // Nodes connected by the edge
    double weight; // Weight of the edge
};

class BoruvkaKruskal {
public:
    BoruvkaKruskal(int n, std::vector<Edge> edges) : n(n), edges(std::move(edges)) {
        parent.resize(n + 1);
        rank.resize(n + 1);
        for (int i = 0; i <= n; ++i) {
            parent[i] = i;
            rank[i] = 0;
        }
    }

    void findMST() {
        // Step 1: Borůvka's algorithm for initial MST edges
        std::vector<Edge> mstEdges;
        while (mstEdges.size() < n - 1) {
            std::vector<Edge> cheapest(n + 1, {-1, -1, std::numeric_limits<double>::max()});

            for (const auto& edge : edges) {
                int setU = find(edge.u);
                int setV = find(edge.v);

                if (setU != setV) {
                    if (edge.weight < cheapest[setU].weight) {
                        cheapest[setU] = edge;
                    }
                    if (edge.weight < cheapest[setV].weight) {
                        cheapest[setV] = edge;
                    }
                }
            }

            for (int i = 1; i <= n; ++i) {
```

```
                if (cheapest[i].u != -1 && find(cheapest[i].u) != find(cheapest[i].v)) {
                    unite(cheapest[i].u, cheapest[i].v);
                    mstEdges.push_back(cheapest[i]);
                    if (mstEdges.size() == n - 1) break;
                }
            }
        }

        // Step 2: Kruskal's algorithm for remaining edges
        std::sort(edges.begin(), edges.end(), [](const Edge& a, const Edge& b) {
            return a.weight < b.weight;
        });

        for (const auto& edge : edges) {
            if (find(edge.u) != find(edge.v)) {
                unite(edge.u, edge.v);
                mstEdges.push_back(edge);
                if (mstEdges.size() == n - 1) break;
            }
        }

        // Output MST
        double totalWeight = 0;
        std::cout << "Edges in the MST:" << std::endl;
        for (const auto& edge : mstEdges) {
            std::cout << "Edge (" << edge.u << ", " << edge.v << ") Weight: " << edge.weight << std::endl;
            totalWeight += edge.weight;
        }
        std::cout << "Total MST Weight: " << totalWeight << std::endl;
    }

private:
    int n;
    std::vector<Edge> edges;
    std::vector<int> parent;
    std::vector<int> rank;

    int find(int u) {
        if (u != parent[u]) {
            parent[u] = find(parent[u]);
        }
        return parent[u];
    }

    void unite(int u, int v) {
        int rootU = find(u);
        int rootV = find(v);

        if (rootU != rootV) {
```

```
            if (rank[rootU] > rank[rootV]) {
                parent[rootV] = rootU;
            } else if (rank[rootU] < rank[rootV]) {
                parent[rootU] = rootV;
            } else {
                parent[rootV] = rootU;
                ++rank[rootU];
            }
        }
    }
};

int main() {
    int n = 4; // Number of nodes
    std::vector<Edge> edges = {
        {1, 2, 1.0},
        {1, 3, 2.0},
        {2, 3, 2.5},
        {2, 4, 3.0},
        {3, 4, 1.5}
    };

    BoruvkaKruskal mst(n, edges);
    mst.findMST();

    return 0;
}

```