《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：张梓良

学号：2021212484

算法设计与分析（第 4 章）：贪心
编程实验报告

张梓良 2021212484

日期：2023 年 12 月 12 日

1 概述

1.1 实验内容及要求

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。

1. 哈夫曼编码
   - 编程实现方案 2
   - 程序运行结果需给出
     (a) 符号表 {a, b, c,..,x, y, z, 0,⋯,9, ⋯} 中各编码成员（字符）在文本中的出现频率，以及其哈夫曼编码
     (b) 分别采用哈夫曼编码、定长编码时，输入文本所需要的存储比特数
2. 单源最短路径
   - 对 22 个基站顶点组成的图，以基站 567443 为源点
     (a) 计算 567443 到其它各点的单源最短路径
     (b) 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
     (a) 计算 565845 到其它各点的单源最短路径
     (b) 计算 565845 到 565667 的最短路径
3. 最小生成树
   - 采用 Prim 算法和 Kruskal 算法分别实现
   - 给出最小生成树的成本/代价/耗费 cost
   - 做图，展现最小生成树

1.2 实验环境

- gcc version 8.1.0
- Visual Studio Code 1.82.2
- python 3.9.13
- Anaconda 22.9.0
- OS: Windows_NT x64 10.0.22621

2 哈夫曼编码

2.1 算法设计

2.1.1 算法逻辑

构造哈夫曼树

1. 初始化：由给定的 n 个权值构造 n 棵只有一个根节点的二叉树，得到一个二叉树集合 F。
2. 选取与合并：从二叉树集合 F 中选取根节点权值最小的两棵二叉树分别作为左右子树构造一棵新的二叉树，这棵新二叉树的根节点的权值为其左、右子树根结点的权值和。
3. 删除与加入：从 F 中删除作为左、右子树的两棵二叉树，并将新建立的二叉树加入到 F 中。
4. 重复 2、3 步，当集合中只剩下一棵二叉树时，这棵二叉树就是哈夫曼树。

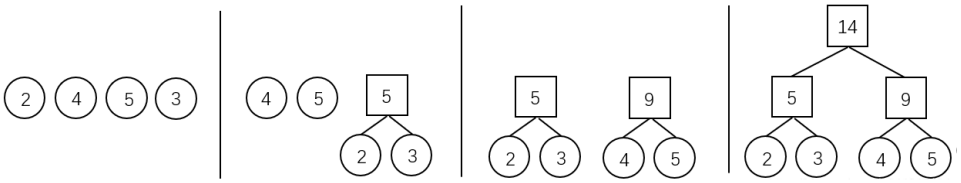

图 1: 哈夫曼树的构造过程

哈夫曼编码

1. 设需要编码的字符集为：$d_1, d_2, \ldots, d_n$，他们在字符串中出现的频率为：$w_1, w_2, \ldots, w_n$。
2. 以 $d_1, d_2, \ldots, d_n$ 作为叶结点，$w_1, w_2, \ldots, w_n$ 作为叶结点的权值，构造一棵霍夫曼树。
3. 规定哈夫曼编码树的左分支代表 0，右分支代表 1，则从根结点到每个叶结点所经过的路径组成的 0、1 序列即为该叶结点对应字符的编码。

2.1.2 算法伪代码

Algorithm 1: Huffman Coding with Priority Queue

```
Function Huffman(char_freq):
    PriorityQueue ← create an empty priority queue;
    for char ∈ char_freq do
        n ← create a new node with char and char_freq[char] as data and frequency;
        insert n into PriorityQueue;
    end
    while |PriorityQueue| > 1 do
        n1 ← extract min from PriorityQueue;
        n2 ← extract min from PriorityQueue;
        n ← create a new node with NULL as data and n1.frequency + n2.frequency as
          frequency;
        n.left ← n1;
        n.right ← n2;
        insert n into PriorityQueue;
    end
    HuffmanTree ← extract min from PriorityQueue;
    Codes ← an empty map;
    generate_codes(HuffmanTree, empty_string);
    return Codes;
Function generate_codes(Node, Code):
    if Node.data is not NULL then
        Codes[Node.data] ← Code;
    end
    if Node.left is not NULL then
        generate_codes(Node.left, Code + "0");
    end
    if Node.right is not NULL then
        generate_codes(Node.right, Code + "1");
    end
char_freq ← { 'a': 5, 'b': 9, 'c': 12, 'd': 13, 'e': 16, 'f': 45 };
result ← Huffman(char_freq);
```

2.1.3 核心代码

优先队列

采用小根堆实现优先队列。对于堆的维护主要涉及**新节点的插入**和**提取根节点**两个操作。

节点插入

```
void insertNode(vector<HuffmanNode *> &heap, HuffmanNode *node, int &heapSize)
{
    heapSize++;
    int i = heapSize - 1;
    heap.push_back(node);

    // 如果节点比其父节点小，则交换二者位置并递归调用插入节点操作
    while (i != 0 && heap[(i - 1) / 2]->frequency > heap[i]->frequency)
    {
        swap(heap[i], heap[(i - 1) / 2]);
        i = (i - 1) / 2;
    }
}
```

提取根节点

```
// 小根堆化操作
void minHeapify(vector<HuffmanNode *> &heap, int i, int heapSize)
{
    int smallest = i;
    int left = 2 * i + 1;
    int right = 2 * i + 2;

    // 找出左右子节点中最小的节点
    if (left < heapSize && heap[left]->frequency < heap[smallest]->frequency)
        smallest = left;

    if (right < heapSize && heap[right]->frequency < heap[smallest]->frequency)
        smallest = right;

    // 如果最小的节点不是当前节点，则交换二者位置并递归调用小根堆化操作
    if (smallest != i)
    {
        swap(heap[i], heap[smallest]);
        minHeapify(heap, smallest, heapSize);
    }
}

// 提取最小节点操作
HuffmanNode *extractMin(vector<HuffmanNode *> &heap, int &heapSize)
{
    HuffmanNode *root = heap[0];
    heap[0] = heap[heapSize - 1];
    heapSize--;
    heap.pop_back();
    minHeapify(heap, 0, heapSize);
```

```
    return root;
}
```

构造哈夫曼树

```
void HuffmanCodes(unordered_map<char, int> freq)
{
    vector<HuffmanNode *> heap;
    int heapSize = 0;

    // 将字符和对应的频率放入小根堆中
    for (const auto &p : freq)
    {
        heap.push_back(new HuffmanNode{p.first, p.second, nullptr, nullptr});
        heapSize++;
    }

    while (heapSize > 1)
    {
        // 提取两个最小节点并合并为一个新节点
        HuffmanNode *left = extractMin(heap, heapSize);
        HuffmanNode *right = extractMin(heap, heapSize);
        HuffmanNode *top = new HuffmanNode{'$', left->frequency + right->frequency,
            left, right};

        // 将新节点插入小根堆中
        insertNode(heap, top, heapSize);
    }

    // 保存哈夫曼编码
    saveCodes(heap[0], "");
}
```

2.2 算法优化及性能分析

上述哈夫曼编码算法采用优先队列优化，下面比较优化和未优化的算法时空复杂度。

时间复杂度

- 不优化的哈夫曼编码的时间复杂度为$O(n^2)$，其中 n 是符号的个数。在不使用优先队列的情况下，需要每次遍历频率数组来找到频率最小的两个符号，然后合并它们。这样的操作需要进行 $n-1$ 次，每次都需要线性搜索频率数组，因此总体时间复杂度为 $O(n^2)$
- 优先队列优化的哈夫曼编码的时间复杂度为$O(n \log n)$。这是因为在构建哈夫曼树时，使用了优先队列来维护符号频率的顺序，每次从队列中取出两个最小频率的符号进行合并，再将合并后的节点插入队列。优先队列的操作包括插入和弹出操作，时间复杂度为 $O(\log n)$。因此，总体时间复杂度为 $O(n \log n)$。

空间复杂度

- 不优化的哈夫曼编码的空间复杂度为$O(n)$，其中 n 是符号的个数。在不使用优先队列的情况下，只需要一个数组来存储符号及其对应的频率，数组的大小与符号个数相等。
- 优先队列优化的哈夫曼编码的空间复杂度为$O(n)$。这是因为需要使用优先队列来存储符号及其对应的频率，队列的大小与符号个数相等。

2.3 结果展示及分析

```
======================Huffman Codes=======================
<n 317> 11111        <d 158> 111101       <O 3> 111100011111
<3 2> 111100011110   <F 3> 111100011101   </ 2> 111100011100
<G 3> 111100011011   <L 1> 1111000110101  <M 1> 1111000110100
<P 1> 1111000110011  <B 1> 1111000110010  <5 2> 111100011000
<A 9> 1111000101     <6 8> 1111000011     <D 4> 11110000101
<z 4> 11110000100    <W 8> 1111000001     <w 77> 1111001
<( 8> 1111000000     <l 291> 11101        <h 265> 11100
<c 212> 01110        <a 422> 0110         <r 372> 0101
<b 103> 011110       <s 364> 0100         <g 176> 00110
<p 186> 00111        <\n 26> 00101111     <T 15> 011111101
<y 129> 101011       <k 26> 00101110      <- 23> 00101101
<f 78> 001010        <e 677> 000          <C 5> 0010110000
<: 6> 0010110001     <u 167> 00100        <1 11> 001011001
<, 55> 0111110       <H 2> 011111100000   <m 243> 10100
<2 2> 011111100001   <v 59> 1010100       <' 2> 01111110001
<I 7> 0111111001     <; 1> 011111110000   <q 9> 1111000100
<" 2> 011111110001   <j 2> 011111110010   <4 2> 011111110011
<) 8> 0111111110     <x 8> 0111111101     <S 8> 0111111111
<. 66> 1010101       <i 442> 1000         <  1078> 110
<o 470> 1001         <t 513> 1011


==================Compare==================
The length of Huffman codes is 31637bits.
The length of equal-length codes is 42870bits.
Saved 26.2025% space.
```

`<n 317> 11111` 表示符号为 `n` ，出现的频率为 `317` ,哈夫曼编码为 `11111` 。可以发现对于给定文本采用哈夫曼编码能够比采用等长编码节省约 26.2025% 的存储空间。

3 单源最短路径

3.1 算法设计

3.1.1 算法逻辑

1. 将结点分成两个集合：已确定最短路长度的点集（记为 S 集合）的和未确定最短路长度的点集（记为 T 集合）。一开始所有的点都属于 T 集合。
2. 初始化 $dis(s) = 0$，其他点的 dis 均为 $+\infty$。
3. 然后重复这些操作：
   (a) 从 T 集合中，选取一个最短路长度最小的结点，移到 S 集合中。
   (b) 对那些刚刚被加入 S 集合的结点的所有出边执行松弛操作。
4. 直到 T 集合为空，算法结束。

3.1.2 算法伪代码

Algorithm 2: Dijkstra's algorithm with priority queue

Function `Dijkstra(`G, s`)`**:**
  $dist[s] \leftarrow 0$;
  Create an empty priority queue pq;
  Insert $(s, 0)$ into pq;
  while pq *is not empty* **do**
    $(u, d) \leftarrow$ Extract minimum element from pq;
    for *each neighbor* v *of* u **do**
      $new\_dist \leftarrow dist[u] + weight(u, v)$;
      if $new\_dist < dist[v]$ **then**
        $dist[v] \leftarrow new\_dist$;
        Insert $(v, dist[v])$ into pq;
      end
    end
  end
  return $dist$;

3.1.3 核心代码

```
void dijkstra(int src)
{
    for (int i = 1; i <= n; i++)
        dist[i] = 0x3f3f3f3f; // 初始化距离为无穷大
    memset(st, 0, sizeof st);
    dist[src] = 0;
    priority_queue<PII, vector<PII>, greater<PII>> heap; // 使用优先队列优化
```

```
    heap.push({0, src});
    path[index_id[src]] = vector<int>(1, index_id[src]); // 初始化路径

    while (!heap.empty())
    {
        auto t = heap.top();
        heap.pop();

        int ver = t.second;
        double distance = t.first;
        if (!st[ver])
        {
            st[ver] = true;
            for (int i = h[ver]; i != -1; i = ne[i])
            {
                int j = e[i];
                if (dist[j] > distance + w[i])
                {
                    dist[j] = distance + w[i];
                    heap.push({dist[j], j});
                    // 更新路径
                    path[index_id[j]] = path[index_id[ver]];
                    path[index_id[j]].push_back(index_id[j]);
                }
            }
        }
    }
}
```

3.2 算法优化及性能分析

有多种方法来维护最短路长度最小的结点，不同的实现导致了 Dijkstra 算法时间复杂度上的差异。在上述算法中采用优先队列进行优化。

时间复杂度

- 暴力：不使用任何数据结构进行维护，直接在 T 集合中暴力寻找最短路长度最小的结点。2 操作总时间复杂度为 $O(m)$，1 操作总时间复杂度为 $O(n^2)$，全过程的时间复杂度为 $O(n^2+m)=O(n^2)$。（此处的操作 1、2 指的是算法逻辑描述中重复的操作）
- 二叉堆：每成功松弛一条边 (u,v)，就将 v 插入二叉堆中（如果 v 已经在二叉堆中，直接修改相应元素的权值即可），1 操作直接取堆顶结点即可。共计 $O(m)$ 次二叉堆上的插入（修改）操作，$O(n)$ 次删除堆顶操作，而插入（修改）和删除的时间复杂度均为 $O(\log n)$，时间复杂度为 $O((n+m)\log n)=O(m\log n)$。
- 优先队列：和二叉堆类似，但使用优先队列时，如果同一个点的最短路被更新多次，因为先前更新时插入的元素不能被删除，也不能被修改，只能留在优先队列中，故优先队列内

的元素个数是 $O(m)$ 的，时间复杂度为 $O(m \log m)$。

- Fibonacci 堆：和前面二者类似，但 Fibonacci 堆插入的时间复杂度为 $O(1)$，故时间复杂度为 $O(n \log n + m)$，时间复杂度最优。
- 线段树：和二叉堆原理类似，不过将每次成功松弛后插入二叉堆的操作改为在线段树上执行单点修改，而 1 操作则是线段树上的全局查询最小值。时间复杂度为 $O(m \log n)$。

在稀疏图中，$m = O(n)$，使用二叉堆实现的 Dijkstra 算法较 Bellman–Ford 算法具有较大的效率优势；而在稠密图中，$m = O(n^2)$，这时候使用暴力做法较二叉堆实现更优。

空间复杂度

- 暴力：不使用任何数据结构进行维护，直接在 T 集合中暴力寻找最短路长度最小的结点。由于不使用额外的数据结构，空间复杂度为 $O(1)$。
- 二叉堆：使用二叉堆进行维护，需要额外的空间来存储二叉堆的结构，因此空间复杂度为 $O(n)$。
- 优先队列：使用优先队列进行维护，优先队列内的元素个数是 $O(m)$ 的，因此空间复杂度为 $O(m)$。
- Fibonacci 堆：使用 Fibonacci 堆进行维护，需要额外的空间来存储 Fibonacci 堆的结构，空间复杂度为 $O(n)$。
- 线段树：使用线段树进行维护，需要额外的空间来存储线段树的结构，空间复杂度为 $O(n)$。

3.3 结果展示及分析

```
====================================22====================================
<567443-33109>  distance: 1956.93 path: 567443->566750->567439->33109
<567443-565696> distance: 1343.41 path: 567443->566783->566993->565696
<567443-566631> distance: 761.94  path: 567443->566783->566631
<567443-566720> distance: 2111.29 path:
↪  567443->566750->567439->566751->566720
<567443-566742> distance: 302.54  path: 567443->566742
<567443-566747> distance: 1988.14 path:
↪  567443->566742->566802->567322->566747
<567443-566750> distance: 683.09  path: 567443->566750
<567443-566751> distance: 1622.91 path: 567443->566750->567439->566751
<567443-566783> distance: 344.55  path: 567443->566783
<567443-566798> distance: 1778.06 path: 567443->566750->567439->566798
<567443-566802> distance: 963.85  path: 567443->566742->566802
<567443-566967> distance: 1562.25 path: 567443->566783->566993->566967
<567443-566993> distance: 988.63  path: 567443->566783->566993
<567443-566999> distance: 2072.92 path:
↪  567443->566783->566993->566967->566999
```

```
<567443-567203> distance: 1592.31 path: 567443->566783->566993->567203
<567443-567238> distance: 780.89  path: 567443->566783->567238
<567443-567260> distance: 244.05  path: 567443->567260
<567443-567322> distance: 1582.91 path: 567443->566742->566802->567322
<567443-567439> distance: 1309.05 path: 567443->566750->567439
<567443-567443> distance: 0.00    path: 567443
<567443-567547> distance: 1733.00 path: 567443->566750->567439->567547
<567443-568098> distance: 810.56  path: 567443->566742->568098

<567443-33109>  distance: 1956.93 path: 567443->566750->567439->33109

====================================42====================================
<565845-565675> distance: 1369.37 path: 565845->567526->567500->565675
<565845-565621> distance: 1928.90 path:
 ↪  565845->566010->565631->565801->565630->565621
<565845-565667> distance: 2900.12 path:
 ↪  565845->567526->567500->565675->565551->565633->565667
<565845-567510> distance: 645.04  path: 565845->567526->567510
<565845-565801> distance: 1153.11 path: 565845->566010->565631->565801
<565845-566010> distance: 403.43  path: 565845->566010
<565845-567891> distance: 2401.90 path:
 ↪  565845->567526->567500->565675->565551->567891
<565845-565492> distance: 2223.01 path:
 ↪  565845->567526->567500->565675->565551->565492
<565845-565558> distance: 2171.29 path:
 ↪  565845->567526->567500->565675->565551->565558
<565845-565627> distance: 2697.46 path:
 ↪  565845->567526->567500->565675->565551->565558->565627
<565845-565572> distance: 2440.92 path:
 ↪  565845->567526->567500->566074->565610->565572
<565845-565610> distance: 2025.89 path:
 ↪  565845->567526->567500->566074->565610
<565845-565859> distance: 2050.98 path:
 ↪  565845->567526->565964->567531->565859
<565845-565630> distance: 1468.96 path:
 ↪  565845->566010->565631->565801->565630
<565845-565559> distance: 2381.34 path:
 ↪  565845->567526->567500->565675->565516->565559
```

```
<565845-565845> distance: 0.00    path: 565845
<565845-565527> distance: 2594.34 path:
 ↪  565845->566010->565631->565801->565630->565648->565527
<565845-565633> distance: 2347.84 path:
 ↪  565845->567526->567500->565675->565551->565633
<565845-565496> distance: 2308.24 path:
 ↪  565845->566010->565631->565801->565630->565648->565496
<565845-565865> distance: 2489.07 path:
 ↪  565845->567526->565964->567531->565859->565865
<565845-565773> distance: 2281.46 path:
 ↪  565845->566010->565631->565801->565630->565621->565773
<565845-567531> distance: 1402.79 path: 565845->567526->565964->567531
<565845-565516> distance: 1918.10 path:
 ↪  565845->567526->567500->565675->565516
<565845-565393> distance: 2339.03 path:
 ↪  565845->567526->565964->567531->565859->565393
<565845-565753> distance: 1122.45 path: 565845->566010->565562->565753
<565845-33566>  distance: 2169.68 path:
 ↪  565845->566010->565562->565753->567618->33566
<565845-566074> distance: 1573.64 path: 565845->567526->567500->566074
<565845-565648> distance: 1997.17 path:
 ↪  565845->566010->565631->565801->565630->565648
<565845-567526> distance: 488.24  path: 565845->567526
<565845-565551> distance: 1806.75 path:
 ↪  565845->567526->567500->565675->565551
<565845-565631> distance: 843.92  path: 565845->566010->565631
<565845-565608> distance: 1883.38 path:
 ↪  565845->566010->565562->565753->567618->565608
<565845-567500> distance: 1055.67 path: 565845->567526->567500
<565845-565531> distance: 2161.48 path:
 ↪  565845->566010->565562->565753->567618->565531
<565845-565562> distance: 853.57  path: 565845->566010->565562
<565845-32788>  distance: 2187.66 path:
 ↪  565845->567526->565964->567531->565859->32788
<565845-567497> distance: 1561.46 path:
 ↪  565845->566010->565562->565753->567497
<565845-566316> distance: 2592.69 path:
 ↪  565845->567526->567500->565675->565551->565558->566316
```

```
<565845-568056> distance: 2787.20 path:
↪  565845->567526->567500->565675->565551->565633->568056
<565845-565964> distance: 741.61  path: 565845->567526->565964
<565845-567618> distance: 1655.16 path:
↪  565845->566010->565562->565753->567618
<565845-565898> distance: 978.43  path: 565845->566010->565898

<565845-565667> distance: 2900.12 path:
↪  565845->567526->567500->565675->565551->565633->565667
```

结果中输出了最短路的长度以及路径。

4 最小生成树-Prim

4.1 算法设计

4.1.1 算法逻辑

1. 设置顶点集合 $S = 1$，边集合 $T = \phi$
2. 当 $S \subset V$，即 S 是 V 的真子集时，作如下的贪心选择
   - 选取满足：$i \in S$，$j \in V - S$，且 $c[i][j]$ 最小的边 $< i, j >$，将顶点 j 添加到 S 中，边 $< i, j >$ 加到边集 T 中
3. 重复上述过程，直到 $S = V$ 为止

4.1.2 算法伪代码

Algorithm 3: Prim's Algorithm with Priority Queue

```
1  Function Prim(G, r):
2      PriorityQueue ← create an empty priority queue;
3      for each vertex v in G do
4          Key[v] ← ∞;
5          Parent[v] ← NULL;
6          insert v into PriorityQueue with priority Key[v];
7      end
8      Key[r] ← 0;
9      while PriorityQueue is not empty do
10         u ← ExtractMin(PriorityQueue);
11         for each neighbor v of u do
12             if v ∈ PriorityQueue and weight of edge (u, v) is less than Key[v] then
13                 Parent[v] ← u;
14                 Key[v] ← weight of edge (u, v);
15                 decrease priority of v in PriorityQueue to Key[v];
16             end
17         end
18     end
19     MST ← edges defined by Parent;
20     return MST;
21 graph ← adjacency list representation of the graph;
22 source ← source vertex for the MST;
23 result ← Prim(graph, source);
```

4.1.3 核心代码

```
double prim()
{
    memset(mst, -1, sizeof mst);
    memset(st, 0, sizeof st);
    for (int i = 1; i <= n; i++)
        dist[i] = INF;
    double res = 0;

    for (int i = 0; i < n; i++)
    {
        int t = -1;
        for (int j = 1; j <= n; j++)
            if (!st[j] && (t == -1 || dist[t] > dist[j]))
                t = j;

        // 图不连通
        if (i && fabs(dist[t] - INF) < 1e-6)
            return INF;

        // 加上树边权重
        if (i)
            res += dist[t];

        st[t] = true;

        // 更新其他点到最小生成树的距离
        for (int j = 1; j <= n; j++)
        {
            if (dist[j] > g[t][j] && !st[j])
            {
                dist[j] = g[t][j];
                mst[j] = t;
            }
        }
    }

    return res;
}
```

4.2 算法优化及性能分析

上述 prim 算法采用优先队列优化。prim 算法的优化和 dijkstra 算法的优化方式类似，下面只比较几种典型优化的时空复杂度。

时间复杂度

- 不优化的 Prim 算法的时间复杂度为$O(V^2)$。
- 优先队列优化的 Prim 算法的时间复杂度为$O((V+E)\log E)$。
- 二叉堆优化的 Prim 算法的时间复杂度为$O((V+E)\log V)$。

空间复杂度

- 不优化的 Prim 算法的空间复杂度为$O(1)$。
- 优先队列优化的 Prim 算法的空间复杂度为$O(E)$。
- 二叉堆优化的 Prim 算法的空间复杂度为$O(V)$。

4.3 结果展示及分析

```
==========22==========
Cost: 6733.57
2-13  3-16  4-1   5-17
6-18  7-11  8-10  9-3
10-1  11-19 12-2  13-3
14-21 15-2  16-7  17-9
18-8  19-8  20-17 21-10
22-11

==========42==========
Cost: 13027.03
2-28  3-39  4-42  5-31
6-31  7-9   8-18  9-30
10-38 11-36 12-23 13-36
14-5  15-9  16-6  17-19
18-7  19-8  20-24 21-2
22-27 23-30 24-36 25-5
26-32 27-23 28-19 29-4
30-1  31-42 32-41 33-1
34-26 35-25 36-12 37-41
38-15 39-7  40-29 41-14
42-33
```

Cost 给出了最小生成树的开销（保留到小数点后两位），下方给出了最小生成树中的边。

使用 `python` 代码生成最小生成树的示意图如下：

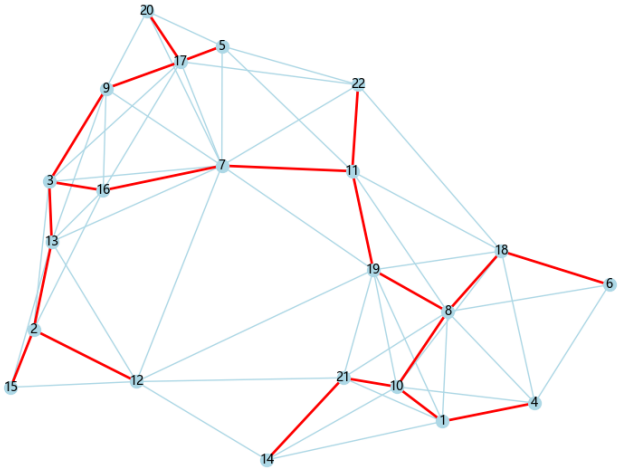

图 2: 22 个基站的最小生成树

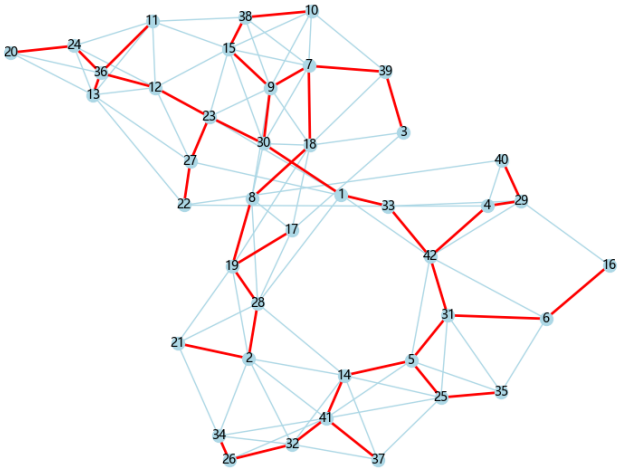

图 3: 42 个基站的最小生成树

5 最小生成树-Kruskal

5.1 算法设计

5.1.1 算法逻辑

1. 将 G 的 n 个顶点看成 n 个孤立的连通分支
2. 将所有的边按权从小到大排序
3. 从权最小的第一条边开始，依边权递增的顺序查看每一条边 (v,w)，并按下述方法连接 2 个不同的连通分支：当查看到第 k 条边 (v,w) 时，
   (a) 如果顶点 v 和 w 分别位于当前 2 个不同的连通分支 T1 和 T2 中，用边 (v,w) 将 T1 和 T2 合并成一个连通分支，然后继续查看后续第 $k+1$ 条边
   (b) 如果顶点 v 和 w 已经属于当前的同一个连通分支中，为防止产生回路，不允许将 (v,w) 加入最小生成树中。此时，直接再查看后续第 $k+1$ 条边
4. 持续上述过程，直至只剩下一个连通分支，该连通分支组成 G 的一个最小生成树

5.1.2 算法伪代码

Algorithm 4: Kruskal's Algorithm with Union-Find

```
Function Kruskal(G):
    SortEdges ← list of edges in G, sorted by weight;
    UF ← empty union-find data structure;
    for each vertex v in G do
        create a set containing only v;
        add this set to UF;
    end
    MST ← empty set of edges;
    for each edge (u, v) in SortEdges do
        if Find(u) ≠ Find(v) then
            add edge (u, v) to MST;
            Union(Find(u), Find(v));
        end
    end
    return MST;
graph ← adjacency list representation of the graph;
result ← Kruskal(graph);
```

5.1.3 核心代码

```
double kruskal()
{
```

```
    sort(edges, edges + m);
    double res = 0;
    int cnt = 0;
    for (int i = 1; i <= n; i++)
        p[i] = i; // 初始化并查集

    for (int i = 0; i < m; i++)
    {
        int a = edges[i].a, b = edges[i].b;
        double w = edges[i].w;

        int p_a = find(a), p_b = find(b);
        // 判断是否已经在一个连通图中
        if (p_a != p_b)
        {
            p[p_a] = p_b;
            res += w;
            mst[cnt] = {a, b};
            cnt++;
        }

        if (cnt == n - 1)
            return res;
    }

    return INF;
}
```

5.2 算法优化及性能分析

使用并查集优化的 Kruskal 算法的时间复杂度可以分为两个部分来分析：排序和并查集操作。

- 排序：对边的权重进行排序的时间复杂度为 $O(m \log m)$，其中 m 是边的数量。
- 并查集操作：对于每条边，我们需要执行 Find 和 Union 操作。对于 Find 操作，其平均时间复杂度可以接近 $O(1)$，而 Union 操作的平均时间复杂度也可以接近 $O(1)$，这是由于使用了路径压缩和按秩合并的优化策略。因此，对于 m 条边，执行并查集操作的时间复杂度为 $O(m\alpha(m,n))$，其中 $\alpha(m,n)$ 是反阿克曼函数的值，可以看作是一个极其缓慢增长的函数，可以近似看作常数级别。

综上所述，使用并查集优化的 Kruskal 算法的时间复杂度为$O((m \log m) + m\alpha(m,n))$。在稀疏图中，边的数量 m 远小于顶点的数量 n，因此可以将其近似为$O(m \log m)$。

算法的空间复杂度取决于并查集数据结构的实现方式，通常为$O(n)$。

5.3 结果展示及分析

得到的结果和 Prim 算法的结果相同。

6 总结

哈夫曼编码 是一种用于数据压缩的无损压缩算法。它根据字符出现的频率构建一颗最优的二叉树，使得高频字符用较短的编码表示，低频字符用较长的编码表示。哈夫曼编码可以实现无损压缩，即在解压缩时可以完全还原原始数据。时间复杂度为 $O(n \log n)$，其中 n 是字符的数量。

Dijkstra 算法 是解决单源最短路径问题的经典算法。它基于贪心策略，每次选择当前路径上最短的节点，并逐步扩展路径直到到达目标节点。Dijkstra 算法适用于有向图和无向图，但不能处理负权边。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。

Prim 算法 是解决最小生成树问题的经典算法。它从一个初始顶点开始，每次选择与当前生成树距离最近的顶点，并将其加入生成树中，直到所有顶点都被连接为止。Prim 算法适用于无向图，可以处理带权边。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。

Kruskal 算法 也是解决最小生成树问题的经典算法。它基于贪心策略，将所有边按权重排序，并依次选择未连接的边，如果不会形成环路，则加入最小生成树中。Kruskal 算法适用于无向图，可以处理带权边。时间复杂度为 $O(E \log E)$，其中 E 是边的数量。

这些算法在图论和数据压缩领域有着广泛的应用。哈夫曼编码可以有效地压缩数据，减小存储空间和传输带宽。Dijkstra 算法可以用于计算网络中的最短路径，如路由算法等。Prim 算法和 Kruskal 算法可以用于构建最小生成树，如网络设计、电力传输等。

A huffman.cpp

```cpp
/**
 * @file huffman.cpp
 * @author zhang ziliang (ziliangzhang@bupt.edu.cn)
 * @brief 哈夫曼编码
 * @date 2023-12-09
 */
#include <iostream>
#include <iomanip>
#include <vector>
#include <unordered_map>
#include <string>
#include <fstream>
#include <cmath>

using namespace std;

// 哈夫曼树节点的结构体
struct HuffmanNode
{
    char data;                  // 存储字符
    int frequency;              // 存储频率
    HuffmanNode *left, *right; // 左右子节点
};

unordered_map<char, pair<int, string>> codes; // 用于保存字符和对应的哈夫曼编码

// 小根堆化操作
void minHeapify(vector<HuffmanNode *> &heap, int i, int heapSize)
{
    int smallest = i;
    int left = 2 * i + 1;
    int right = 2 * i + 2;

    // 找出左右子节点中最小的节点
    if (left < heapSize && heap[left]->frequency < heap[smallest]->frequency)
        smallest = left;

    if (right < heapSize && heap[right]->frequency < heap[smallest]->frequency)
        smallest = right;

    // 如果最小的节点不是当前节点，则交换二者位置并递归调用小根堆化操作
    if (smallest != i)
    {
        swap(heap[i], heap[smallest]);
        minHeapify(heap, smallest, heapSize);
```

```
46      }
47  }
48
49  // 提取最小节点操作
50  HuffmanNode *extractMin(vector<HuffmanNode *> &heap, int &heapSize)
51  {
52      HuffmanNode *root = heap[0];
53      heap[0] = heap[heapSize - 1];
54      heapSize--;
55      heap.pop_back();
56      minHeapify(heap, 0, heapSize);
57      return root;
58  }
59
60  // 插入节点操作
61  void insertNode(vector<HuffmanNode *> &heap, HuffmanNode *node, int &heapSize)
62  {
63      heapSize++;
64      int i = heapSize - 1;
65      heap.push_back(node);
66
67      // 如果节点比其父节点小，则交换二者位置并递归调用插入节点操作
68      while (i != 0 && heap[(i - 1) / 2]->frequency > heap[i]->frequency)
69      {
70          swap(heap[i], heap[(i - 1) / 2]);
71          i = (i - 1) / 2;
72      }
73  }
74
75  // 保存哈夫曼编码
76  void saveCodes(HuffmanNode *root, string str)
77  {
78      if (!root)
79          return;
80
81      // 如果节点是叶子节点，则将字符和对应的编码保存到codes中
82      if (root->data != '$')
83          codes[root->data] = {root->frequency, str};
84
85      // 分别递归遍历左右子树，并将当前节点的编码加上'0'或'1'
86      saveCodes(root->left, str + "0");
87      saveCodes(root->right, str + "1");
88  }
89
90  // 构建哈夫曼树并保存哈夫曼编码
91  void HuffmanCodes(unordered_map<char, int> freq)
92  {
```

```
93      vector<HuffmanNode *> heap;
94      int heapSize = 0;
95
96      // 将字符和对应的频率放入小根堆中
97      for (const auto &p : freq)
98      {
99          heap.push_back(new HuffmanNode{p.first, p.second, nullptr, nullptr});
100         heapSize++;
101     }
102
103     while (heapSize > 1)
104     {
105         // 提取两个最小节点并合并为一个新节点
106         HuffmanNode *left = extractMin(heap, heapSize);
107         HuffmanNode *right = extractMin(heap, heapSize);
108         HuffmanNode *top = new HuffmanNode{'$', left->frequency + right->frequency,
                left, right};
109
110         // 将新节点插入小根堆中
111         insertNode(heap, top, heapSize);
112     }
113
114     // 保存哈夫曼编码
115     saveCodes(heap[0], "");
116 }
117
118 int main()
119 {
120     // 读入数据
121     ifstream file("DATA/huffman.txt", ios::in);
122     if (!file.is_open())
123     {
124         cerr << "Failed to open file." << endl;
125         return 1;
126     }
127     unordered_map<char, int> freq; // 用于统计字符频率
128
129     char c;
130     while (file.get(c))
131     {
132         if (isalnum(c) || isspace(c) || ispunct(c) || iscntrl(c))
133         {
134             freq[c]++;
135         }
136     }
137
138     HuffmanCodes(freq);
```

```
139
140     // 打印字符和对应的哈夫曼编码
141     cout << "========================Huffman Codes=======================\n";
142     int cnt = 0;
143     for (const auto &p : codes)
144     {
145         string s(1, p.first);
146         if (s == "\n")
147             s = "\\n";
148         cout << setw(20) << left << "<" + s + " " + to_string(p.second.first) + ">
                " + p.second.second;
149         if (++cnt % 3 == 0)
150             cout << endl;
151     }
152     cout << endl
153          << endl;
154
155     // 与等长编码比较
156     cout << "====================Compare==================\n";
157     int sum = 0;
158     for (const auto &p : codes)
159         sum += p.second.first * p.second.second.size();
160     int sum2 = 0;
161     for (const auto &p : freq)
162         sum2 += p.second;
163     sum2 *= ceil(log2(freq.size()));
164     cout << "The length of Huffman codes is " << sum << "bits." << endl;
165     cout << "The length of equal-length codes is " << sum2 << "bits." << endl;
166     cout << "Saved " << double(sum2 - sum) / sum2 * 100 << "% space." << endl;
167
168     return 0;
169 }
```

B dijkstra.cpp

```
1  /**
2   * @file dijkstra.cpp
3   * @author zhang ziliang (ziliangzhang@bupt.edu.cn)
4   * @brief dijkstra算法
5   * @date 2023-12-09
6   */
7  #include <iostream>
8  #include <iomanip>
9  #include <queue>
10 #include <unordered_map>
11 #include <fstream>
```

```cpp
#include <sstream>
#include <string>
#include <cstring>
#include <cmath>

using namespace std;

typedef pair<double, int> PII; // distance , index
const int N = 100;
const int M = 10000;

int h[N], e[M], ne[M], idx;            // 邻接表存储所有边
double w[M];                           // 存储所有边的权重
double dist[N];                        // 存储所有点到源点的距离
bool st[N];                            // 存储每个点的最短距离是否已经确定
int n;                                 // 点的数量
unordered_map<int, int> index_id;      // 存储index和基站id的映射关系
unordered_map<int, int> id_index;      // 存储基站id和index的映射关系
unordered_map<int, vector<int>> path; // 存储最短路中的路径

// 添加一条边
void add(int a, int b, double c)
{
    e[idx] = b, ne[idx] = h[a], w[idx] = c, h[a] = idx++;
}

// dijkstra算法
void dijkstra(int src)
{
    for (int i = 1; i <= n; i++)
        dist[i] = 0x3f3f3f3f; // 初始化距离为无穷大
    memset(st, 0, sizeof st);
    dist[src] = 0;
    priority_queue<PII, vector<PII>, greater<PII>> heap; // 使用优先队列优化
    heap.push({0, src});
    path[index_id[src]] = vector<int>(1, index_id[src]); // 初始化路径

    while (!heap.empty())
    {
        auto t = heap.top();
        heap.pop();

        int ver = t.second;
        double distance = t.first;
        if (!st[ver])
        {
            st[ver] = true;
```

```
            for (int i = h[ver]; i != -1; i = ne[i])
            {
                int j = e[i];
                if (dist[j] > distance + w[i])
                {
                    dist[j] = distance + w[i];
                    heap.push({dist[j], j});
                    // 更新路径
                    path[index_id[j]] = path[index_id[ver]];
                    path[index_id[j]].push_back(index_id[j]);
                }
            }
        }
    }
}

// 打印结果
void print_result(int src, int dst)
{
    cout << setw(16) << left << "<" + to_string(src) + "-" + to_string(dst) + "> "
         << "distance: ";
    if (fabs(dist[id_index[dst]] - 0x3f3f3f3f) < 1e-6)
        cout << "INF path: NULL" << endl;
    else
    {
        cout << fixed << setprecision(2) << setw(8) << left << dist[id_index[dst]]
            << "path: ";
        for (int j = 0; j < path[dst].size() - 1; j++)
            cout << path[dst][j] << "->";
        cout << path[dst][path[dst].size() - 1] << endl;
    }
}

int main()
{
    // 读入22个基站的数据
    memset(h, -1, sizeof h);
    idx = 0;
    n = 22;
    ifstream file("DATA/dijkstra_22.txt", ios::in);
    if (!file.is_open())
    {
        cerr << "Failed to open file." << endl;
        return 1;
    }
    string line;
    while (getline(file, line))
```

```
105     {
106         stringstream ss(line);
107         int a, b, a_id, b_id;
108         double c;
109         string _a, _b, _a_id, _b_id, _c;
110         ss >> _a >> _b >> _a_id >> _b_id >> _c;
111         a = stoi(_a), b = stoi(_b), a_id = stoi(_a_id), b_id = stoi(_b_id), c =
                stod(_c);
112         index_id[a] = a_id;
113         id_index[a_id] = a;
114         index_id[b] = b_id;
115         id_index[b_id] = b;
116         if (c > 0)
117             add(a, b, c);
118     }
119
120     // 以567443为源点，进行dijkstra算法
121     dijkstra(id_index[567443]);
122     cout << "
            ==========================================22==========================================\
            n";
123     for (int i = 1; i <= n; i++)
124         print_result(567443, index_id[i]);
125     cout << endl;
126     print_result(567443, 33109);
127     cout << endl;
128
129     // 读入42个基站的数据
130     memset(h, -1, sizeof h);
131     idx = 0;
132     n = 42;
133     ifstream file2("DATA/dijkstra_42.txt", ios::in);
134     if (!file2.is_open())
135     {
136         cerr << "Failed to open file." << endl;
137         return 1;
138     }
139     while (getline(file2, line))
140     {
141         stringstream ss(line);
142         int a, b, a_id, b_id;
143         double c;
144         string _a, _b, _a_id, _b_id, _c;
145         ss >> _a >> _b >> _a_id >> _b_id >> _c;
146         a = stoi(_a), b = stoi(_b), a_id = stoi(_a_id), b_id = stoi(_b_id), c =
                stod(_c);
147         index_id[a] = a_id;
```

```
148         id_index[a_id] = a;
149         index_id[b] = b_id;
150         id_index[b_id] = b;
151         if (c > 0)
152             add(a, b, c);
153     }
154 
155     // 以565845为源点，进行dijkstra算法
156     dijkstra(id_index[565845]);
157     cout << "
        ==========================================42==========================================\
        n";
158     for (int i = 1; i <= n; i++)
159         print_result(565845, index_id[i]);
160     cout << endl;
161     print_result(565845, 565667);
162 
163     return 0;
164 }
```

C prim.cpp

```
1  /**
2   * @file prim.cpp
3   * @author zhang ziliang (ziliangzhang@bupt.edu.cn)
4   * @brief prim算法
5   * @date 2023-12-10
6   */
7  #include <iostream>
8  #include <iomanip>
9  #include <fstream>
10 #include <sstream>
11 #include <string>
12 #include <cstring>
13 #include <cmath>
14 
15 using namespace std;
16 
17 const int N = 100;
18 const double INF = 0x3f3f3f3f;
19 
20 double g[N][N], dist[N]; // 邻接矩阵存储所有边
21 bool st[N];              // 判断点是否已经加入最小生成树
22 int mst[N];              // 存储最小生成树
23 int n;                   // 点的数量
24 
```

```
25  double prim()
26  {
27      memset(mst, -1, sizeof mst);
28      memset(st, 0, sizeof st);
29      for (int i = 1; i <= n; i++)
30          dist[i] = INF;
31      double res = 0;
32
33      for (int i = 0; i < n; i++)
34      {
35          int t = -1;
36          for (int j = 1; j <= n; j++)
37              if (!st[j] && (t == -1 || dist[t] > dist[j]))
38                  t = j;
39
40          // 图不连通
41          if (i && fabs(dist[t] - INF) < 1e-6)
42              return INF;
43
44          // 加上树边权重
45          if (i)
46              res += dist[t];
47
48          st[t] = true;
49
50          // 更新其他点到最小生成树的距离
51          for (int j = 1; j <= n; j++)
52          {
53              if (dist[j] > g[t][j] && !st[j])
54              {
55                  dist[j] = g[t][j];
56                  mst[j] = t;
57              }
58          }
59      }
60
61      return res;
62  }
63
64  int main()
65  {
66      // 读入22个基站的数据
67      n = 22;
68      ifstream file("DATA/mst_22.txt", ios::in);
69      if (!file.is_open())
70      {
71          cerr << "Failed to open file." << endl;
```

```
72          return 1;
73      }
74      string line;
75      while (getline(file, line))
76      {
77          stringstream ss(line);
78          int a, b, a_id, b_id;
79          double c;
80          string _a, _b, _a_id, _b_id, _c;
81          ss >> _a >> _b >> _a_id >> _b_id >> _c;
82          a = stoi(_a), b = stoi(_b), a_id = stoi(_a_id), b_id = stoi(_b_id), c =
                stod(_c);
83          if (c > 0)
84              g[a][b] = c;
85          else
86              g[a][b] = INF;
87      }
88
89      double res = prim();
90      cout << "==========22==========\n";
91      if (fabs(res - INF) < 1e-6)
92          puts("Impossible!");
93      else
94      {
95          cout << "Cost: " << fixed << setprecision(2) << res << endl;
96          int cnt = 0;
97          for (int i = 2; i <= n; i++)
98          {
99              cout << setw(6) << left << to_string(i) + "-" + to_string(mst[i]);
100             cnt++;
101             if (cnt % 4 == 0)
102                 cout << endl;
103         }
104     }
105     cout << endl
106          << endl;
107
108     // 将最小生成树的结果写入文件
109     ofstream file2("RESULT/prim_22_res.txt", ios::out);
110     if (!file2.is_open())
111     {
112         cerr << "Failed to open file." << endl;
113         return 1;
114     }
115     for (int i = 2; i <= n; i++)
116         file2 << mst[i] << "\n";
117
```

```
118     // 读入42个基站的数据
119     n = 42;
120     ifstream file3("DATA/mst_42.txt", ios::in);
121     if (!file3.is_open())
122     {
123         cerr << "Failed to open file." << endl;
124         return 1;
125     }
126     while (getline(file3, line))
127     {
128         stringstream ss(line);
129         int a, b, a_id, b_id;
130         double c;
131         string _a, _b, _a_id, _b_id, _c;
132         ss >> _a >> _b >> _a_id >> _b_id >> _c;
133         a = stoi(_a), b = stoi(_b), a_id = stoi(_a_id), b_id = stoi(_b_id), c =
                stod(_c);
134         if (c > 0)
135             g[a][b] = c;
136         else
137             g[a][b] = INF;
138     }
139
140     res = prim();
141     cout << "==========42==========\n";
142     if (fabs(res - INF) < 1e-6)
143         puts("Impossible!");
144     else
145     {
146         cout << "Cost: " << fixed << setprecision(2) << res << endl;
147         int cnt = 0;
148         for (int i = 2; i <= n; i++)
149         {
150             cout << setw(6) << left << to_string(i) + "-" + to_string(mst[i]);
151             cnt++;
152             if (cnt % 4 == 0)
153                 cout << endl;
154         }
155     }
156     cout << endl;
157
158     // 将最小生成树的结果写入文件
159     ofstream file4("RESULT/prim_42_res.txt", ios::out);
160     if (!file4.is_open())
161     {
162         cerr << "Failed to open file." << endl;
163         return 1;
```

```
    }
    for (int i = 2; i <= n; i++)
        file4 << mst[i] << "\n";

    return 0;
}
```

D kruskal.cpp

```
/**
 * @file kruskal.cpp
 * @author zhang ziliang (ziliangzhang@bupt.edu.cn)
 * @brief kruskal算法
 * @date 2023-12-10
 */
#include <iostream>
#include <iomanip>
#include <fstream>
#include <sstream>
#include <string>
#include <cstring>
#include <cmath>
#include <algorithm>

using namespace std;

const int N = 100;
const int M = 10000;
const double INF = 0x3f3f3f3f;

typedef pair<int, int> PII;

struct Edge
{
    int a, b;
    double w;

    bool operator<(const Edge &W) const
    {
        return w < W.w;
    }
} edges[M];

int p[N];    // 并查集
int n, m;    // 点的数量，边的数量
PII mst[N]; // 存储最小生成树
```

```
38
39  int find(int x)
40  {
41      if (p[x] != x)
42          p[x] = find(p[x]);
43      return p[x];
44  }
45
46  double kruskal()
47  {
48      sort(edges, edges + m);
49      double res = 0;
50      int cnt = 0;
51      for (int i = 1; i <= n; i++)
52          p[i] = i; // 初始化并查集
53
54      for (int i = 0; i < m; i++)
55      {
56          int a = edges[i].a, b = edges[i].b;
57          double w = edges[i].w;
58
59          int p_a = find(a), p_b = find(b);
60          // 判断是否已经在一个连通图中
61          if (p_a != p_b)
62          {
63              p[p_a] = p_b;
64              res += w;
65              mst[cnt] = {a, b};
66              cnt++;
67          }
68
69          if (cnt == n - 1)
70              return res;
71      }
72
73      return INF;
74  }
75
76  int main()
77  {
78      // 读入22个基站的数据
79      n = 22;
80      m = 0;
81      ifstream file("DATA/mst_22.txt", ios::in);
82      if (!file.is_open())
83      {
84          cerr << "Failed to open file." << endl;
```

```
85          return 1;
86      }
87      string line;
88      while (getline(file, line))
89      {
90          stringstream ss(line);
91          int a, b, a_id, b_id;
92          double c;
93          string _a, _b, _a_id, _b_id, _c;
94          ss >> _a >> _b >> _a_id >> _b_id >> _c;
95          a = stoi(_a), b = stoi(_b), a_id = stoi(_a_id), b_id = stoi(_b_id), c =
                stod(_c);
96          if (c > 0)
97              edges[m++] = {a, b, c};
98      }
99
100     double res = kruskal();
101     cout << "==========22==========\n";
102     if (fabs(res - INF) < 1e-6)
103         puts("Impossible!");
104     else
105     {
106         cout << "Cost: " << fixed << setprecision(2) << res << endl;
107         int cnt = 0;
108         for (int i = 0; i < n - 1; i++)
109         {
110             cout << setw(6) << left << to_string(mst[i].first) + "-" + to_string(
                    mst[i].second);
111             if (++cnt % 4 == 0)
112                 cout << endl;
113         }
114     }
115     cout << endl
116          << endl;
117
118     // 将最小生成树的结果写入文件
119     ofstream file2("RESULT/kruskal_22_res.txt", ios::out);
120     if (!file2.is_open())
121     {
122         cerr << "Failed to open file." << endl;
123         return 1;
124     }
125     for (int i = 0; i < n - 1; i++)
126         file2 << mst[i].first << " " << mst[i].second << "\n";
127
128     // 读入42个基站的数据
129     n = 42;
```

```
130     m = 0;
131     ifstream file3("DATA/mst_42.txt", ios::in);
132     if (!file3.is_open())
133     {
134         cerr << "Failed to open file." << endl;
135         return 1;
136     }
137     while (getline(file3, line))
138     {
139         stringstream ss(line);
140         int a, b, a_id, b_id;
141         double c;
142         string _a, _b, _a_id, _b_id, _c;
143         ss >> _a >> _b >> _a_id >> _b_id >> _c;
144         a = stoi(_a), b = stoi(_b), a_id = stoi(_a_id), b_id = stoi(_b_id), c =
                stod(_c);
145         if (c > 0)
146             edges[m++] = {a, b, c};
147     }
148 
149     res = kruskal();
150     cout << "==========42==========\n";
151     if (fabs(res - INF) < 1e-6)
152         puts("Impossible!");
153     else
154     {
155         cout << "Cost: " << fixed << setprecision(2) << res << endl;
156         int cnt = 0;
157         for (int i = 0; i < n - 1; i++)
158         {
159             cout << setw(6) << left << to_string(mst[i].first) + "-" + to_string(
                    mst[i].second);
160             if (++cnt % 4 == 0)
161                 cout << endl;
162         }
163     }
164     cout << endl;
165 
166     // 将最小生成树的结果写入文件
167     ofstream file4("RESULT/kruskal_42_res.txt", ios::out);
168     if (!file4.is_open())
169     {
170         cerr << "Failed to open file." << endl;
171         return 1;
172     }
173     for (int i = 0; i < n - 1; i++)
174         file4 << mst[i].first << " " << mst[i].second << "\n";
```

```
175
176     return 0;
177 }
```