《算法设计与分析》

课程实验报告

专业：计算机科学与技术
班级：2021211304
姓名：杨晨
学号：2021212171

1 概述

1.1 实验内容

编程实现下述 3 个算法，并利用给定的数据，验证算法正确性，三选一

1. 哈夫曼编码
    - 针对“附件 2. 哈夫曼编码输入文本-23”给出的文本信息，构造哈夫曼编码，文本中的数字 0-9、空格、标点符号、控制符参与编码，英文字母区分大小写
    - 统计 26 个英文字母、数字 0-9、空格、标点符号、控制符的出现的频率
    - 对 a, b, c,..,x, y, z, 0,···,9, 空格, 标点符号，设计哈夫曼编码
    - 按照哈夫曼编码，对输入文本重新编码
    - 统计采用哈夫曼编码，输入文本需要的存储比特数，并与定长编码方式需要的存储比特数进行比较
2. 单源最短路径
    - 利用“附件 1-1. 基站图的邻接矩阵-v1-23”给出的 LTE 网络基站数据，以基站为顶点，以基站间的距离连线为边，组成图，计算图中的单源最短路径
    - 图构造
      从昆明 LTE 网络中，选取部分基站，计算基站间的距离，在部分基站间引入边，得到图
        – 对 22 个基站顶点组成的图，以基站 567443 为源点
          计算 567443 到其它各点的单源最短路径
          计算 567443 到 33109 的最短路径
        – 对 42 个基站顶点组成的图，以基站 565845 为源点
          计算 565845 到其它各点的单源最短路径
          计算 565845 到 565667 的最短路径
3. 最小生成树
    - 利用“附件 1-1. 基站图的邻接矩阵-v1-23”给出的 LTE 网络基站数据，以基站为顶点，以基站间的距离连线为边，组成图，计算图的最小生成树
    - 图构造
      从昆明 LTE 网络中，选取部分基站，计算基站间的距离，在部分基站间引入边，得到
      22 个基站顶点组成的图
      42 个基站顶点组成的图
    - 采用 K 算法，或 P 算法，二选一
    - 给出最小生成树的成本/代价/耗费 cost
    - 做图，展现最小生成树
      * 注：可以在原图上，用红色、粗线条，标记最小生成树

1.2 开发环境

- Windows10

- PyCharm 2023.2.4 (Professional Edition)
- Visual Studio Code 1.84.2

2 实验过程

2.1 哈夫曼编码

2.1.1 介绍

哈夫曼编码是一种用于数据压缩的算法。它通过对不同字符赋予不同的编码，使得出现频率较高的字符具有较短的编码，从而达到压缩数据的目的。哈夫曼编码的基本思想是根据字符的频率构建一棵二叉树，频率越高的字符离根节点越近，频率越低的字符离根节点越远。在构建的二叉树中，从根节点到叶子节点的路径上的编码表示对应字符。

2.1.2 算法描述

下面是使用 C++ 实现的哈夫曼编码算法的伪代码：

Algorithm 1 哈夫曼编码算法

```
procedure BUILDTREE(freq)
    创建优先队列 pq
    for all (c, f) in freq do
        if c 为控制符 then
            continue
        创建新的 HuffmanNode 节点 n，设置 n 的字符为 c，频率为 f，左子节点和右子节点为空
        将 n 插入 pq
    while pq 的大小大于 1 do
        从 pq 中取出频率最小的两个节点 left 和 right
        创建新的 HuffmanNode 节点 parent，设置 parent 的字符为空字符，频率为 left 和 right 的频率之和，左子节点为 left，右子节点为 right
        将 parent 插入 pq
    return pq 中剩下的唯一节点作为根节点
procedure GENERATECODES(root, code, codes)
    if root 为空 then
        return
    if root 的左子节点和右子节点都为空 then
        将 code 赋值给 codes[root]，表示 root 的编码为 code
        return
    调用 GENERATECODES(root 的左子节点, code + "0", codes)
    调用 GENERATECODES(root 的右子节点, code + "1", codes)
```

2.1.3 分析

哈夫曼编码算法的时间复杂度主要集中在两个步骤：构建优先队列 pq 和生成编码字典 $codes$。假设输入文本中有 n 个不同的字符。

在构建优先队列的过程中，需要将 n 个节点插入到优先队列中，每次插入的时间复杂度为 $O(\log n)$。因此，构建优先队列的时间复杂度为 $O(n \log n)$。

在生成编码字典的过程中，对于每个字符，需要在哈夫曼树上进行遍历以找到对应的编码。由于哈夫曼树是一棵二叉树，最坏情况下需要遍历树的高度，即 $O(\log n)$。因此，生成编码字典的时间复杂度为 $O(n \log n)$。

综上所述，哈夫曼编码算法的总时间复杂度为 $O(n \log n)$。

2.1.4 运行结果

```
字符    频率    哈夫曼编码        定长编码
B       1       1111000010100     000000
```

```
P       1       1111000010111   000001
M       1       1111000010101   000010
D       4       11110001110     000011
L       1       001011001000    000100
"       2       101010011000    000101
H       2       001011001001    000110
S       8       1010100101      000111
G       3       00101100111     001000
F       3       00101100110     001001
z       4       11110000010     001010
O       3       00101100101     001011
x       8       1111000100      001100
C       5       11110001111     001101
j       2       111100001000    001110
/       2       111100000111    001111
T       15      101010001       010000
'       2       101010011010    010001
b       103     011110          010010
y       129     101011          010011
1       11      001011000       010100
l       291     11101           010101
g       176     00110           010110
,       55      0111110         010111
f       78      001010          011000
I       7       1010100001      011001
2       2       101010011001    011010
m       243     10100           011011
r       372     0101            011100
4       2       111100001001    011101
o       470     1001            011110
        1078    110             011111
w       77      1111001         100000
a       422     0110            100001
\n      26      00101111        100010
3       2       101010011011    100011
n       317     11111           100100
t       513     1011            100101
W       8       1010100100      100110
-       23      00101101        100111
h       265     11100           101000
i       442     1000            101001
.       66      1010101         101010
s       364     0100            101011
e       677     000             101100
)       8       1111000011      101101
d       158     111101          101110
(       8       1111000000      101111
```

```
c         212      01110             110000
5         2        111100000110      110001
p         186      00111             110010
:         6        1010100000        110011
u         167      00100             110100
k         26       00101110          110101
;         1        1111000010110     110110
v         59       0111111           110111
A         9        1111000101        111000
6         8        1010100111        111001
q         9        1111000110        111010
哈夫曼编码存储比特数： 31635比特
定长编码存储比特数： 42870比特
哈夫曼编码相较定长编码节省空间26.2071%
```

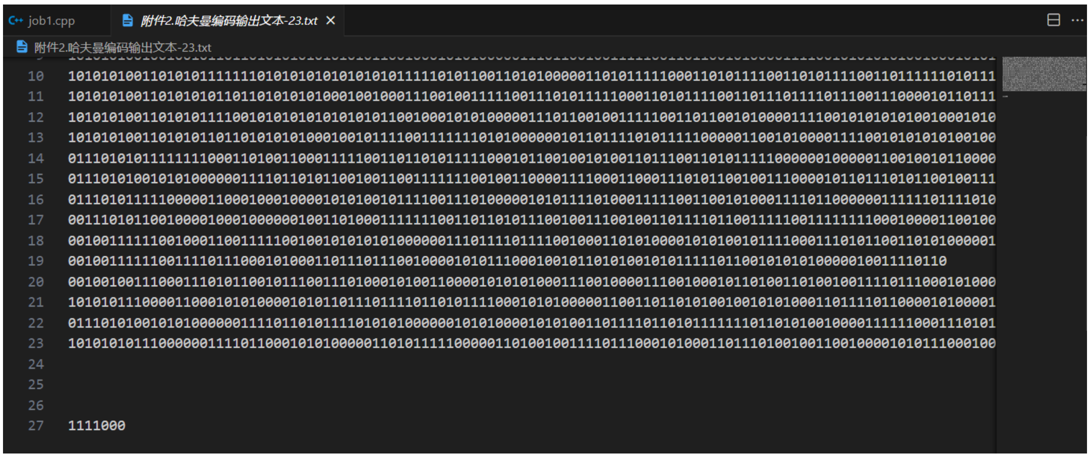


图 1: 哈夫曼编码后的输出文本

2.1.5 结论

哈夫曼编码是一种有效的数据压缩算法，通过根据字符频率分配不同的编码来实现数据的压缩。该算法的时间复杂度为 $O(n \log n)$，其中 n 是不同字符的数量。通过比较哈夫曼编码和定长编码的存储比特数，我们可以看到哈夫曼编码在频率较高的字符上具有更好的压缩效果。因此，哈夫曼编码在数据压缩和存储方面具有广泛的应用前景。

2.2 单源最短路径

2.2.1 介绍

单源最短路径问题是在给定一个加权有向图中寻找从一个固定起点到所有其他顶点的最短路径的问题。该问题在实际应用中具有广泛的应用，例如路由算法、地图导航等。下面是 Dijkstra 算法，它是解决单源最短路径问题的经典算法之一。

2.2.2 算法描述

Dijkstra 算法是一种贪心算法，用于解决单源最短路径问题。它通过逐步确定从起点到每个顶点的最短路径，并将其保存在一个距离数组中。

Algorithm 2 Dijkstra 算法

```
procedure DIJKSTRA(adjacency_matrix,start)
    n ← 邻接矩阵的大小
    初始化 dis 为长度为 n 的数组，初始值为正无穷    ▷ dis[i] 表示从 start 到 i 的最短距离
    dis[start] ← 0
    初始化 visited 为长度为 n 的数组，初始值为 False
    for i ← 0 to n − 1 do
        u ← −1
        min_dis ← 正无穷
        for j ← 0 to n − 1 do
            if not visited[j] and dis[j] < min_di then
                u ← j
                min_di ← dis[j]
        if u = −1 then
            break
        visited[u] ← True
        for v ← 0 to n − 1 do
            if adjacency_matrix[u][v] = −1 then
                continue
            if not visited[v] and dis[v] > dis[u] + adjacency_matrix[u][v] then
                dis[v] ← dis[u] + adjacency_matrix[u][v]
    if len(visited)! = n then
        print 不连通
        return None
    return dis
```

2.2.3 分析和改进

原始的 Dijkstra 算法的时间复杂度为 $O(V^2)$，其中 V 是顶点的数量。然而，通过使用优先队列（例如堆）来维护候选顶点集合，我们可以改进算法的时间复杂度。

以下是改进后的 Dijkstra 算法的伪代码：

Algorithm 3 改进后的 Dijkstra 算法

procedure DIJKSTRA_SAVING_TIME($adjacency\_matrix$,$start$)
　　初始化 dis 为长度为 n 的数组，初始值为正无穷
　　$dis[start] \leftarrow 0$
　　初始化 $visited$ 为长度为 n 的数组，初始值为 False
　　初始化空的优先队列 $heap$
　　将 $(0.0, start)$ 插入 $heap$
　　while $heap$ 不为空 **do**
　　　　$(d, u) \leftarrow$ 从 $heap$ 中弹出最小元素
　　　　if $visited[u]$ **then**
　　　　　　continue
　　　　$visited[u] \leftarrow$ True
　　　　for $v \leftarrow 0$ to $n - 1$ **do**
　　　　　　if $adjacency\_matrix[u][v] = -1$ **then**
　　　　　　　　continue
　　　　　　if not $visited[v]$ and $dis[v] > dis[u] + adjacency\_matrix[u][v]$ **then**
　　　　　　　　$dis[v] \leftarrow dis[u] + adjacency\_matrix[u][v]$
　　　　　　　　将 $(dis[v], v)$ 插入 $heap$
　　if $len(visited)! = n$ **then**
　　　　$print$ 不连通
　　　　return $None$
　　return dis

改进后的 Dijkstra 算法的时间复杂度为 $O((V + E)\log V)$，其中 V 是顶点的数量，E 是边的数量。通过使用优先队列来选择下一个最短路径顶点，我们减少了在每次迭代中查找最小距离顶点的时间。

2.2.4 运行结果

```
以 567443 为起点，到各点的最短距离为
33109: 1956.93
565696: 1343.41
566631: 761.94
566720: 2111.29
566742: 302.54
566747: 1988.14
566750: 683.09
566751: 1622.91
566783: 344.55
566798: 1778.06
566802: 963.85
```

```
566967: 1562.25
566993: 988.63
566999: 2072.92
567203: 1592.31
567238: 780.89
567260: 244.05
567322: 1582.91
567439: 1309.05
567443: 0.00
567547: 1733.00
568098: 810.56
以 567443 为起点，到 33109 的最短距离为:1956.93
经验证，优化后的 dijkstra 算法正确

以 565845 为起点，到各点的最短距离为
565675: 1369.37
565621: 1928.90
565667: 2900.12
567510: 645.04
565801: 1153.11
566010: 403.43
567891: 2401.90
565492: 2223.01
565558: 2171.29
565627: 2697.46
565572: 2440.92
565610: 2025.89
565859: 2050.98
565630: 1468.96
565559: 2381.34
565845: 0.00
565527: 2594.34
565633: 2347.84
565496: 2308.24
565865: 2489.07
565773: 2281.46
567531: 1402.79
565516: 1918.10
565393: 2339.03
565753: 1122.45
33566: 2169.68
566074: 1573.64
565648: 1997.17
567526: 488.24
565551: 1806.75
565631: 843.92
565608: 1883.38
```

```
567500: 1055.67
565531: 2161.48
565562: 853.57
32788: 2187.66
567497: 1561.46
566316: 2592.69
568056: 2787.20
565964: 741.61
567618: 1655.16
565898: 978.43
以 565845 为起点，到 565667 的最短距离为:2900.12
经验证，优化后的 dijkstra 算法正确
```

2.2.5 结论

上面介绍了单源最短路径问题及其解决方案，介绍了 Dijkstra 算法的原始版本和改进版本，并分析了它们的时间复杂度。通过使用优先队列，改进后的 Dijkstra 算法能够更高效地找到最短路径。在实际应用中，Dijkstra 算法被广泛应用于解决单源最短路径问题。

2.3 最小生成树

2.3.1 介绍

最小生成树是图论中的一个重要概念，用于找到一个连通图的最小权重生成树。最小生成树在很多应用中都有广泛的应用，例如网络设计、电路设计、城市规划等。下面将介绍两种常用的最小生成树算法：Kruskal 算法和 Prim 算法。

2.3.2 算法描述

Kruskal 算法是一种基于边的贪心算法，用于构建最小生成树。它的基本思想是从图的边集中选择具有最小权重的边，然后逐步扩展生成树，直到生成树包含了所有的顶点。下面是 Kruskal 算法的伪代码描述：

Algorithm 4 Kruskal 算法

```
function KRUSKAL(adjacency_matrix)    ▷ 最小生成树，Kruskal 算法，时间复杂度 O(E log E)
    function FIND(parent, i)    ▷ 查找节点 i 的根节点
        while parent[i] ≠ i do
            i ← parent[i]
        return i
    procedure UNION(parent, i, j)    ▷ 合并 i 和 j 所在的集合
        i_root ← find(parent, i)
        j_root ← find(parent, j)
        if i_root ≠ j_root then
            parent[i_root] ← j_root
    n ← len(adjacency_matrix)
    edges_list ← []
    for i ← 0 到 n − 1 do
        for j ← 0 到 n − 1 do
            if adjacency_matrix[i][j] ≠ −1 then
                edges_list.append((i, j, adjacency_matrix[i][j]))
    edges_list.sort(key = λx : x[2])
    edges ← []
    min_cost ← 0
    parent ← [i for i in range(n)]
    for u, v, cost in edges_list do
        if find(parent, u) ≠ find(parent, v) then
            union(parent, u, v)
            min_cost ← min_cost + cost
            edges.append((u, v))
        if len(edges) = n − 1 then
            break
    if len(edges) ≠ n − 1 then
        print('不连通')
        return None
    return min_cost, edges
```

Prim 算法是一种基于顶点的贪心算法，用于构建最小生成树。它的基本思想是从一个顶点开始，逐步选择与当前生成树连接的具有最小权重的边所连接的顶点，直到生成树包含了所有的顶点。下面是 Prim 算法的伪代码描述：

Algorithm 5 Prim 算法

```
function PRIM(adjacency_matrix)                ▷ 最小生成树，Prim 算法，时间复杂度 O(n^2)
    Input: 邻接矩阵 adjacency_matrix
    Output: 最小生成树的权值
    n ← len(adjacency_matrix)
    min_cost ← 0
    lowcost ← [∞] ∗ n
    lowcost[0] ← 0
    visited ← [False] ∗ n
    edges ← []                                  ▷ 最小生成树的边
    for _ ← 0 to n − 1 do
        min_dis ← ∞
        u ← −1
        for i in range(n) do
            if not visited[i] and lowcost[i] < min_dis then
                min_dis ← lowcost[i]
                u ← i
        if u = −1 then
            break
        pre_u ← next(pre_u for pre_u in range(n) if adjacency_matrix[pre_u][u] = min_dis)
        edges.append((pre_u, u))
        min_cost ← min_cost + min_dis
        visited[u] ← True
        for v in range(n) do
            if adjacency_matrix[u][v] = −1 then
                continue
            if not visited[v] and adjacency_matrix[u][v] < lowcost[v] then
                lowcost[v] ← adjacency_matrix[u][v]
    if len(visited) ≠ n then
        print(' 不连通')
        return None
    return min_cost, edges
```

2.3.3 分析和改进

Kruskal 算法的时间复杂度为 $O(E\log E)$，其中 E 是边的数量。算法首先对边的集合进行排序，时间复杂度为 $O(E\log E)$；然后通过并查集的操作判断边是否形成环路，并将不形成环路的边添加到最小生成树中，时间复杂度为 $O(E\alpha(V))$，其中 $\alpha(V)$ 是 Ackermann 函数的反函数，近

似于 $O(\log V)$。综合考虑，Kruskal 算法使用并查集后的时间复杂度可以近似看作 $O(E\log E+\alpha(V))$。由于通常 E 远小于 V^2，因此可以进一步简化为 O(ElogV)。

Prim 算法的时间复杂度为 $O(n^2)$，其中 n 是图中顶点的数量。算法的时间复杂度主要来自于每次找到未访问顶点中的最小 lowcost 值的操作，需要遍历 n 个顶点，每次遍历需要 $O(n)$ 的时间。同时，更新最小距离、标记顶点访问状态以及添加边的操作都可以在常数时间内完成。

然而，如果图是稀疏的，即顶点数 n 远大于边数 E，可以考虑使用优先队列（例如最小堆）来维护未访问顶点中的最小 lowcost 值，从而将时间复杂度降低，这样可以更快地找到最小值。伪代码如下：

Algorithm 6 堆优化 Prim 算法

```
function PRIM(adjacency_matrix)
    n ← len(adjacency_matrix)
    min_cost ← 0
    lowcost ← [∞] * n
    lowcost[0] ← 0
    visited ← [False] * n
    heap ← [(0.0, 0)]                                        ▷ (权值, 下标)
    edges ← []                                               ▷ 最小生成树的边
    while heap is not empty do
        (cost, u) ← heappop(heap)
        if visited[u] then
            continue
        min_cost ← min_cost + cost
        visited[u] ← True
        pre_u ← next((pre_u for pre_u in range(n) if adjacency_matrix[pre_u][u] = cost), None)
        edges.append((pre_u, u))
        for v in range(n) do
            if adjacency_matrix[u][v] = −1 then
                continue
            if not visited[v] and adjacency_matrix[u][v] < lowcost[v] then
                lowcost[v] ← adjacency_matrix[u][v]
                heappush(heap, (lowcost[v], v))
    if len(visited) ≠ n then
        print(' 不连通')
        return None
    return min_cost, edges
```

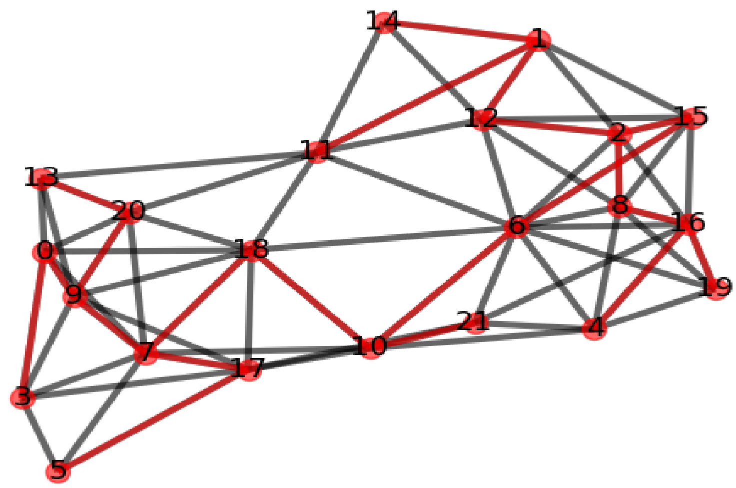
优化后的 Prim 算法的时间复杂度为 $O((V+E)\log V)$，其中 V 是顶点的数量，E 是边的数量。算法使用了优先队列来选择具有最小权重的边，每次从队列中取出具有最小权重的边的时

间复杂度为 $O(\log V)$。在每次选择边后，需要更新与新加入的顶点相连的边的权重，并将其插入优先队列中，时间复杂度为 $O(E\log V)$。因此，总的时间复杂度为 $O((V+E)\log V)$。

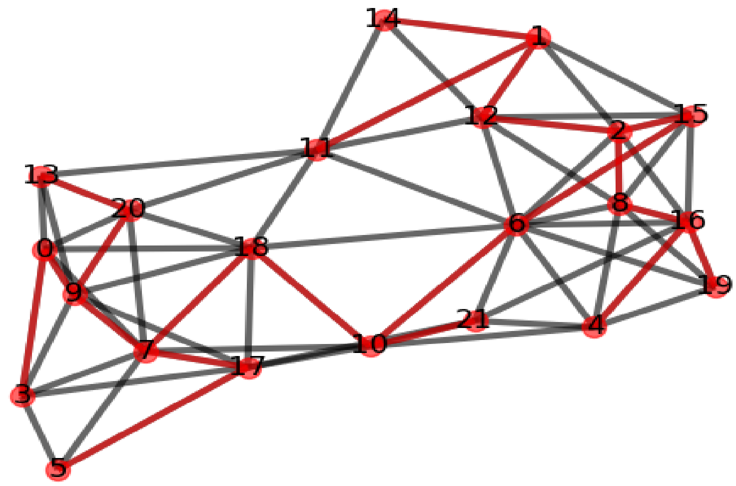
2.3.4 运行结果

```
Prim算法最小生成树的权值: 6733.57
Kruskal算法最小生成树的权值: 6733.57
经检验, Prim算法和Prim_saving_time算法结果相同
```

数据集1的Prim算法

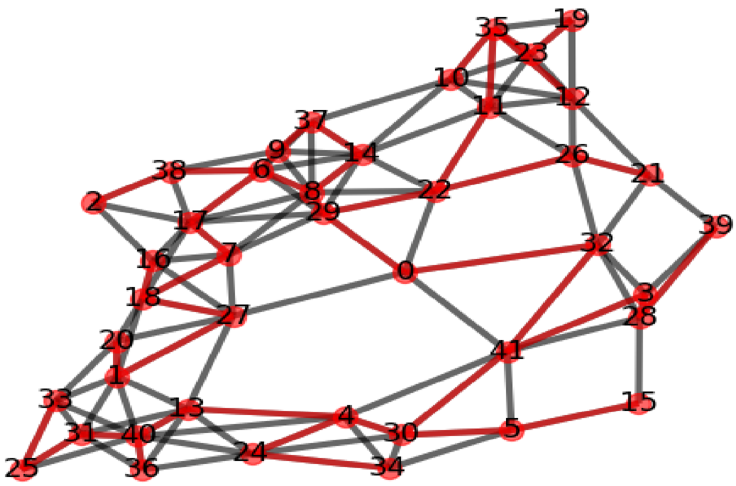


数据集1的Kruskal算法

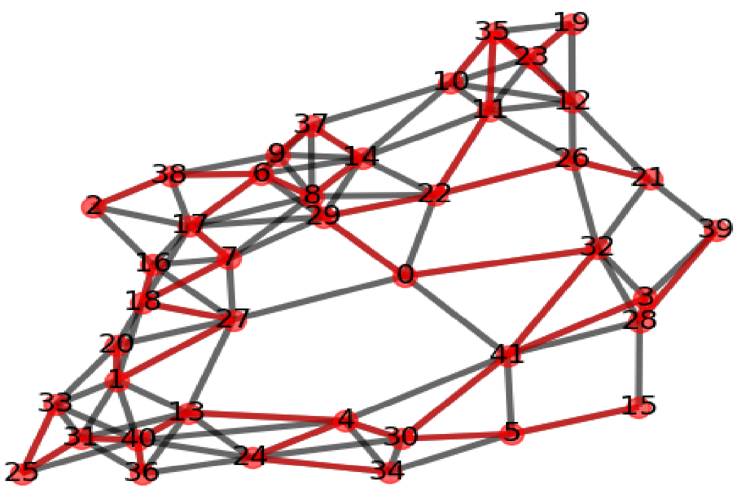


```
Prim算法最小生成树的权值: 13027.03
Kruskal算法最小生成树的权值: 13027.03
经检验, Prim算法和Prim_saving_time算法结果相同
```

数据集2的Prim算法



数据集2的Kruskal算法



2.3.5 结论

综上所述，Prim 算法和 Kruskal 算法都是用于解决最小生成树问题的常见算法。两种算法都可以用于求解最小生成树问题，选择哪种算法取决于具体的应用场景和问题要求。Prim 算法在稠密图上的效果较好，而 Kruskal 算法在稀疏图上的效果较好。

3 附录：完整代码

3.1 哈夫曼编码

```
#include <iostream>
#include <fstream>
#include <string>
#include <unordered_map>
#include <vector>
#include <queue>
#include <bitset>
#include <iomanip>
#include <cmath>

void input(std::unordered_map<char, int> &freq)
{
    std::ifstream file("附件2.哈夫曼编码输入文本-23.txt");

    if (!file.is_open())
    {
        std::cout << "无法打开文件！" << std::endl;
        return;
    }

    char c;
    while (file.get(c))
    {
        if (freq.find(c) == freq.end())
        {
            freq[c] = 1;
        }
        else
        {
            freq[c]++;
        }
    }

    file.close();
}
```

```cpp
class HuffmanNode
{
public:
    char character;
    int frequency;
    HuffmanNode *left;
    HuffmanNode *right;
    // 构造函数
    HuffmanNode(char c, int freq) : character(c), frequency(freq), left(nullptr),
        right(nullptr) {}
};

struct CompareNodes // 用于优先队列的比较函数，使得优先队列中的节点按照频率从小到大
    排列
{
    bool operator()(HuffmanNode *a, HuffmanNode *b)
    {
        return a->frequency > b->frequency;
    }
};

HuffmanNode *buildTree(std::unordered_map<char, int> &freq)
{
    std::priority_queue<HuffmanNode *, std::vector<HuffmanNode *>, CompareNodes> pq
        ;
    for (auto &p : freq)
    {
        pq.push(new HuffmanNode(p.first, p.second));
    }

    while (pq.size() > 1)
    {
        HuffmanNode *left = pq.top();
        pq.pop();
        HuffmanNode *right = pq.top();
        pq.pop();
        HuffmanNode *parent = new HuffmanNode('\0', left->frequency + right->
            frequency);
        parent->left = left;
        parent->right = right;
        pq.push(parent);
    }

    return pq.top();
}

void generateCodes(HuffmanNode *root, std::string code, std::unordered_map<char,
```

```cpp
    std::string> &codes)
{
    if (root == nullptr)
    {
        return;
    }

    if (root->left == nullptr && root->right == nullptr)
    {
        codes[root->character] = code;
        return;
    }

    generateCodes(root->left, code + "0", codes);
    generateCodes(root->right, code + "1", codes);
}

void printFrequenciesAndCodes(std::unordered_map<char, int> &freq, std::::
    unordered_map<char, std::string> &codes)
{
    int totalBitsHuffman = 0;
    int totalBitsFixed = 0;
    int bitLength = ceil(log2(freq.size())); // 定长编码的二进制数的位数
    int count = 0;                           // 用于计算定长编码的二进制数

    std::cout << std::left << std::setw(8) << "字符" << std::setw(8) << "频率" <<
        std::setw(16) << "哈夫曼编码" << std::setw(16) << "定长编码" << std::endl;
    for (auto &p : freq)
    {
        char character = p.first;
        int frequency = p.second;

        std::string huffmanCode = codes[character];
        std::string fixedCode = std::bitset<8>(count++).to_string();
        fixedCode = fixedCode.substr(fixedCode.length() - bitLength, bitLength);

        totalBitsHuffman += frequency * huffmanCode.length();
        totalBitsFixed += frequency * bitLength;

        if (character == '\n') // 换行符
        {
            std::cout << std::left << std::setw(8) << "\\n" << std::setw(8) <<
                frequency << std::setw(16) << huffmanCode << std::setw(16) <<
                fixedCode << std::endl;
            continue;
        }
```

```
        std::cout << std::left << std::setw(8) << character << std::setw(8) <<
            frequency << std::setw(16) << huffmanCode << std::setw(16) << fixedCode
            << std::endl;
    }

    std::cout << "哈夫曼编码存储比特数: " << totalBitsHuffman << "比特" << std::
        endl;
    std::cout << "定长编码存储比特数: " << totalBitsFixed << "比特" << std::endl;
    std::cout << "哈夫曼编码相较定长编码节省空间" << (1 - (double)totalBitsHuffman
        / totalBitsFixed) * 100 << "%" << std::endl;
}

void encode(std::unordered_map<char, int> &freq, std::unordered_map<char, std::
    string> &codes)
{
    std::ifstream file("附件2.哈夫曼编码输入文本-23.txt");

    if (!file.is_open())
    {
        std::cout << "无法打开文件! " << std::endl;
        return;
    }

    std::ofstream fileEncoded("附件2.哈夫曼编码输出文本-23.txt");

    if (!fileEncoded.is_open())
    {
        std::cout << "无法打开文件! " << std::endl;
        return;
    }

    char c;
    while (file.get(c))
    {
        if (c == '\n') // 控制符不参与编码
        {
            fileEncoded << c;
            continue;
        }
        fileEncoded << codes[c];
    }

    file.close();
    fileEncoded.close();
}

int main()
```

```
{
    std::unordered_map<char, int> freq;
    input(freq);
    HuffmanNode *root = buildTree(freq);
    std::unordered_map<char, std::string> codes;
    generateCodes(root, "", codes);
    printFrequenciesAndCodes(freq, codes);
    encode(freq, codes);
    return 0;
}
```

3.2 单源最短路径

```
import pandas as pd
import heapq
import numpy as np


def init_data(num):
    """
    初始化数据
    :param num: 数据集编号,[1,2]
    :return: 邻接矩阵, 下标对应的基站id
    """
    df = pd.read_excel('附件1-1.基站图的邻接矩阵-v1-23.xls', sheet_name='数据' +
        num, header=None)
    size = 0
    if num == '1':
        size = 24
    elif num == '2':
        size = 44
    # 邻接矩阵
    adjacency_matrix = df.values[2:size, 2:size].astype(float)
    # 下标对应的基站id
    station_ids = df.values[2:size, 1].astype(int)
    return adjacency_matrix, station_ids


# 单源最短路径dijkstra算法, 时间复杂度O(V^2)
def dijkstra(adjacency_matrix, start):
    """
    单源最短路径dijkstra算法, 时间复杂度O(V^2)
    :param adjacency_matrix: 邻接矩阵
    :param start: 起点
    :return: dis[i]表示从start到i的最短距离
    """
```

```python
    n = len(adjacency_matrix)
    # 初始化
    dis = [float('inf')] * n
    dis[start] = 0
    visited = [False] * n

    for i in range(n):
        u = -1
        min_dis = float('inf')
        for j in range(n):
            if not visited[j] and dis[j] < min_dis:
                u = j
                min_dis = dis[j]
        if u == -1:
            break
        visited[u] = True
        for v in range(n):
            if adjacency_matrix[u][v] == -1:
                continue
            if not visited[v] and dis[v] > dis[u] + adjacency_matrix[u][v]:
                dis[v] = dis[u] + adjacency_matrix[u][v]

    if len(visited) != n:
        print('不连通')
        return None

    return dis


# 单源最短路径dijkstra算法, 时间复杂度O((V+E)logV)
def dijkstra_saving_time(adjacency_matrix, start):
    """
    单源最短路径dijkstra算法, 时间复杂度O((V+E)logV)
    :param adjacency_matrix: 邻接矩阵
    :param start: 起点
    :return: dis[i]表示从start到i的最短距离
    """
    n = len(adjacency_matrix)
    # 初始化
    dis = [float('inf')] * n
    dis[start] = 0
    visited = [False] * n

    heap = [(0.0, start)]

    while heap:
        d, u = heapq.heappop(heap)
```

```
        if visited[u]:
            continue
        visited[u] = True
        for v in range(n):
            if adjacency_matrix[u][v] == -1:
                continue
            if not visited[v] and dis[v] > dis[u] + adjacency_matrix[u][v]:
                dis[v] = dis[u] + adjacency_matrix[u][v]
                heapq.heappush(heap, (dis[v], v))

    if len(visited) != n:
        print('不连通')
        return None

    return dis


def solve(num, start, end):
    """
    解决问题
    :param num: 数据集编号，[1,2]
    :param start: 起点id
    :param end: 终点id
    """
    adjacency_matrix, station_ids = init_data(num)
    start = np.where(station_ids == start)[0][0]
    end = np.where(station_ids == end)[0][0]
    dist = dijkstra(adjacency_matrix, start)
    print(f'\033[93m以{station_ids[start]}为起点，到各点的最短距离为\033[0m')
    for i in range(len(dist)):
        print(f'{station_ids[i]}:␣{dist[i]:.2f}')
    print(f'\033[92m以{station_ids[start]}为起点，到{station_ids[end]}的最短距离为:{dist[end]:.2f}\033[0m')
    # 验证优化后的dijkstra算法
    new_dist = dijkstra_saving_time(adjacency_matrix, start)
    if dist == new_dist:
        print('\033[92m经验证,␣优化后的dijkstra算法正确\033[0m')
    else:
        print('\033[91m经验证,␣优化后的dijkstra算法错误\033[0m')
    print()


if __name__ == '__main__':
    solve('1', 567443, 33109)
    solve('2', 565845, 565667)
```

3.3 最小生成树

```python
import pandas as pd
import matplotlib.pyplot as plt
import networkx as nx
import heapq
import matplotlib

matplotlib.rc("font", family="SimHei", weight="bold")  # 解决中文乱码
plt.rcParams["axes.unicode_minus"] = False  # 解决负号无法显示的问题


def init_data(num):
    """
    初始化数据
    :param num: 数据集编号,[1,2]
    :return: 邻接矩阵, 下标对应的基站id
    """
    df = pd.read_excel('附件1-1.基站图的邻接矩阵-v1-23.xls', sheet_name='数据' +
        num, header=None)
    size = 0
    if num == '1':
        size = 24
    elif num == '2':
        size = 44
    # 邻接矩阵
    adjacency_matrix = df.values[2:size, 2:size].astype(float)
    # 下标对应的基站id
    station_ids = df.values[2:size, 1].astype(int)
    return adjacency_matrix, station_ids


def draw(adjacency_matrix, edges, title):
    """
    绘制最小生成树的图形
    :param adjacency_matrix: 邻接矩阵
    :param edges: 最小生成树的边
    :param title: 图形标题
    """
    G = nx.Graph()
    n = len(adjacency_matrix)

    # 添加顶点
    for i in range(n):
        G.add_node(i)

    min_edge = []
```

```python
    if edges[0][0] is None:  # 去掉第一个空边(None, 0)
        min_edge = edges[1:]
    else:  # Kruskal算法没有空边
        min_edge = edges

    # 添加边
    for i in range(n):
        for j in range(n):
            if adjacency_matrix[i][j] != -1:
                G.add_edge(i, j, weight=adjacency_matrix[i][j])

    # 绘制图形
    pos = nx.spring_layout(G, seed=1)
    fig, ax = plt.subplots()
    ax.set_title(title)
    nx.draw(G, pos, node_size=100, node_color='red', edge_color='black', width=3.0,
         alpha=0.6, ax=ax)
    nx.draw_networkx_labels(G, pos, font_size=13, font_color='black', alpha=1.0)

    # 绘制最小生成树的边
    nx.draw_networkx_edges(G, pos, edgelist=min_edge, width=3.0, alpha=0.6,
        edge_color='red')
    plt.savefig(title + '.png')
    plt.show()

# 最小生成树,Prim算法,O(n^2)
def Prim(adjacency_matrix):
    """
    最小生成树,Prim算法,O(n^2)
    :param adjacency_matrix: 邻接矩阵
    :return: 最小生成树的权值
    """
    n = len(adjacency_matrix)
    min_cost = 0
    # 初始化
    lowcost = [float('inf')] * n
    lowcost[0] = 0
    visited = [False] * n
    edges = []  # 最小生成树的边

    for _ in range(n):
        min_dis = float('inf')
        u = -1
        for i in range(n):
            if not visited[i] and lowcost[i] < min_dis:
                min_dis = lowcost[i]
```

```
                u = i
        if u == -1:
            break
        pre_u = next((pre_u for pre_u in range(n) if adjacency_matrix[pre_u][u] ==
            min_dis), None)
        edges.append((pre_u, u))
        min_cost += min_dis
        visited[u] = True
        for v in range(n):
            if adjacency_matrix[u][v] == -1:
                continue
            if not visited[v] and adjacency_matrix[u][v] < lowcost[v]:
                lowcost[v] = adjacency_matrix[u][v]

    if len(visited) != n:
        print('不连通')
        return None

    return min_cost, edges


# 最小生成树,Prim算法,O((V+E)logV)
def Prim_saving_time(adjacency_matrix):
    """
    最小生成树,Prim算法,O((V+E)logV)
    :param adjacency_matrix: 邻接矩阵
    :return: 最小生成树的权值
    """
    n = len(adjacency_matrix)
    min_cost = 0
    # 初始化
    lowcost = [float('inf')] * n
    lowcost[0] = 0
    visited = [False] * n
    heap = [(0.0, 0)]  # (权值, 下标)
    edges = []  # 最小生成树的边

    while heap:
        cost, u = heapq.heappop(heap)
        if visited[u]:
            continue
        min_cost += cost
        visited[u] = True
        pre_u = next((pre_u for pre_u in range(n) if adjacency_matrix[pre_u][u] ==
            cost), None)
        edges.append((pre_u, u))
        for v in range(n):
```

```
            if adjacency_matrix[u][v] == -1:
                continue
            if not visited[v] and adjacency_matrix[u][v] < lowcost[v]:
                lowcost[v] = adjacency_matrix[u][v]
                heapq.heappush(heap, (lowcost[v], v))

    if len(visited) != n:
        print('不连通')
        return None

    return min_cost, edges


# 最小生成树,Kruskal算法,O(ElogV)
def Kruskal(adjacency_matrix):
    """
    最小生成树,Kruskal算法,O(ElogE)
    :param adjacency_matrix: 邻接矩阵
    :return: 最小生成树的权值
    """

    def find(parent, i):
        """
        查找i的根节点
        :param parent: parent[i]表示i的父节点
        :param i: 节点i
        :return: i的根节点
        """
        while parent[i] != i:
            i = parent[i]
        return i

    def union(parent, i, j):
        """
        合并i和j所在的集合
        :param parent: parent[i]表示i的父节点
        :param i: 节点i
        :param j: 节点j
        """
        i_root = find(parent, i)
        j_root = find(parent, j)
        if i_root != j_root:
            parent[i_root] = j_root

    n = len(adjacency_matrix)
    edges_list = []
    # 初始化
```

```
    for i in range(n):
        for j in range(n):
            if adjacency_matrix[i][j] != -1:
                edges_list.append((i, j, adjacency_matrix[i][j]))
    edges_list.sort(key=lambda x: x[2])
    edges = []  # 最小生成树的边
    min_cost = 0
    parent = [i for i in range(n)]

    for u, v, cost in edges_list:
        if find(parent, u) != find(parent, v):
            union(parent, u, v)
            min_cost += cost
            edges.append((u, v))

        if len(edges) == n - 1:
            break

    if len(edges) != n - 1:
        print('不连通')
        return None

    return min_cost, edges


def solve(num):
    adjacency_matrix, station_ids = init_data(num)
    min_cost, edges = Prim(adjacency_matrix)
    print('Prim算法最小生成树的权值: {:.2f}'.format(min_cost))
    draw(adjacency_matrix, edges, '数据集' + num + '的Prim算法')
    min_cost, edges = Kruskal(adjacency_matrix)
    print('Kruskal算法最小生成树的权值: {:.2f}'.format(min_cost))
    draw(adjacency_matrix, edges, '数据集' + num + '的Kruskal算法')
    min_cost_saving_time, edges = Prim_saving_time(adjacency_matrix)
    if min_cost - min_cost_saving_time < 1e-9:
        print('经检验, Prim算法和Prim_saving_time算法结果相同')
    print()


if __name__ == '__main__':
    solve('1')
    solve('2')
```