《离散数学》课程实验报告文档

题目：最小生成树

姓名：　　　赵卓冰

学号：　　　2252750

专业：　　　软件工程

年级：　　　2023 级

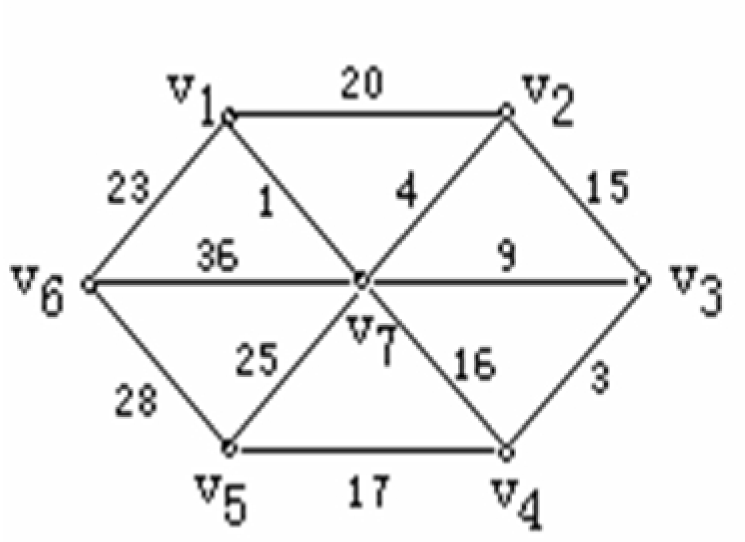
指导教师：唐剑锋

2024 年 11 月 21 日

项目背景及问题描述
思路
最小生成树算法原理
1 **最小生成树的基本概念**
2 **最小生成树的特点**
3 **求解最小生成树的常见算法**
3. 1 **Prim算法**
3. 2 **Prim算法步骤**:
3. 3 **Prim算法的特点**:
3. 4 **Kruskal算法**
3. 5 **Kruskal算法步骤**:
3. 6 **Kruskal算法的特点**:
4 **Prim与Kruskal算法的比较**
数据结构设计
1 最小生成树类实现
1. 1 1. **类的成员变量和方法设计**
2 图的表示（邻接矩阵和邻接表）
2. 1 邻接矩阵实现
2. 2 邻接表实现
3 优先队列（用于 Prim 算法）
4 并查集（用于 Kruskal 算法）
最小生成树算法实现
1 **Prim算法**（邻接矩阵实现）
2 **Prim算法**（邻接表实现）
3 **Kruskal算法**
流程图
1 **程序启动**
2 **用户输入数据**
3 **选择算法**
4 **执行算法**
5 **显示结果**
6 说明:
测试结果
1 Windows平台
2 Linux平台
心得体会

1. 项目背景及问题描述

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。



这个问题实际应用中可以被看作 **最小生成树**（MST）问题，常见的解决方法有以下几种：

1. **Prim算法**（适用于稠密图，基于邻接矩阵或邻接表）
2. **Kruskal算法**（适用于稀疏图，基于边的排序和并查集）

2. 思路

我们可以通过以下几个步骤来解决该问题：

1. **数据结构设计**：首先，我们需要设计合适的数据结构来表示图，保存节点信息、边的信息。
2. **最小生成树算法的实现**：然后，选择合适的最小生成树算法来解决问题，如 Prim 算法或 Kruskal 算法。
3. **用户交互界面设计**：为了让用户可以交互式地操作程序，需要设计一个简单的命令行交互界面，支持用户输入节点信息、边信息，以及选择最小生成树算法。

3. 最小生成树算法原理

最小生成树（MST，Minimum Spanning Tree）是一个图论中的经典问题，目标是从一个连通的无向图中选取若干条边，构建一个树，且这棵树包含所有节点，且边的权重总和最小。

3.1. 最小生成树的基本概念

- **图**：由节点（顶点）和边（连接两个节点的线段）构成的集合。
- **生成树**：是一个包含图中所有节点的树（无环，且连通）。生成树的边数为$n-1$，其中n是图中节点的数量。
- **最小生成树**：在所有生成树中，边的权重和最小的那一棵树称为最小生成树。

3.2. 最小生成树的特点

- **唯一性**：如果图的边权不全相等，则最小生成树是唯一的；如果存在相同边权的多条边，则最小生成树可以有多个解。
- **连通性**：最小生成树包含图中的所有节点，并保证这些节点通过$n-1$ 条边连通。
- **边权和最小化**：最小生成树是所有可能生成树中，边权和最小的树。

3.3. 求解最小生成树的常见算法

3.3.1. Prim算法

Prim算法是一个贪心算法，用来计算最小生成树。它的基本思想是：从图中的一个节点开始，逐步选择与当前生成树相连的、具有最小权重的边，直到所有节点都被包含在生成树中。

3.3.2. Prim算法步骤：

1. **初始化**：选择一个任意的节点作为生成树的起点，将该节点标记为已访问。
2. **选择最小边**：在已访问的节点集合与未访问的节点集合之间，选择权重最小的边。
3. **更新已访问集合**：将选中的边的另一端节点加入生成树。
4. **重复步骤**：直到所有节点都被包含进生成树。

3.3.3. Prim算法的特点：

- 适用于稠密图，因为其时间复杂度是$O(V^2)$ 或$O(E \log V)$。
- 使用优先队列（最小堆）可以优化选择最小边的过程。

3.3.4. Kruskal算法

Kruskal算法也是一个贪心算法，算法的核心思想是：将图中所有的边按照权重从小到大排序，然后逐一检查边是否能加入到当前生成树中（即检查是否形成环）。如果加入该边不形成环，则将其加入生成树中。

3.3.5. Kruskal算法步骤：

1. **排序边**：首先对图中的所有边按权重进行升序排序。
2. **选择边**：从排序后的边列表中选择边，检查该边是否与已选边构成环（可以通过并查集判断）。
3. **加入生成树**：如果边不构成环，则将其加入生成树。
4. **重复步骤**：直到生成树包含$n-1$ 条边为止。

3.3.6. Kruskal算法的特点：

- 适用于稀疏图，因为其时间复杂度主要取决于边数$O(E \log E)$。
- 使用并查集（Union-Find）来判断是否形成环，并高效地进行合并和查找操作。

3.4. Prim与Kruskal算法的比较

特性	Prim算法	Kruskal算法
适用场景	稠密图（边数较多）	稀疏图（边数较少）
算法类型	贪心算法，逐步扩展生成树	贪心算法，边权从小到大选择边
时间复杂度	$O(E\log V)$ 或$O(V^2)$	$O(E\log E)$ 或$O(E\log V)$
数据结构	优先队列（最小堆）	并查集（Union-Find）
选择方式	每次选择生成树外与当前生成树最短的边	每次选择未加入生成树的最小边，并判断是否会形成环

4. 数据结构设计

为了表示电网的结构，我们采用以下几个数据结构：

1. **图的表示**：使用邻接矩阵和邻接表两种方式来表示图。邻接矩阵适合稠密图，邻接表适合稀疏图。
2. **优先队列**：用于 Prim 算法的最小边选择。
3. **并查集**：用于 Kruskal 算法中处理集合的合并与查找。

4.1. 最小生成树类实现

4.1.1. 1. 类的成员变量和方法设计

- **成员变量**：
  - `vertices`：图的节点数。
  - `graph_matrix`：邻接矩阵表示的图。
  - `graph_list`：邻接表表示的图。
  - `mst_edges`：最小生成树的边集合。
  - `mst_cost`：最小生成树的总权重。
- **成员方法**：
  - `AddEdge()`：添加边到图中（邻接矩阵或邻接表）。
  - `PrimWithMatrix()`：使用Prim算法（邻接矩阵）计算最小生成树。
  - `PrimWithList()`：使用Prim算法（邻接表）计算最小生成树。
  - `Kruskal()`：使用Kruskal算法计算最小生成树。

- `Disp()`：输出最小生成树的边和总权重。

4.2. 图的表示（邻接矩阵和邻接表）

在实现中，我们提供了邻接矩阵和邻接表的两种实现方式：

- **邻接矩阵**：适合于边较多的图，每个元素表示从一个节点到另一个节点的边权。
- **邻接表**：适合于边较少的图，使用链表或向量来存储每个节点的邻居。

4.2.1. 邻接矩阵实现

```
Vector<Vector<int>> graph_matrix;  // 用于存储图的邻接矩阵
graph_matrix.Resize(n, Vector<int>(n, 0));  // 初始化 n 个节点的邻接矩阵，初始边权设为 0
```

4.2.2. 邻接表实现

```
Vector<Vector<pair<int, int>>> graph_list;  // 用于存储图的邻接表
graph_list.Resize(n);  // 初始化图的邻接表
```

4.3. 优先队列（用于 Prim 算法）

优先队列（最小堆）用于 Prim 算法的最小边选择。以下是最小堆的实现：

```
template <typename T, typename Compare>
class PriorityQueue {
    // 内部实现省略
    void Push(const T& value) {
        // 插入元素
    }

    void Pop() {
        // 弹出最小元素
    }

    const T& Top() const {
        // 获取最小元素
    }
};
```

4.4. 并查集（用于 Kruskal 算法）

并查集用于处理图的连通性问题，判断两点是否属于同一连通块。以下是并查集的实现：

```
class UnionFind {
private:
    Vector<int> parent; // 父节点数组
    Vector<int> rank; // 每个节点的秩

public:
```

```
    UnionFind(int n) {
        parent.Resize(n);
        rank.Resize(n);
        for (int i = 0; i < n; ++i) {
            parent[i] = i; // 每个节点初始化为自成一棵树
        }
    }

    int Find(int x) {
        if (parent[x] != x) {
            parent[x] = Find(parent[x]);  // 路径压缩
        }
        return parent[x];
    }

    void Union(int x, int y) {
        int root_x = Find(x);
        int root_y = Find(y);
        if (root_x != root_y) {
            // 按秩合并
            if (rank[root_x] > rank[root_y]) {
                parent[root_y] = root_x;
            } else if (rank[root_x] < rank[root_y]) {
                parent[root_x] = root_y;
            } else {
                parent[root_y] = root_x;
                ++rank[root_x];
            }
        }
    }
};
```

5. 最小生成树算法实现

5.1. Prim算法（邻接矩阵实现）

Prim算法从任意一个节点出发，逐步扩展生成树，每次选择边权最小的边加入树中。以下是基于邻接矩阵的 Prim 算法实现：

```
void MinSpanTree::PrimWithMatrix() {
    mst_edges.Clear();
    mst_cost = 0;
    Vector<bool> visited(vertices, false); // 记录节点是否被访问
    Vector<int> min_weight(vertices, INT_MAX); // 到生成树的最小边权值
    Vector<int> parent(vertices, -1); // 记录最小生成树每个节点的父节点
    min_weight[0] = 0; // 从第一个节点开始

    for (int i = 0; i < vertices; ++i) {
        int u = -1, min_edge = INT_MAX;

        // 找到未访问中权值最小的顶点
```

```
        for (int v = 0; v < vertices; ++v) {
            if (!visited[v] && min_weight[v] < min_edge) {
                u = v;
                min_edge = min_weight[v];
            }
        }
        if (u == -1) break; // 没有可以加入的节点
        visited[u] = true; // 标记为已访问
        mst_cost += min_edge; // 累加最小边权值
        if (parent[u] != -1) { // u 有父节点
            mst_edges.PushBack({ parent[u], u });
        }

        // 更新相邻节点的最小权值
        for (int v = 0; v < vertices; ++v) {
            if (graph_matrix[u][v] != 0 && !visited[v] && graph_matrix[u][v] < min_weight[v]) {
                min_weight[v] = graph_matrix[u][v];
                parent[v] = u;
            }
        }
    }
}
```

5.2. Prim算法（邻接表实现）

以下是基于邻接表的 Prim 算法实现，利用优先队列来管理待处理的节点。

```
void MinSpanTree::PrimWithList() {
    mst_edges.Clear();
    mst_cost = 0;
    Vector<bool> visited(vertices, false); // 记录节点是否被访问
    Vector<int> min_weight(vertices, INT_MAX); // 到生成树的最小边权值
    Vector<int> parent(vertices, -1); // 记录最小生成树每个节点的父节点
    min_weight[0] = 0;

    PriorityQueue<pair<int, int>, PairGreater> pq;
    pq.Push({ 0, 0 }); // 初始节点，权值为0

    while (!pq.Empty()) {
        int weight = pq.Top().first;
        int u = pq.Top().second;
        pq.Pop();

        if (visited[u]) continue;
        visited[u] = true;
        mst_cost += weight; // 累加当前边的权重

        if (parent[u] != -1) {
            mst_edges.PushBack({ parent[u], u });
        }
```

```

        // 遍历当前节点的邻居
        for (auto& neighbor : graph_list[u]) {
            int v = neighbor.first;
            int w = neighbor.second;
            if (!visited[v] && w < min_weight[v]) {
                min_weight[v] = w;
                parent[v] = u;
                pq.Push({ w, v });
            }
        }
    }
}
```

5.3. Kruskal算法

Kruskal算法将图中的所有边按权重排序，逐一加入生成树，并用并查集判断是否会形成环。

```
void MinSpanTree::Kruskal() {
    mst_edges.Clear();
    mst_cost = 0;

    UnionFind uf(vertices);

    // 将所有边按权值排序
    Vector<Edge> edges;
    for (int u =

 0; u < vertices; ++u) {
        for (auto& neighbor : graph_list[u]) {
            edges.PushBack({ u, neighbor.first, neighbor.second });
        }
    }
    sort(edges.begin(), edges.end(), [](const Edge& a, const Edge& b) {
        return a.weight < b.weight;
    });

    for (auto& edge : edges) {
        int u = edge.u, v = edge.v, w = edge.weight;
        if (uf.Find(u) != uf.Find(v)) {
            uf.Union(u, v);
            mst_edges.PushBack({ u, v });
            mst_cost += w;
        }
    }
}
```

6. 流程图

为了更清晰地展示如何求解最小生成树问题，以下是一个概括项目实现的流程图。流程图将涵盖从用户输入到最小生成树计算的整个过程。

6.1. 程序启动

程序首先启动并展示主界面，提示用户输入图的顶点数量、边信息，并选择要使用的最小生成树算法。

6.2. 用户输入数据

用户输入顶点数、边的信息，包括各个小区之间的连接和对应的电网造价。

6.3. 选择算法

用户选择一种最小生成树算法（Prim 算法或 Kruskal 算法）来计算最小生成树。

6.4. 执行算法

根据用户选择的算法，程序会调用相应的实现（Prim 或 Kruskal），计算最小生成树，并输出结果。

6.5. 显示结果

程序计算出最小生成树后，显示计算的结果，包括最小生成树的边和总造价。

以下是对应的流程图：

```
+------------------------+
|        程序启动        |
+------------------------+
            |
            v
+------------------------+
|  输入顶点数量和边信息    |  <--- 用户输入图的节点和边信息
+------------------------+
            |
            v
+------------------------+
|  选择最小生成树算法      |  <--- 用户选择 Prim 算法或 Kruskal 算法
+------------------------+
            |
            v
+---------------------------+
| 执行所选算法(Prim/Kruskal) |
+---------------------------+
            |
            v
+------------------------+
|  输出最小生成树结果      |  <--- 显示计算得到的最小生成树的边和总造价
+------------------------+
```

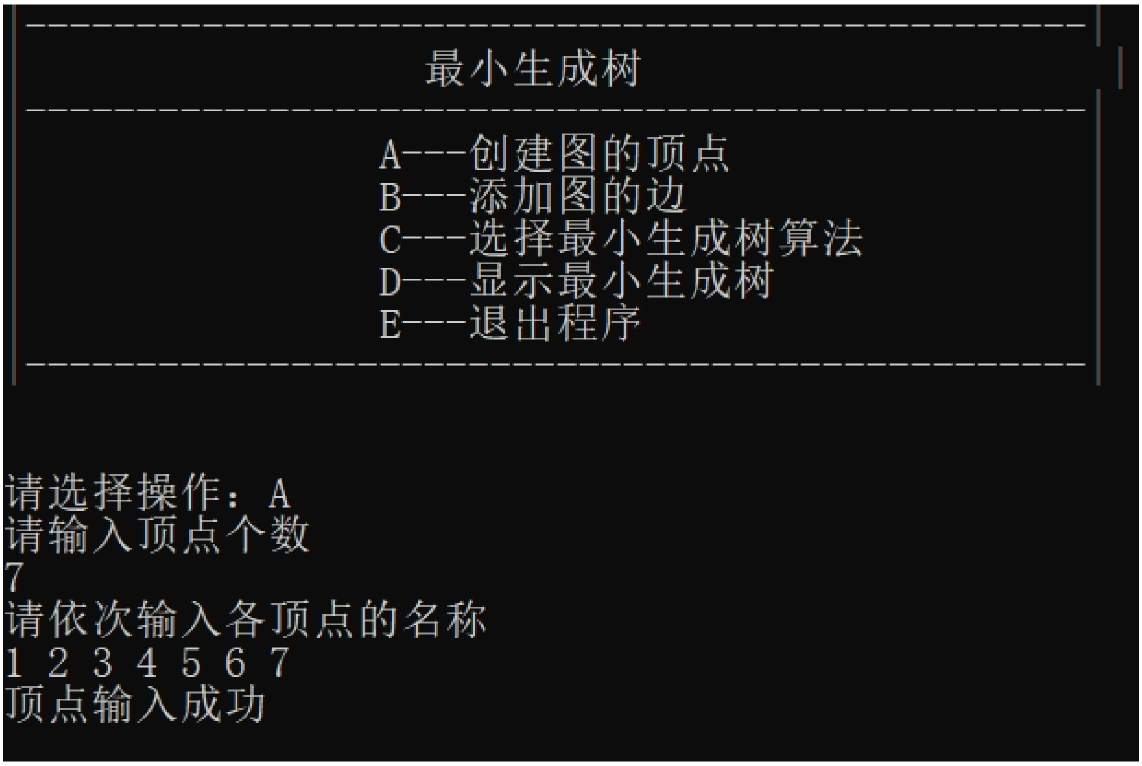
6.6. 说明：

- **用户输入**：在这一步，用户输入图的数据，包括顶点数量、每条线路的两个端点和线路的造价。

- **选择算法**：用户可以选择两种常见的算法之一来解决最小生成树问题：Prim算法（适用于稠密图）或者Kruskal算法（适用于稀疏图）。
- **执行算法**：根据用户选择的算法，程序会调用相应的实现：
    - **Prim算法**会通过逐步扩展最小生成树，选择最小权值的边。
    - **Kruskal算法**会先对所有的边进行排序，然后按权重从小到大逐个选边并检查是否会形成环，最终得到最小生成树。
- **输出结果**：计算完成后，程序输出生成树中的所有边，并计算出最小生成树的总造价。

7. 测试结果

7.1. Windows平台

- 创建图的顶点
- 

```
--------------------------------------------------
                    最小生成树
--------------------------------------------------
                 A---创建图的顶点
                 B---添加图的边
                 C---选择最小生成树算法
                 D---显示最小生成树
                 E---退出程序
--------------------------------------------------


请选择操作：A
请输入顶点个数
7
请依次输入各顶点的名称
1 2 3 4 5 6 7
顶点输入成功
```

- 添加图的边

- ```
  请选择操作：B
  请输入两个边顶点名称及边的权值(输入0结束)
  1 2 20
  请输入两个边顶点名称及边的权值(输入0结束)
  2 3 15
  请输入两个边顶点名称及边的权值(输入0结束)
  3 4 3
  请输入两个边顶点名称及边的权值(输入0结束)
  4 5 17
  请输入两个边顶点名称及边的权值(输入0结束)
  5 6 28
  请输入两个边顶点名称及边的权值(输入0结束)
  6 1 23
  请输入两个边顶点名称及边的权值(输入0结束)
  1 7 1
  请输入两个边顶点名称及边的权值(输入0结束)
  2 7 4
  请输入两个边顶点名称及边的权值(输入0结束)
  3 7 9
  请输入两个边顶点名称及边的权值(输入0结束)
  4 7 16
  请输入两个边顶点名称及边的权值(输入0结束)
  5 7 25
  请输入两个边顶点名称及边的权值(输入0结束)
  6 7 36
  请输入两个边顶点名称及边的权值(输入0结束)
  0
  输入完毕
  ```
- Prim算法（邻接矩阵实现）
  ```

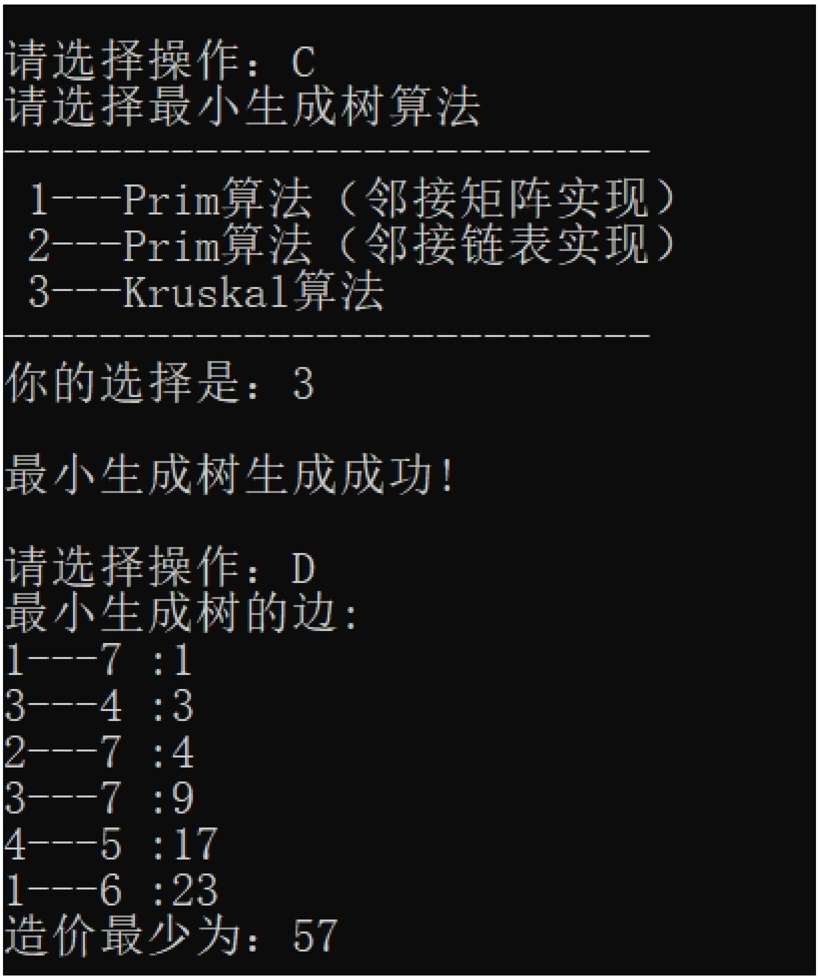
-
```
请选择操作: C
请选择最小生成树算法
----------------------------
 1---Prim算法（邻接矩阵实现）
 2---Prim算法（邻接链表实现）
 3---Kruskal算法
----------------------------
你的选择是: 1

最小生成树生成成功!

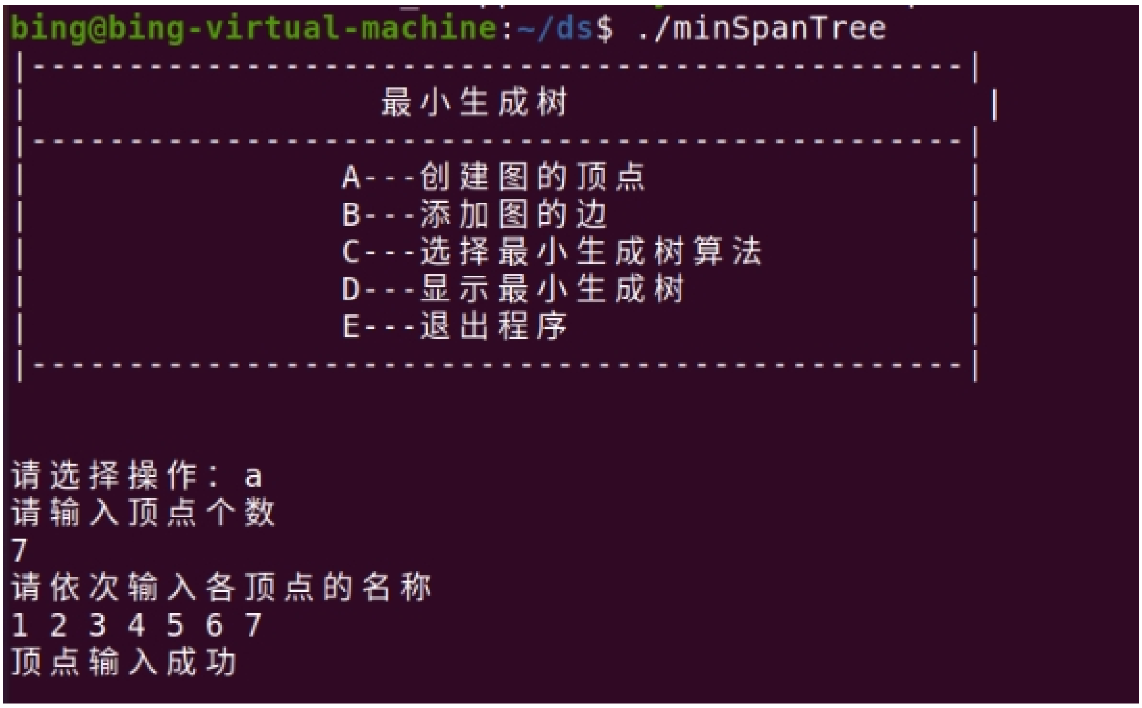
请选择操作: D
最小生成树的边:
1---7 :1
7---2 :4
7---3 :9
3---4 :3
4---5 :17
1---6 :23
造价最少为: 57
```
- Prim算法（邻接链表实现）
-
```
请选择操作: C
请选择最小生成树算法
----------------------------
 1---Prim算法（邻接矩阵实现）
 2---Prim算法（邻接链表实现）
 3---Kruskal算法
----------------------------
你的选择是: 2

最小生成树生成成功!

请选择操作: D
最小生成树的边:
1---7 :1
7---2 :4
7---3 :9
3---4 :3
4---5 :17
1---6 :23
造价最少为: 57
```
- Kruskal算法

- 

7.2. Linux平台

- 创建图的顶点
- 

- 添加图的边

-
```
请选择操作: B
请输入两个边顶点名称及边的权值(输入0结束)
1 2 20
请输入两个边顶点名称及边的权值(输入0结束)
2 3 15
请输入两个边顶点名称及边的权值(输入0结束)
3 4 3
请输入两个边顶点名称及边的权值(输入0结束)
4 5 17
请输入两个边顶点名称及边的权值(输入0结束)
5 6 28
请输入两个边顶点名称及边的权值(输入0结束)
6 1 23
请输入两个边顶点名称及边的权值(输入0结束)
1 7 1
请输入两个边顶点名称及边的权值(输入0结束)
2 7 4
请输入两个边顶点名称及边的权值(输入0结束)
3 7 9
请输入两个边顶点名称及边的权值(输入0结束)
4 7 16
请输入两个边顶点名称及边的权值(输入0结束)
5 7 25
请输入两个边顶点名称及边的权值(输入0结束)
6 7 36
请输入两个边顶点名称及边的权值(输入0结束)
0
输入完毕
```
- Prim算法（邻接矩阵实现）
-
```
请选择操作: C
请选择最小生成树算法
---------------------------
 1---Prim算法 (邻接矩阵实现)
 2---Prim算法 (邻接链表实现)
 3---Kruskal算法
---------------------------
你的选择是: 1

最小生成树生成成功!

请选择操作: D
最小生成树的边:
1---7 :1
7---2 :4
7---3 :9
3---4 :3
4---5 :17
1---6 :23
造价最少为: 57
```
- Prim算法（邻接链表实现）

-

```
请选择操作: C
请选择最小生成树算法
---------------------------
 1---Prim算法(邻接矩阵实现)
 2---Prim算法(邻接链表实现)
 3---Kruskal算法
---------------------------
你的选择是: 2

最小生成树生成成功!

请选择操作: D
最小生成树的边:
1---7 :1
7---2 :4
7---3 :9
3---4 :3
4---5 :17
1---6 :23
造价最少为: 57
```

- Kruskal算法
-

```
请选择操作: C
请选择最小生成树算法
---------------------------
 1---Prim算法(邻接矩阵实现)
 2---Prim算法(邻接链表实现)
 3---Kruskal算法
---------------------------
你的选择是: 3

最小生成树生成成功!

请选择操作: D
最小生成树的边:
1---7 :1
3---4 :3
2---7 :4
3---7 :9
4---5 :17
1---6 :23
造价最少为: 57
```

8. 心得体会

在本项目中，我深入理解了最小生成树（MST）的基本概念，并实现了两种主流的算法：**Prim算法**和**Kruskal算法。**通过对比这两种算法，我认识到：

1. **Prim算法**适用于稠密图，尤其是在图的边数较多时，使用邻接矩阵或者邻接表能显著提升效率。

2. **Kruskal算法**则更适合处理稀疏图，它通过将所有边排序，利用并查集处理图的连通性，从而高效地找到最小生成树。

此外，项目还涉及到一些重要的数据结构，如**优先队列**和**并查集**，它们分别帮助我们实现了Prim和Kruskal算法的高效性。通过实现这些算法和数据结构，我对图论的基本算法有了更深入的理解。

在实现过程中，我还设计了一个简单的命令行界面，使得用户可以通过输入顶点、边信息来选择最小生成树算法进行计算，并输出结果。

总的来说，整个项目不仅帮助我加深了对最小生成树问题的理解，还提升了我在数据结构和算法设计方面的能力。