

算法设计与分析

实验报告

Algorithm Design and Analysis

张恒硕

2212266

人工智能学院 智能科学与技术



南开大学
Nankai University



目录

- 1. 实验目的与实验内容3
 - 1.1 实验目的3
 - 1.2 实验内容3
- 2. 实验原理3
 - 2.1 最小生成树3
 - 2.2 Kruskal算法4
 - 2.3 Prim算法4
 - 2.4 反向删除算法5
 - 2.5 Boruvka算法6
 - 2.6 对比6
- 3. 代码解析7
 - 3.1 基础7
 - 3.2 Kruskal算法8
 - 3.3 Prim算法9
 - 3.4 反向删除算法9
 - 3.5 Boruvka算法11
 - 3.6 可视化12
- 4. 实验结果展示与分析13
- 5. 实验总结14
- 附录14

1. 实验目的与实验内容

1.1 实验目的

- 理解最小生成树的概念。
- 学习Kruskal算法和Prim算法的原理，并拓展反向删除算法、Boruvka算法的原理。
- 实现以上四种算法，比较它们的异同。
- 对不同原始图尝试算法，进行更深入的比较。

1.2 实验内容

- 使用Visual Studio 2022编写c++代码。
- 掌握Kruskal算法、Prim算法、反向删除算法、Boruvka算法生成最小生成树的原理。
- 实现四种算法的代码编写，为以下图1给出的题目原始图生成最小生成树，分析比较性能指标与结果。

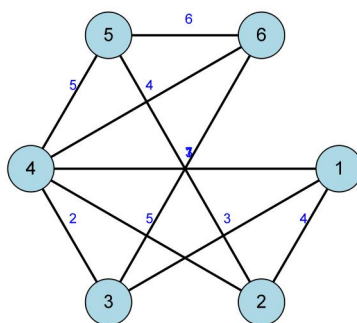


图1 题目原始图

- 进行可视化展示。
- 尝试不同的原始图。

2. 实验原理

2.1 最小生成树

生成树（Spanning Tree）是图论中的重要概念。给定一个无向图（undirected graph） $G = (V, E)$ ，其生成树是一个无环且连通的子图，包含所有顶点（nodes），但只有足够数量（顶点数-1）的部分边（edges）。切分指将顶点任意分为两个不相交的集合。

最小生成树（Minimum Spanning Tree, MST）是连通加权无向图中权值总和最小的生成树。其满足以下性质：

- 唯一性：如果图中所有边权都不相同，则唯一。
- 环路特性：不会包含图中任意环中权值最大的边。
- 切分特性：包含横跨切分的最小权值边。

生成最小生成树一般依赖贪心策略。

2.2 Kruskal算法

Kruskal（克鲁斯卡尔）算法，时间复杂度为 $O(E\log E)$ ，空间复杂度为 $O(V + E)$ ，适合边稀疏的图。

基本思想：初始为只有 n 个顶点而无边的非连通图 $T = (V, \{\})$ ，每个顶点自成一个连通分量（树）。在 E 中选择边权最小的边，若该边依附的顶点分属 T 中不同连通分量，则将此边加入 T 中；否则，舍去此边而选择下一条代价最小的边。依此类推，直至 T 中所有顶点构成一个连通分量为止。如下图2。

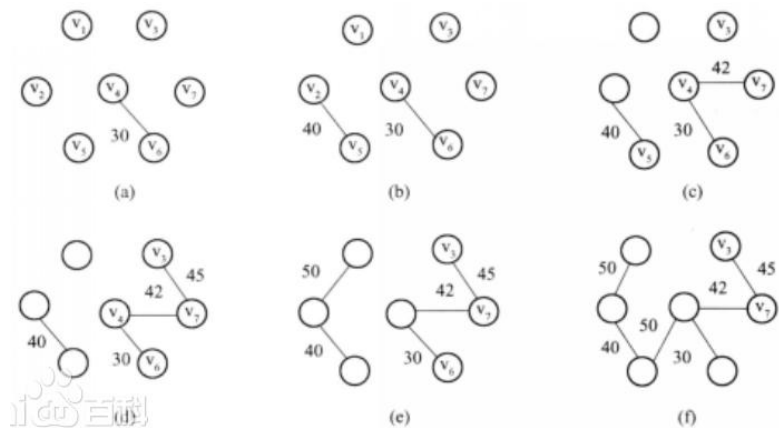


图2 Kruskal算法

输入：带权无向图 $G = (V, E)$
输出：最小生成树的边集合MST
过程 KRUSKAL_MST(G):
 1. 将MST初始化为空集合，将所有顶点初始化为独立集合
 2. 将所有边按权重从小到大排序
 3. 按次序对每条边 (u, v) :
 a. 如果顶点 u 和 v 不在同一集合中（即添加此边不会形成环）：
 i. 将边 (u, v) 加入到MST中
 ii. 合并包含 u 、 v 的两个集合
 4. 返回MST

正确性证明：基于切分定理。假设 A 是目前已确定的部分最小生成树的边集，考虑切分 $(S, V - S)$ ，使 A 中没有横跨该切分的边。假设边 e 是横跨此切分的最小权值边，则存在一个包含 $A \cup \{e\}$ 的最小生成树，确保了加入的是某个切分的安全边。

并查集（Disjoint-Set）：判断加入一条边是否会形成环。其采用路径压缩（执行Find操作时，将路径上所有节点直接连接到根节点，减少未来查询的深度）和按秩合并（执行Union操作时，将较小树连接到较大树上，避免树过深）来提高效率。

2.3 Prim算法

Prim（普里姆）算法，在使用优先队列实现时，时间复杂度为 $O(E \log V)$ ，空间复杂度为 $O(V + E)$ ，适合边稠密的图。

基本思想：从任意顶点开始，逐步构建最小生成树。每步都选择一条连接最小生成树内外顶点的权重最小的边。重复此过程，直到所有顶点都加入最小生成树中。如下图3。

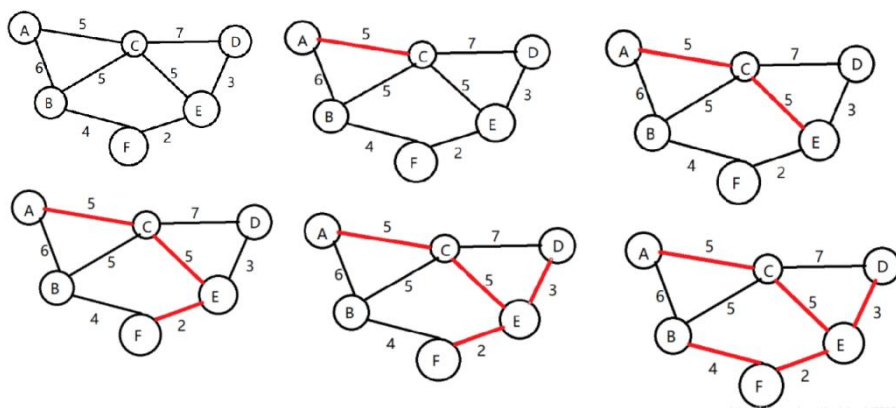


图3 prim算法

输入：带权无向图 $G = (V, E)$

输出：最小生成树的边集合MST

过程 PRIM_MST(G):

1. 选择任意顶点 s 作为MST的起点，将MST初始化为空集合
2. 初始化优先队列 key ，所有顶点的 key 值设为无穷大， s 的 key 值设为0
3. 初始化 $parent$ 数组，记录每个顶点在MST中的父节点， s 的 $parent$ 为 -1
4. 当优先队列不为空时，重复：
 - a. 从优先队列中取出 key 值最小的顶点 u ，标记为已访问
 - b. 如果 u 不是 s ，则将边 $(parent[u], u)$ 加入到MST中
 - c. 遍历 u 的所有邻接顶点 v ：
 - i. 如果 v 未访问且边 (u, v) 的权重小于 v 当前的 key 值：
 - 更新 v 的 $parent$ 为 u ， key 值为边 (u, v) 的权重
 - 将 v 的新 key 值加入优先队列

5. 返回MST

正确性证明：基于切分定理。在算法的每一步，维护已加入MST的顶点集合 A 和未加入MST的顶点集合 B 两个集合，选择横跨切分 (A, B) 的最小权重边 (u, v) ，其中 $u \in A$ 且 $v \in B$ 。根据切分定理，存在一个包含所选边的最小生成树，保证每次选择的边都是安全的。

key数组：维护了每个顶点加入最小生成树的最小成本，可采用线性查找 ($O(V^2)$)、二叉堆优先队列 ($O(E \log V)$) 或斐波那契堆 ($O(E + V \log V)$)，它们的时间复杂度依次下降。

2.4 反向删除算法

反向删除 (Reverse Delete) 算法，时间复杂度为 $O(E \log E + E^2)$ ，空间复杂度为 $O(V + E)$ ，适合于需要考虑整体图结构的场景。

基本思想：与Kruskal算法相反，从包含所有边的完整图开始，按边权重从大到小依次考虑，删除不会影响连通性的边。最终剩下的边构成最小生成树。

输入：带权无向图 $G = (V, E)$

输出：最小生成树的边集合MST

过程 REVERSE_DELETE_MST(G):

1. 将所有边加入集合 S (初始为完整图)，并按权重从大到小排序
2. 按次序对每条边：
 - a. 移除不影响连通性的边
3. 返回剩余边集合 S 作为MST

正确性证明：基于切分定理。假设 e 是当前最大权重边，如果删除 e 后图仍然连通，说明存在其他路径连接 e 的两个端点，这些路径中的每条边权重都不大于 e 。根据切分定理， e 不可能是最小生成树的一部分。如果删除 e 后图变为不连通，则 e 是连接两部分的唯一边，必须是最小生成树的一部分。

连通性检测：结合DFS或BFS与并查集快速进行。

2.5 Boruvka算法

Boruvka（博鲁夫卡）算法是最早的最小生成树算法，由捷克数学家Otakar Boruvka于1926年提出。时间复杂度为 $O(E \log V)$ ，空间复杂度为 $O(V + E)$ ，适合并行计算。

基本思想：各顶点独立初始化，每一步同时考虑所有连通分量，找到一条连接到其他连通分量的最小权重边，然后将这些边添加到最小生成树中，合并相应的连通分量。重复此过程，直到只剩下一个连通分量。

输入：带权无向图 $G = (V, E)$
输出：最小生成树的边集合MST
过程 BORUVKA_MST(G):

1. 将MST初始化为空集合，将每个顶点初始化为一个单独的连通分量
2. 当连通分量数量大于1时，重复：
 - a. 对每个连通分量C找到一条最小权重的边e，该边连接C与另一个连通分量
 - b. 将所有找到的最小权重边加入MST
 - c. 合并被这些边连接的连通分量
3. 返回MST

正确性证明：基于切分定理。在算法的每一步中，对于每个连通分量C，选择横跨切分 $(C, V - C)$ 的最小权重边。根据切分定理，这条边必属于某个最小生成树。由于同时对所有连通分量执行此操作，所有选择的边都是安全的。

2.6 对比

以下表一展示了算法的各方面比较情况。

表一 各算法比较

算法	Kruskal	Prim	反向删除	Boruvka
时间复杂度	$O(E \log E)$	$O(E \log V)$	$O(E \log E + E^2)$	$O(E \log V)$
空间复杂度	$O(V + E)$			
并行性	顺序化			适合并行
迭代特性	添加一条边	添加一个顶点	删除一条边	连通分量数量至少减半
增长模式	多棵树合并	单棵树生长	单棵树收缩	多棵树合并
决策策略	添加不形成环的最小权重边	选择连接MST内外顶点的最小权重边	删除不破坏连通性的最大权重边	为每个连通分量选择一条最小权重外部边
切分应用	隐式	显式	显式	隐式
环路检测	并查集	优先队列	图遍历	并查集
连通性维护	并查集隐式	顶点集合隐式	显式	并查集隐式
构建起始	从空图开始添加边	从单一顶点开始添加边	从完整图开始删除边	从所有顶点开始选择最小权重边
起始点指定	不需要	需要	不需要	
缺点	使用并查集，需要额外空间	使用优先队列，不适合稀疏图	频繁检查连通性，计算开销大	实现复杂，查找最小边开销大
适用场景	稀疏图	稠密图		皆可
增量构建	较难	容易	较难	

3. 代码解析

3.1 基础

```
// 边结构体
struct Edge {
    int u, v, weight; // 起始点、终止点、权重
    // 重载：边权比较
    bool operator<(const Edge& other) const {
        return weight < other.weight;
    }
};

// 边列表->邻接表
vector<vector<pair<int, int>>> edgesToAdjList(const vector<Edge>& edges, int V) {
    vector<vector<pair<int, int>>> adjList(V);
    for (const auto& edge : edges) {
        adjList[edge.u].push_back({ edge.v, edge.weight });
        adjList[edge.v].push_back({ edge.u, edge.weight });
    }
    return adjList;
}

// 并查集
struct DisjointSet {
    vector<int> parent; // 父节点
    vector<int> rank; // 秩
    // 初始化为父节点是自己，秩为0
    DisjointSet(int n) {
        parent.resize(n);
        rank.resize(n, 0);
        for (int i = 0; i < n; ++i) {
            parent[i] = i;
        }
    }
    // 递归查找根节点
    int find(int x) {
        if (parent[x] != x) {
            parent[x] = find(parent[x]);
        }
        return parent[x];
    }
    // 合并
    void unite(int x, int y) {
        int rootX = find(x);
        int rootY = find(y);
        // 同根不需要合并
        if (rootX == rootY)
            return;
        // 将秩小的树连接到秩大的树下
        if (rank[rootX] < rank[rootY]) {
            parent[rootX] = rootY;
        }
    }
};
```

```

    }
    else if (rank[rootX] > rank[rootY]) {
        parent[rootY] = rootX;
    }
    // 两秩相等，任选一个作为父节点，并将其秩加1
    else {
        parent[rootY] = rootX;
        rank[rootX]++;
    }
}
};
// 计算最小生成树的总权重
int calculateTotalWeight(const vector<Edge>& mstEdges) {
    int totalWeight = 0;
    for (const auto& edge : mstEdges) {
        totalWeight += edge.weight;
    }
    return totalWeight;
}

```

- 定义边结构，包含两个顶点和权重信息。
- 重载<运算符，按权重比较边。
- 实现边列表转化为邻接表的函数，后者适用于稀疏矩阵。
- 定义并查集，包含记录父节点的parent数组和记录树高（秩）的rank数组，实现初始化、查找（路径压缩）、合并操作。
- 实现由最小生成树计算总权重的函数。

3.2 Kruskal算法

```

// Kruskal: 按权重从小到大考虑每条边，如果加入该边不会形成环，则将其加入MST
vector<Edge> kruskalMST(const vector<Edge>& edges, int V, double& executionTime) {
    auto startTime = chrono::high_resolution_clock::now();
    vector<Edge> result;
    vector<Edge> sortedEdges = edges;
    sort(sortedEdges.begin(), sortedEdges.end()); // 按边权升序排序
    DisjointSet ds(V); // 检测环路
    // 按权重升序考虑每条边
    for (const auto& edge : sortedEdges) {
        int rootU = ds.find(edge.u);
        int rootV = ds.find(edge.v);
        // 添加边不会形成环（不在同一树下），将边加入MST，合并树
        if (rootU != rootV) {
            result.push_back(edge);
            ds.unite(rootU, rootV);
        }
    }
    auto endTime = chrono::high_resolution_clock::now();
    executionTime = chrono::duration<double, milli>(endTime - startTime).count();
    return result;
}

```

- 按边权升序排序，检测环路，添加不形成环的边。

3.3 Prim算法

```
// Prim: 从起始顶点开始, 每次选择权重最小且连接MST与非MST顶点的边
vector<Edge> primMST(const vector<vector<pair<int, int>>>& adjList, int V, double&
executionTime) {
    auto startTime = chrono::high_resolution_clock::now();
    vector<Edge> result;
    vector<bool> visited(V, false); // 是否已加入MST
    vector<int> key(V, numeric_limits<int>::max()); // 各顶点到MST的最小边权
    vector<int> parent(V, -1); // MST中各顶点的父节点
    priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> pq; //
    优先队列, 选择权重最小的边
    // 起始顶点加入优先队列
    key[0] = 0;
    pq.push({ 0, 0 });
    while (!pq.empty()) {
        int u = pq.top().second; // 获取key值最小的顶点
        pq.pop();
        if (visited[u]) // 跳过已访问
            continue;
        visited[u] = true; // 标记已访问
        // 不是起始节点, 添加边到结果
        if (parent[u] != -1) {
            result.push_back({ parent[u], u, key[u] });
        }
        // 更新key值
        for (const auto& neighbor : adjList[u]) {
            int v = neighbor.first;
            int weight = neighbor.second;
            // 未访问顶点权重变小
            if (!visited[v] && weight < key[v]) {
                parent[v] = u; // 更新父节点
                key[v] = weight; // 更新key值
                pq.push({ key[v], v }); // 加入优先队列
            }
        }
    }
    auto endTime = chrono::high_resolution_clock::now();
    executionTime = chrono::duration<double, milli>(endTime - startTime).count();
    return result;
}
```

- 维护优先队列, 包括key值 (各顶点到MST的最小边权)。
- 从起始顶点开始, 添加key值最小的未访问顶点的边, 更新key值。

3.4 反向删除算法

```
// 深度优先搜索
bool dfs(int u, vector<bool>& visited, const vector<vector<pair<int, int>>>& graph) {
    visited[u] = true;
    for (const auto& neighbor : graph[u]) {
        int v = neighbor.first;
        if (!visited[v]) {
```

```

        dfs(v, visited, graph);
    }
}
return true;
}

// 检查连通性: DFS后检查所有顶点是否被访问
bool isConnected(const vector<vector<pair<int, int>>>& graph, int V) {
    vector<bool> visited(V, false);
    dfs(0, visited, graph);
    for (int i = 0; i < V; ++i) {
        if (!visited[i]) {
            return false;
        }
    }
    return true;
}

// 移除一条边 (双向)
vector<vector<pair<int, int>>> removeEdge(const vector<vector<pair<int, int>>>& graph,
const Edge& edge) {
    vector<vector<pair<int, int>>> newGraph = graph;
    for (auto it = newGraph[edge.u].begin(); it != newGraph[edge.u].end(); ++it) {
        if (it->first == edge.v && it->second == edge.weight) {
            newGraph[edge.u].erase(it);
            break;
        }
    }
    for (auto it = newGraph[edge.v].begin(); it != newGraph[edge.v].end(); ++it) {
        if (it->first == edge.u && it->second == edge.weight) {
            newGraph[edge.v].erase(it);
            break;
        }
    }
    return newGraph;
}

// 反向删除: 按权重从大到小考虑每条边, 如果删除该边不会导致图不连通, 则将其从图中删除
vector<Edge> reverseDeleteMST(const vector<Edge>& edges, int V, double& executionTime) {
    auto startTime = chrono::high_resolution_clock::now();
    // 初始图包含所有边
    vector<vector<pair<int, int>>> graph(V);
    for (const auto& edge : edges) {
        graph[edge.u].push_back({ edge.v, edge.weight });
        graph[edge.v].push_back({ edge.u, edge.weight });
    }
    // 按边权降序排序
    vector<Edge> sortedEdges = edges;
    sort(sortedEdges.begin(), sortedEdges.end(), [](const Edge& a, const Edge& b) {
        return a.weight > b.weight;
    });
    vector<Edge> result = edges;
    // 按边权降序考虑每条边: 尝试移除后检查是否连通
    for (const auto& edge : sortedEdges) {
        auto tempGraph = removeEdge(graph, edge);
        if (isConnected(tempGraph, V)) {

```

```

        result.erase(remove_if(result.begin(), result.end(),
            [&edge](const Edge& e) {
                return (e.u == edge.u && e.v == edge.v && e.weight == edge.weight) ||
                    (e.u == edge.v && e.v == edge.u && e.weight == edge.weight);
            }), result.end());
        graph = tempGraph;
    }
}

auto endTime = chrono::high_resolution_clock::now();
executionTime = chrono::duration<double, milli>(endTime - startTime).count();
return result;
}

```

- DFS遍历后检查连通性。
- 实现双向移除边函数。
- 按边权降序排序，检测连通性，删去不影响连通性的边。

3.5 Boruvka算法

```

// Boruvka: 每一步同时考虑所有树，为每个树寻找最小权重的外部连接边
vector<Edge> boruvkaMST(const vector<vector<pair<int, int>>>& adjList, int V, double&
    executionTime) {
    auto startTime = chrono::high_resolution_clock::now();
    vector<Edge> result;
    DisjointSet ds(V); // 合并树
    int numComponents = V; // 树数
    while (numComponents > 1) {
        // 每个树的最小边（双向只计算一个）
        vector<Edge> cheapest(V, { -1, -1, numeric_limits<int>::max() });
        for (int u = 0; u < V; u++) {
            for (const auto& neighbor : adjList[u]) {
                int v = neighbor.first;
                int weight = neighbor.second;
                if (u < v) {
                    int set1 = ds.find(u);
                    int set2 = ds.find(v);
                    // 如果边连接不同的树，检查并更新二者最小边
                    if (set1 != set2) {
                        if (weight < cheapest[set1].weight) {
                            cheapest[set1] = { u, v, weight };
                        }
                        if (weight < cheapest[set2].weight) {
                            cheapest[set2] = { u, v, weight };
                        }
                    }
                }
            }
        }
        // 添加所有找到的最小边到MST
        for (int i = 0; i < V; i++) {
            Edge edge = cheapest[i];
            if (edge.weight != numeric_limits<int>::max()) {
                int set1 = ds.find(edge.u);
                int set2 = ds.find(edge.v);
                if (set1 != set2) {

```

```

        result.push_back(edge);
        ds.unite(set1, set2);
        numComponents--;
    }
}
}
}

auto endTime = chrono::high_resolution_clock::now();
executionTime = chrono::duration<double, milli>(endTime - startTime).count();
return result;
}

```

- 为每个树寻找最小权重的外部连接边。
- 合并树，迭代进行。

3.6 可视化

```

// SVG图像
void generateSVG(const vector<Edge>& edges, int V, const string& filename) {
    const int nodeRadius = 20; // 节点圆形半径
    const int width = 500;      // 图像宽度
    const int height = 400;     // 图像高度
    const int centerX = width / 2; // 中心点X坐标
    const int centerY = height / 2; // 中心点Y坐标
    const int radius = min(width, height) / 3; // 节点分布半径
    // 创建文件并写入头部声明
    ofstream svgFile(filename + ".svg");
    svgFile << "<?xml version='1.0' encoding='UTF-8' standalone='no'?'>\n";
    svgFile << "<svg width='\"' << width << "\" height='\"' << height << "\"
    xmlns='\"http://www.w3.org/2000/svg\"'>\n";
    // 环形布局节点，极坐标计算
    vector<pair<int, int>> nodePositions(V);
    for (int i = 0; i < V; ++i) {
        double angle = 2.0 * M_PI * i / V;
        nodePositions[i].first = centerX + int(radius * cos(angle));
        nodePositions[i].second = centerY + int(radius * sin(angle));
    }
    // 绘制边
    for (const auto& edge : edges) {
        int x1 = nodePositions[edge.u].first;
        int y1 = nodePositions[edge.u].second;
        int x2 = nodePositions[edge.v].first;
        int y2 = nodePositions[edge.v].second;
        // 黑色边线
        string strokeColor = "black";
        int strokeWidth = 2;
        svgFile << "<line x1='\"' << x1 << "\" y1='\"' << y1
            << "\" x2='\"' << x2 << "\" y2='\"' << y2
            << "\" stroke='\"' << strokeColor << "\" stroke-width='\"' << strokeWidth << "\"
    />\n";
        // 边中点绘制权重
        int textX = (x1 + x2) / 2;
        int textY = (y1 + y2) / 2 - 10;
        svgFile << "<text x='\"' << textX << "\" y='\"' << textY
            << "\" font-family='\"Arial\"' font-size='\"12\"' fill='\"blue\"'>"

```

```
        << edge.weight << "</text>\n";
    }
    // 绘制节点
    for (int i = 0; i < V; ++i) {
        int x = nodePositions[i].first;
        int y = nodePositions[i].second;
        // 圆形节点
        svgFile << "<circle cx=\"" << x << "\" cy=\"" << y
            << "\" r=\"" << nodeRadius << "\" fill=\""lightblue\" stroke=\""black\"
stroke-width="1\" />\n";
        // 节点编号
        svgFile << "<text x=\"" << x << "\" y=\"" << y + 5
            << "\" font-family=\""Arial\" font-size=\""16\" text-anchor=\""middle\"
fill=\""black\">"
            << i + 1 << "</text>\n";
    }
    // 写入尾部并关闭文件
    svgFile << "</svg>\n";
    svgFile.close();
}
```

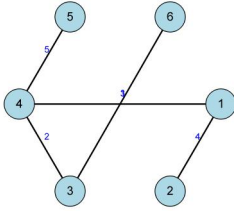
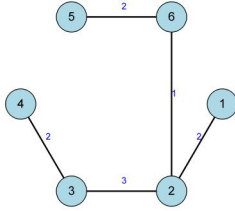
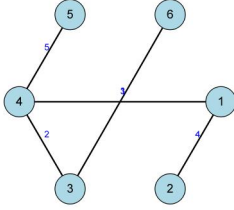
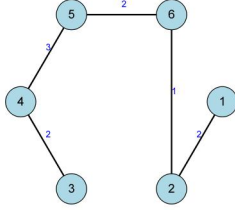
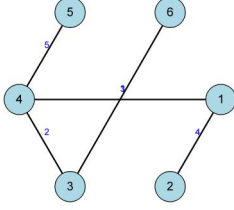
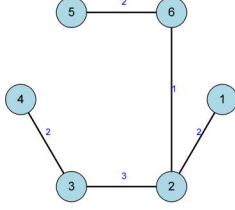
- 输出svg格式的图，节点序号加1。

4. 实验结果展示与分析

以下进行了两组实验，第一组采用题目原始图，第二组采用修改后的原始图（修改边权并添加边），它们的结果如下表二。

表二 实验结果展示

算法	总权 值1	耗时 1(ms)	最小生成树1	总权 值2	耗时 2(ms)	最小生成树2
原始图						
Kruskal	15	0.0208		10	0.0191	

Prim	0.0412		10	0.0369	
反向删除	0.2156		10	0.2409	
Boruvka	0.0174		10	0.0161	
<p>结果分析：第一组题目原始图在不同算法下的最小生成树是一致的，这是因为其绝大部分边的权值都不相等，经理论分析，其最小生成树唯一。给出的新原始图中有不少等值边权，故不同算法的结果有所差异。</p> <p>耗时分析：Kruskal和Boruvka最快，Prim大约是它俩的二倍，而反向删除要大一个数量级。Kruskal适合当前边稀疏的场景。Prim本身适用于稠密图，优先队列的优势没有充分表现。对比两个例子，2比1多出数个边，然而在Kruskal耗时增加时，Prim的耗时却减少了，这也能反应出二者在适应场景上的分野。反向删除要多次进行遍历检查连通性，显然要消耗更多的时间。Boruvka由于并行处理，是最快的。</p> <p>总结：各算法具有不同的适应场景，Kruskal和Prim作为最常用的两种算法，分别适应稀疏图和稠密图场景，而最古老的Boruvka却在并行上显现优势。在具体使用时，需根据具体情况选择。</p>					

5. 实验总结

本次实验实现并比较了四种最小生成树算法，加深了对算法理论的理解，提高了编程能力。实验结果表明，不同算法各有特点，需要根据具体应用场景选择合适的算法。

附录

完整代码：

```
#include <iostream>
```

```

#include <vector>
#include <algorithm>
#include <fstream>
#include <queue>
#include <limits>
#include <cstdlib>
#include <string>
#include <sstream>
#include <cmath>
#define _USE_MATH_DEFINES
#include <math.h>
#include <chrono>
using namespace std;

// 边结构体
struct Edge {
    int u, v, weight; // 起始点、终止点、权重
    // 重载：边权比较
    bool operator<(const Edge& other) const {
        return weight < other.weight;
    }
};

// 边列表->邻接表
vector<vector<pair<int, int>>> edgesToAdjList(const vector<Edge>& edges, int V) {
    vector<vector<pair<int, int>>> adjList(V);
    for (const auto& edge : edges) {
        adjList[edge.u].push_back({ edge.v, edge.weight });
        adjList[edge.v].push_back({ edge.u, edge.weight });
    }
    return adjList;
}

// 并查集
struct DisjointSet {
    vector<int> parent; // 父节点
    vector<int> rank; // 秩
    // 初始化为父节点是自己，秩为0
    DisjointSet(int n) {
        parent.resize(n);
        rank.resize(n, 0);
        for (int i = 0; i < n; ++i) {
            parent[i] = i;
        }
    }
    // 递归查找根节点
    int find(int x) {
        if (parent[x] != x) {
            parent[x] = find(parent[x]);
        }
        return parent[x];
    }
    // 合并
    void unite(int x, int y) {
        int rootX = find(x);
        int rootY = find(y);
        // 同根不需要合并
        if (rootX == rootY)
            return;
        // 将秩小的树连接到秩大的树下
        if (rank[rootX] < rank[rootY]) {
            parent[rootX] = rootY;
        }
        else if (rank[rootX] > rank[rootY]) {
            parent[rootY] = rootX;
        }
        // 两秩相等，任选一个作为父节点，并将其秩加1
        else {
            parent[rootY] = rootX;
            rank[rootX]++;
        }
    }
};

```

```

// Kruskal: 按权重从小到大考虑每条边, 如果加入该边不会形成环, 则将其加入MST
vector<Edge> kruskalMST(const vector<Edge>& edges, int V, double& executionTime) {
    auto startTime = chrono::high_resolution_clock::now();
    vector<Edge> result;
    vector<Edge> sortedEdges = edges;
    sort(sortedEdges.begin(), sortedEdges.end()); // 按边权升序排序
    DisjointSet ds(V); // 检测环路
    // 按权重升序考虑每条边
    for (const auto& edge : sortedEdges) {
        int rootU = ds.find(edge.u);
        int rootV = ds.find(edge.v);
        // 添加边不会形成环 (不在同一树下), 将边加入MST, 合并树
        if (rootU != rootV) {
            result.push_back(edge);
            ds.unite(rootU, rootV);
        }
    }
    auto endTime = chrono::high_resolution_clock::now();
    executionTime = chrono::duration<double, milli>(endTime - startTime).count();
    return result;
}

// Prim: 从起始顶点开始, 每次选择权重最小且连接MST与非MST顶点的边
vector<Edge> primMST(const vector<vector<pair<int, int>>>& adjList, int V, double& executionTime) {
    auto startTime = chrono::high_resolution_clock::now();
    vector<Edge> result;
    vector<bool> visited(V, false); // 是否已加入MST
    vector<int> key(V, numeric_limits<int>::max()); // 各顶点到MST的最小边权
    vector<int> parent(V, -1); // MST中各顶点的父节点
    priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> pq; // 优先队列, 选择权重最小的边
    // 起始顶点加入优先队列
    key[0] = 0;
    pq.push({ 0, 0 });
    while (!pq.empty()) {
        int u = pq.top().second; // 获取key值最小的顶点
        pq.pop();
        if (visited[u]) // 跳过已访问
            continue;
        visited[u] = true; // 标记已访问
        // 不是起始节点, 添加边到结果
        if (parent[u] != -1) {
            result.push_back({ parent[u], u, key[u] });
        }
        // 更新key值
        for (const auto& neighbor : adjList[u]) {
            int v = neighbor.first;
            int weight = neighbor.second;
            // 未访问顶点权重变小
            if (!visited[v] && weight < key[v]) {
                parent[v] = u; // 更新父节点
                key[v] = weight; // 更新key值
                pq.push({ key[v], v }); // 加入优先队列
            }
        }
    }
    auto endTime = chrono::high_resolution_clock::now();
    executionTime = chrono::duration<double, milli>(endTime - startTime).count();
    return result;
}

// 深度优先搜索
bool dfs(int u, vector<bool>& visited, const vector<vector<pair<int, int>>>& graph) {
    visited[u] = true;
    for (const auto& neighbor : graph[u]) {
        int v = neighbor.first;
        if (!visited[v]) {
            dfs(v, visited, graph);
        }
    }
    return true;
}

```



```

// 检查连通性: DFS后检查所有顶点是否被访问
bool isConnected(const vector<vector<pair<int, int>>>& graph, int V) {
    vector<bool> visited(V, false);
    dfs(0, visited, graph);
    for (int i = 0; i < V; ++i) {
        if (!visited[i]) {
            return false;
        }
    }
    return true;
}

// 移除一条边 (双向)
vector<vector<pair<int, int>>> removeEdge(const vector<vector<pair<int, int>>>& graph, const Edge& edge) {
    vector<vector<pair<int, int>>> newGraph = graph;
    for (auto it = newGraph[edge.u].begin(); it != newGraph[edge.u].end(); ++it) {
        if (it->first == edge.v && it->second == edge.weight) {
            newGraph[edge.u].erase(it);
            break;
        }
    }
    for (auto it = newGraph[edge.v].begin(); it != newGraph[edge.v].end(); ++it) {
        if (it->first == edge.u && it->second == edge.weight) {
            newGraph[edge.v].erase(it);
            break;
        }
    }
    return newGraph;
}

// 反向删除: 按权重从大到小考虑每条边, 如果删除该边不会导致图不连通, 则将其从图中删除
vector<Edge> reverseDeleteMST(const vector<Edge>& edges, int V, double& executionTime) {
    auto startTime = chrono::high_resolution_clock::now();
    // 初始图包含所有边
    vector<vector<pair<int, int>>> graph(V);
    for (const auto& edge : edges) {
        graph[edge.u].push_back({ edge.v, edge.weight });
        graph[edge.v].push_back({ edge.u, edge.weight });
    }
    // 按边权降序排序
    vector<Edge> sortedEdges = edges;
    sort(sortedEdges.begin(), sortedEdges.end(), [](const Edge& a, const Edge& b) {
        return a.weight > b.weight;
    });
    vector<Edge> result = edges;
    // 按边权降序考虑每条边: 尝试移除后检查是否连通
    for (const auto& edge : sortedEdges) {
        auto tempGraph = removeEdge(graph, edge);
        if (isConnected(tempGraph, V)) {
            result.erase(remove_if(result.begin(), result.end(),
                [&edge](const Edge& e) {
                    return (e.u == edge.u && e.v == edge.v && e.weight == edge.weight) ||
                        (e.u == edge.v && e.v == edge.u && e.weight == edge.weight);
                }), result.end());
            graph = tempGraph;
        }
    }
    auto endTime = chrono::high_resolution_clock::now();
    executionTime = chrono::duration<double, milli>(endTime - startTime).count();
    return result;
}

// Boruvka: 每一步同时考虑所有树, 为每个树寻找最小权重的外部连接边
vector<Edge> boruvkaMST(const vector<vector<pair<int, int>>>& adjList, int V, double& executionTime) {
    auto startTime = chrono::high_resolution_clock::now();
    vector<Edge> result;
    DisjointSet ds(V); // 合并树
    int numComponents = V; // 树数
    while (numComponents > 1) {
        // 每个树的最小边 (双向只计算一个)
        vector<Edge> cheapest(V, { -1, -1, numeric_limits<int>::max() });
        for (int u = 0; u < V; u++) {
            for (const auto& neighbor : adjList[u]) {
                int v = neighbor.first;

```

```

        int weight = neighbor.second;
        if (u < v) {
            int set1 = ds.find(u);
            int set2 = ds.find(v);
            // 如果边连接不同的树，检查并更新二者最小边
            if (set1 != set2) {
                if (weight < cheapest[set1].weight) {
                    cheapest[set1] = { u, v, weight };
                }
                if (weight < cheapest[set2].weight) {
                    cheapest[set2] = { u, v, weight };
                }
            }
        }
    }
}

// 添加所有找到的最小边到MST
for (int i = 0; i < V; i++) {
    Edge edge = cheapest[i];
    if (edge.weight != numeric_limits<int>::max()) {
        int set1 = ds.find(edge.u);
        int set2 = ds.find(edge.v);
        if (set1 != set2) {
            result.push_back(edge);
            ds.unite(set1, set2);
            numComponents--;
        }
    }
}

}

auto endTime = chrono::high_resolution_clock::now();
executionTime = chrono::duration<double, milli>(endTime - startTime).count();
return result;
}

// SVG图像
void generateSVG(const vector<Edge>& edges, int V, const string& filename) {
    const int nodeRadius = 20; // 节点圆形半径
    const int width = 500; // 图像宽度
    const int height = 400; // 图像高度
    const int centerX = width / 2; // 中心点X坐标
    const int centerY = height / 2; // 中心点Y坐标
    const int radius = min(width, height) / 3; // 节点分布半径
    // 创建文件并写入头部声明
    ofstream svgFile(filename + ".svg");
    svgFile << "<?xml version='1.0' encoding='UTF-8' standalone='no'?'>\n";
    svgFile << "<svg width='\" << width << '\" height='\" << height << '\" xmlns='\"http://www.w3.org/2000/svg\">\n";
    // 环形布局节点，极坐标计算
    vector<pair<int, int>> nodePositions(V);
    for (int i = 0; i < V; ++i) {
        double angle = 2.0 * M_PI * i / V;
        nodePositions[i].first = centerX + int(radius * cos(angle));
        nodePositions[i].second = centerY + int(radius * sin(angle));
    }
    // 绘制边
    for (const auto& edge : edges) {
        int x1 = nodePositions[edge.u].first;
        int y1 = nodePositions[edge.u].second;
        int x2 = nodePositions[edge.v].first;
        int y2 = nodePositions[edge.v].second;
        // 黑色边线
        string strokeColor = "black";
        int strokeWidth = 2;
        svgFile << "<line x1='\" << x1 << '\" y1='\" << y1
            << '\" x2='\" << x2 << '\" y2='\" << y2
            << '\" stroke='\" << strokeColor << '\" stroke-width='\" << strokeWidth << '\" />\n";
        // 边中点绘制权重
        int textX = (x1 + x2) / 2;
        int textY = (y1 + y2) / 2 - 10;
        svgFile << "<text x='\" << textX << '\" y='\" << textY
            << '\" font-family='\"Arial\" font-size='\"12\" fill='\"blue\">\"
            << edge.weight << "</text>\n";
    }
}

```

```

// 绘制节点
for (int i = 0; i < V; ++i) {
    int x = nodePositions[i].first;
    int y = nodePositions[i].second;
    // 圆形节点
    svgFile << "<circle cx=\"" << x << "\" cy=\"" << y
        << "\" r=\"" << nodeRadius << "\" fill=\""lightblue\" stroke=\""black\" stroke-width=\""1\" />\n";
    // 节点编号
    svgFile << "<text x=\"" << x << "\" y=\"" << y + 5
        << "\" font-family=\""Arial\" font-size=\""16\" text-anchor=\""middle\" fill=\""black\">"
        << i + 1 << "</text>\n";
}
// 写入尾部并关闭文件
svgFile << "</svg>\n";
svgFile.close();
}

// 计算最小生成树的总权重
int calculateTotalWeight(const vector<Edge>& mstEdges) {
    int totalWeight = 0;
    for (const auto& edge : mstEdges) {
        totalWeight += edge.weight;
    }
    return totalWeight;
}

int main() {
    int V = 6; // 顶点数
    // 边集合
    vector<Edge> edges = {
        {0, 1, 4}, {0, 2, 3}, {0, 3, 1}, {1, 3, 5}, {1, 4, 7},
        {2, 3, 2}, {2, 5, 3}, {3, 4, 5}, {3, 5, 4}, {4, 5, 6}
    };
    auto adjList = edgesToAdjList(edges, V); // 邻接表
    generateSVG(edges, V, "original_graph"); // 原始图
    // Kruskal
    double kruskalTime;
    vector<Edge> kruskalResult = kruskalMST(edges, V, kruskalTime);
    generateSVG(kruskalResult, V, "kruskal_mst");
    int kruskalTotalWeight = calculateTotalWeight(kruskalResult);
    cout << "Kruskal MST总权重: " << kruskalTotalWeight << endl;
    cout << "Kruskal算法执行时间: " << kruskalTime << " 毫秒" << endl;
    // Prim
    double primTime;
    vector<Edge> primResult = primMST(adjList, V, primTime);
    generateSVG(primResult, V, "prim_mst");
    int primTotalWeight = calculateTotalWeight(primResult);
    cout << "Prim MST总权重: " << primTotalWeight << endl;
    cout << "Prim算法执行时间: " << primTime << " 毫秒" << endl;
    // 反向删除
    double reverseDeleteTime;
    vector<Edge> reverseDeleteResult = reverseDeleteMST(edges, V, reverseDeleteTime);
    generateSVG(reverseDeleteResult, V, "reverse_delete_mst");
    int reverseDeleteTotalWeight = calculateTotalWeight(reverseDeleteResult);
    cout << "反向删除MST总权重: " << reverseDeleteTotalWeight << endl;
    cout << "反向删除算法执行时间: " << reverseDeleteTime << " 毫秒" << endl;
    // Boruvka
    double boruvkaTime;
    vector<Edge> boruvkaResult = boruvkaMST(adjList, V, boruvkaTime);
    generateSVG(boruvkaResult, V, "boruvka_mst");
    int boruvkaTotalWeight = calculateTotalWeight(boruvkaResult);
    cout << "Boruvka MST总权重: " << boruvkaTotalWeight << endl;
    cout << "Boruvka算法执行时间: " << boruvkaTime << " 毫秒" << endl;
    return 0;
}

```