Shortest Path Algorithm with Heaps

Date: 2024-10-23

目录

Chapter 1: Introduction

Dijkstra算法是一种用于寻找图中最短路径的贪心算法。该算法主要用于解决带权有向图或无向图中的单源最短路径问题，即从起始节点到其他所有节点的最短路径。

本项目的目标是实现 Dijkstra 算法以解决最短路径问题，并比较不同数据结构（包括斐波那契堆和其他堆）在该算法中的性能表现，并找到最适合该算法的数据结构类型。

整体流程主要包括：

1. 实现算法
2. 使用美国公路网络数据集进行评估（生成至少1000对查询以测试运行时间）
3. 将评估结果以图表形式呈现
4. 分析结果

Chapter 2: Data Structure / Algorithm Specification

我们分别使用最小堆、左式堆和斐波那契堆作为数据结构来实现算法：

2.1 所用堆的模板类实现

2.1.1 最小堆

- **上浮操作：**

```
FUNCTION heapify( index )
    WHILE index > 0 // 当前节点不是根节点时继续上浮
        parent_index = (index-1)/2 //计算父节点的索引（基于0-based)
        IF heap[index].second < heap[parent_index].second THEN //此处`second`在具体实现中指的就是`dist`优先级
            SWAP heap[index] WITH heap[parent_index] //交换节点
            index = parent_index //更新索引
        ELSE //当前节点满足最小堆的性质，停止上浮
            BREAK
        END IF
    END WHILE
END FUNCTION
```

- **下滤操作：**

```
FUNCTION heapifyDown(index)
        WHILE TRUE
            leftChild = 2 * index + 1  // 左孩子索引
            rightChild = 2 * index + 2  // 右孩子索引
            smallest = index  // 假设当前节点是最小的
```

```
            // 检查左孩子
            IF leftChild < SIZE OF heap AND leftChild's priority < smallest's
priority THEN
                smallest = leftChild  // 更新最小节点为左孩子
            END IF

            // 检查右孩子
            IF rightChild < SIZE OF heap AND rightChild's priority < smallest's
priority THEN
                smallest = rightChild  // 更新最小节点为右孩子
            END IF

            // 如果最小节点不是当前节点，则交换
            IF smallest != index THEN
                SWAP current node WITH smallest  // 交换
                index = smallest  // 更新当前索引为最小节点索引
            ELSE
                BREAK  // 已满足堆性质，停止调整
            END IF
        END WHILE
    END FUNCTION

PUBLIC
    FUNCTION MinHeap()  // 构造函数
        // 初始化空堆
    END FUNCTION

    FUNCTION ~MinHeap()  // 析构函数
        // 无需特殊清理
    END FUNCTION
```

- **插入新元素**

```
 FUNCTION insert(value)
        APPEND value TO heap  // 添加到堆末尾
        CALL heapifyUp(SIZE OF heap - 1)  // 调整堆
    END FUNCTION
```

- **删除并返回最小元素**

```
    FUNCTION deleteMin() RETURNS pair<int, int>
        IF heap IS EMPTY THEN
            THROW "Heap is empty"  // 如果heap已经为空，抛出异常
        END IF

        minValue = heap[0]  // 保存堆顶元素
        heap[0] = last element  // 将最后一个元素移到堆顶
        REMOVE LAST ELEMENT FROM heap  // 移除最后元素
        IF heap IS NOT EMPTY THEN
            CALL heapifyDown(0)  // 通过下滤调整堆，将最后一个元素调整到合适为止
        END IF
        RETURN minValue  // 返回最小值
    END FUNCTION
```

- **获取堆顶最小元素，但是不删除**

```
FUNCTION getMin() RETURNS pair<int, int>
        IF heap IS EMPTY THEN
            THROW "Heap is empty"  // 如果heap为空，不存在最小元素，抛出异常
        END IF
        RETURN heap[0]  // 返回堆顶元素
    END FUNCTION
```

2.1.2 左式堆

- **节点结构体**

```
STRUCT Node
    T key  // 节点的值，可以是 std::pair<int, int>
    int npl  // null path length
    Node* left  // 左子节点
    Node* right  // 右子节点

    // 节点构造函数
    FUNCTION Node(T value)
        key = value
        npl = 0
        left = NULL
        right = NULL
    END FUNCTION
END STRUCT
```

- **合并左式堆**

```
FUNCTION merge(Node* h1, Node* h2) RETURNS Node*
        IF h1 IS NULL THEN RETURN h2  // 如果 h1 为空，返回 h2
        IF h2 IS NULL THEN RETURN h1  // 如果 h2 为空，返回 h1

        // 确保 h1 的键小于 h2 的键
        IF h1->key.second > h2->key.second THEN
```

```
            SWAP h1 WITH h2  // 交换
        END IF

        // 递归合并 h1 的右子树和 h2
        h1->right = merge(h1->right, h2)

        // 确保左子树的 NPL（null path length）大于等于右子树
        IF h1->left IS NULL OR (h1->right IS NOT NULL AND h1->left->npl < h1-
>right->npl) THEN
            SWAP h1->left WITH h1->right  // 交换左右子树
        END IF

        // 更新 h1 的 NPL
        h1->npl = (h1->right IS NOT NULL ? h1->right->npl + 1 : 0)
        RETURN h1  // 返回合并后的堆
    END FUNCTION
```

- **插入新的元素**

```
 FUNCTION insert(T value)
        Node* newNode = NEW Node(value)  // 创建新节点
        root = merge(root, newNode)  // 合并新节点与当前堆
END FUNCTION
```

- **删除并返回堆顶元素**

```
FUNCTION deleteMin() RETURNS std::pair<int, int>
    IF root IS NULL THEN
        THROW "Heap is empty"  // 抛出异常
    END IF

    minValue = root->key  // 保存最小元素
    Node* oldRoot = root  // 保存旧根节点
    root = merge(root->left, root->right)  // 合并左子树和右子树
    DELETE oldRoot  // 删除旧根节点
    RETURN minValue  // 返回最小元素
END FUNCTION
```

- **获取堆顶的最小元素**

```
FUNCTION getMin() RETURNS std::pair<int, int>
    IF root IS NULL THEN
        THROW "Heap is empty"  // 如果heap为空，不存在最小元素，抛出异常
    END IF
    RETURN root->key  // 返回最小元素
END FUNCTION
```

- **中序遍历显示堆中元素**

```
FUNCTION inorder(Node* node)  // 递归遍历节点
    IF node IS NOT NULL THEN
        inorder(node->left)  // 遍历左子树
        OUTPUT "(" + node->key.first + ", " + node->key.second + ") "  // 输出当前
节点
        inorder(node->right)  // 遍历右子树
    END IF
END FUNCTION
```

2.1.3 斐波那契堆

- **节点结构体**

```
STRUCT Node
    T key          // 存储的值（键）
    int val        // 顶点的值
    int degree     // 子节点的数量
    Node* parent   // 父节点
    Node* child    // 子节点
    Node* left     // 左兄弟
    Node* right    // 右兄弟
    bool mark      // 标记节点是否被切割

    // 节点构造函数
    FUNCTION Node(T k, int v)
        key = k
        val = v
        degree = 0
        //初始化parent 与 child都为空;
        //左右子节点指向自己，形成双向循环链表
        parent = NULL
        child = NULL
        left = this
        right = this
        mark = false
    END FUNCTION
END STRUCT
```

- **插入新元素**

```
FUNCTION insert(T value, int vertex)
    Node* node = NEW Node(value, vertex)
    //如果堆为空，将该节点设置为最小节点
    IF minNode IS NULL THEN
        minNode = node
    ELSE
        // 将新节点加入到根列表中
        node->left = minNode
        node->right = minNode->right
        minNode->right->left = node
        minNode->right = node
```

```
        IF value < minNode->key THEN
            minNode = node  // 更新最小节点
        END IF
    END IF
    heapSize++ //堆中节点个数自增
END FUNCTION
```

- **删除最小节点**

```
FUNCTION deleteMin()
    IF minNode IS NULL THEN RETURN  // 如果堆为空，返回

    Node* oldMin = minNode
    IF oldMin->child IS NOT NULL THEN
        // 将子节点加入根列表
        Node* child = oldMin->child
        DO
            child->parent = NULL
            child = child->right
        WHILE child != oldMin->child

        // 连接子节点与根列表
        CONNECT child NODES
    END IF
    // 从根列表中移除最小节点
    REMOVE oldMin FROM ROOT LIST
    IF minNode IS minNode->right THEN
        minNode = NULL
    ELSE
        minNode = minNode->right
        consolidate()  // 合并根列表
    END IF
    DELETE oldMin
    heapSize--
END FUNCTION
```

- **减小节点的键值**

```
 FUNCTION decreaseKey(Node* node, T newValue)
    //检查新的键值是否满足输入要求
    IF newValue > node->key THEN
        THROW "New value is greater than current value." //异常则抛出警告
    END IF

    node->key = newValue
    Node* parent = node->parent

    IF parent IS NOT NULL AND node->key < parent->key THEN
        cut(node, parent)  // 切割
        cascadingCut(parent)  // 级联切割
    END IF

    IF minNode IS NOT NULL AND node->key < minNode->key THEN
        minNode = node  // 更新最小节点
```

```
    END IF
END FUNCTION
```

- **将两个节点连接**

```
FUNCTION link(Node* y, Node* x)
    REMOVE y FROM ROOT LIST
    // 将 y 连接到 x 的子节点
    y->parent = x
    IF x->child IS NULL THEN
        x->child = y
        y->left = y->right = y
    ELSE
        // 将 y 插入到 x 的子列表中
        INSERT y INTO CHILD LIST OF x
    END IF
    x->degree++  // 更新 x 的度数
    y->mark = false  // 重置标记
END FUNCTION
```

- **合并根列表**

```
FUNCTION consolidate()
    int D = LOG2(heapSize) + 1  // 计算最大度数
    CREATE ARRAY A[D]  // 初始化根列表

    // 遍历根节点，合并同样度数的节点
    FOR EACH node IN rootList DO
        Node* x = node
        int d = x->degree
        WHILE A[d] IS NOT NULL DO
            Node* y = A[d]
            IF x->key > y->key THEN
                SWAP x WITH y  // 确保 x 是最小的
            END IF
            link(y, x)  // 连接
            A[d] = NULL  // 清空
            d++
        END WHILE
        A[d] = x  // 将 x 放入数组
    END FOR
    // 重新建立最小节点
    minNode = NULL
    FOR EACH node IN A DO
        IF node IS NOT NULL THEN
            ADD node TO ROOT LIST
            IF minNode IS NULL OR node->key < minNode->key THEN
                minNode = node  // 更新最小节点
            END IF
        END IF
    END FOR
END FUNCTION
```

- **切割节点**

```
FUNCTION cut(Node* x, Node* y)
    // 从父节点的子列表中移除 x
    REMOVE x FROM y->child
    y->degree--

    // 将 x 添加到根列表
    ADD x TO ROOT LIST
    x->parent = NULL
    x->mark = false
END FUNCTION
```

- **级联切割**

```
FUNCTION cascadingCut(Node* y)
    Node* z = y->parent
    IF z IS NOT NULL THEN
        IF NOT y->mark THEN
            y->mark = true  // 标记mark，表示节点被切割
        ELSE
            cut(y, z)  // 进行切割
            cascadingCut(z)  // 对父节点递归进行级联切割
        END IF
    END IF
END FUNCTION
```

- **删除指定节点**

```
FUNCTION deleteNode(Node* node)
    decreaseKey(node, MIN_VALUE)  // 将节点值减至最小
    deleteMin()  // 删除最小节点
END FUNCTION
```

2.2 主函数分析

2.2.1 整体流程：

1. 初始化和设置：

- 定义常量（如 `DEMO`）。
- 创建一个包含图文件名的数组。
- 设置线程数量以优化文件读取。

2. 循环读取文件：

> 对于每个图文件：

- 输出当前读取的文件名。
- 开始计时，创建图对象并加载数据。
- 停止计时，输出读取成功的信息。

3. **性能测试**：

- 对加载的图执行多个性能测试，使用不同的堆实现（如包括斐波那契堆、左式堆和二叉堆）。

2.2.2 多线程读取文件

在读取大规模的图信息时，我们采取多线程读取的方式来加快文件读取的处理速度：

```
void Graph::loadGraphFromFileMultithreaded(const std::string& filename, int
numThreads) {
    std::ifstream infile(filename, std::ios::ate);  // 打开文件并定位到文件末尾
    if (!infile.is_open()) {
        throw std::runtime_error("文件打开错误:  " + filename);
    }
    // 获取文件大小
    std::streampos fileSize = infile.tellg();
    infile.seekg(0, std::ios::beg);  // 重置到文件开头

    // 读取文件头部信息，找到顶点和边的数量
    std::string line;
    while (std::getline(infile, line)) {
        if (line[0] == 'p') {
            std::istringstream iss(line);
            std::string type;
            iss >> type >> type >> numVertices >> numEdges; //总的图信息，在p一行
            break;
        }
    }

    // 初始化邻接列表
    adjList.resize(numVertices, nullptr);

    // 找到每个线程应该开始的位置，按行划分
    std::vector<std::streampos> threadStarts;
    threadStarts.push_back(infile.tellg());  // 第一个线程从当前文件位置开始

    std::streamoff partSize = static_cast<std::streamoff>(fileSize) / numThreads;
    for (int i = 1; i < numThreads; ++i) {
        infile.seekg(i * partSize);  // 定位到每个部分的大致开始位置（按字节划分）
        // 跳到下一行，避免截断
        std::string dummyLine;
        std::getline(infile, dummyLine);
        threadStarts.push_back(infile.tellg());  // 每个线程的起始点
    }

    // 最后一个线程处理到文件末尾
    threadStarts.push_back(fileSize);

    // 创建线程并分配工作
    std::vector<std::thread> threads;
    for (int i = 0; i < numThreads; ++i) { //线程创建
        std::streampos start = threadStarts[i];
        std::streampos end = threadStarts[i + 1];
```

```
        threads.emplace_back(&Graph::processFilePart, this, filename, start, end,
i);
        //把创建好的工作放入线程池
    }

    // 等待所有线程完成
    for (auto& th : threads) {
        th.join(); //挂起
    }

    infile.close();//完毕
}
```

1. 首先打开文件并定位到文件末尾，如果失败则抛出异常信号
2. 通过`infile.tellg();`获取文件末尾的位置，然后重置到文件开头
3. 读取文件头部信息，找到顶点和边的数量进而初始化图（邻接链表的形式）
4. `std::vector<std::streampos> threadStarts;`存储各线程的起始位置，并通过`static_cast<std::streamoff>(fileSize) / numThreads;`计算各个线程需要处理的文件大小；将先前存储的文件末尾位置作为最后一个线程的结束位置
5. 调用`processFilePart`执行各个线程的读取工作

- **`processFilePart`单线程的读取工作**

```
void Graph::processFilePart(const std::string& filename, std::streampos start,
std::streampos end, int tqdm) {
    //tqdm参数请无视它，这里本来有进度条实现的，但是在WSL和VS里结果不一样，所以废弃了，保留这个
参数为了保留这个功能的实现空间
    std::ifstream infile(filename);//文件打开为流
    if (!infile.is_open()) {
        throw std::runtime_error("文件打开错误：" + filename);
    }//抛出错误信息 在main被捕获

    infile.seekg(start);  // 从文件的指定位置开始读取，这是多线程的实现函数
    std::string line; //一行一行的开始读取
    while (infile.tellg() < end && std::getline(infile, line)) {
        if (line[0] == 'a') {
            int src, dest, weight;//a的处理
            std::istringstream iss(line);
            char a;
            iss >> a >> src >> dest >> weight; // 输入的a就是开头字符，但是没用

            src -= 1;   // 转为从0开始的索引 因为数据里面的节点是：12345678
            dest -= 1;  // 我们想要把它变成                          01234567

            // 添加边，使用互斥锁保证线程安全
            {
                std::lock_guard<std::mutex> lock(graphMutex);  //互斥锁，如果两个线
程同时要操作链表，互斥锁可以保证不产生冲突，依序操作
                AdjListNode* newNode = new AdjListNode(dest, weight); //新建节点
                newNode->next = adjList[src];
                adjList[src] = newNode;
```

```
                // 如果是无向图，添加反向边
                /*
                newNode = new AdjListNode(src, weight);
                newNode->next = adjList[dest];
                adjList[dest] = newNode;
                */
            }
        }
    }
    infile.close(); //文件关闭
}
```

> 通过`loadGraphFromFileMultithreaded`函数给出的文件位置，定位到该线程需要处理的文件位置，然后读取文件内容来构建图的部分。

互斥锁

- 由于数据集被分割成尽可能均匀的部分，并被多线程处理，各线程共享了对同一个邻接表的访问权，因此可能存在数据竞争问题：如果多个线程同时访问和修改`adjList`而没有同步机制，就可能导致数据损坏或程序崩溃。为了解决该问题，引入互斥锁。

```
`std::lock_guard<std::mutex>`
```

> 通过自动管理锁的生命周期。当控制流进入这个代码块时，锁被获取；当控制流离开时，无论是正常退出还是由于异常退出，锁都会自动释放。

Chapter 3: Testing Results

3.1 测试目的及结果

- 测试目的：利用`USA road networks`的数据集测试，以比较不同堆的性能

> 数据集规模对应如下:

Name	Desc	Nodes	Arcs
USA	Full USA	23947347	58333344
CTR	Central USA	14081816	34292496
W	Western USA	6262104	15248146
E	Eastern USA	3598623	8778114
LKS	Great Lakes	2758119	6885658
CAL	California and Nevada	1890815	4657742
NE	Northeast USA	1524453	3897636
NW	Northwest USA	1207945	2840208
FLA	Florida	1070376	2712798
COL	Colorado	435666	1057066
BAY	San Francisco Bay Area	321270	800172
NY	New York City	264346	733846

> 数据集的来源：http://www.dis.uniroma1.it/challenge9/download.shtml

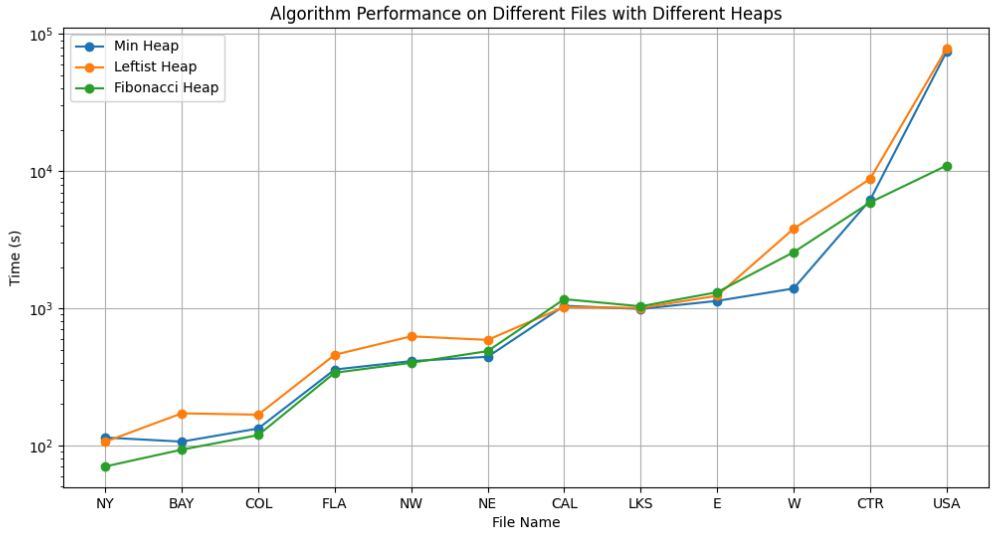
- `run time table`

数据集	最小堆	左式堆	斐波那契堆
USA	74080	78282.5	10975
CTR	6152.9	8763.20	5912.8
W	1394.33	3804.37	2563.87
E	1132.47	1234.7	1309.6
LKS	989.32	1005.2	1034.1
CAL	1043.43	1023.21	1166.07
NE	443.21	588.43	487.47
NW	412.33	624.9	400.7

数据集	最小堆	左式堆	斐波那契堆
FLA	357.88	458.6	339.17
COL	133.06	167.76	119.1
BAY	106.73	171.61	93.29
NY	114.33	106.57	70.37

> 分别为执行`1000`次问询对所需的时间（单位为s）

- `Plot`



> 1. 上图的横坐标向右，是按照输入规模递增的顺序；纵坐标是运行各执行1000次问询对之后的时间(取对数)
> 2. 在输入规模较小时，三种堆实现的算法在时间上没有较大的差异；

3.2 结果分析

> 1. 当图文件的规模达到一定程度时，斐波那契堆实现的算法，其所需的时间明显少于最小堆和左式堆所需的时间。经过复杂度分析可知，这得益于斐波那契堆的`decreaseKey`函数在均摊时间复杂度下的O(1)成本，这为`Dijkstra`算法需要频繁更新节点`key`的情景节约了明显更多的时间成本；
> 2. 综合上述分析，在图的规模较小时，考虑堆实现的难易程度与实际时间成本，建议优先选取**最小堆**作为数据结构；当图的规模较大时，时间成本成为主要的考虑因素，优先选取**斐波那契堆**来实现。

Chapter 4: Analysis and Comments

4.1 时间复杂度分析

4.1.1 读入数据集的时间复杂度

- **读取文件头　O(1)**

```
while (std::getline(infile, line)) {
    if (line[0] == 'p') {
        std::istringstream iss(line);
        std::string type;
        iss >> type >> type >> numVertices >> numEdges;
        break;
    }
}
```

读取文件头来解析顶点数和边数，时间复杂度为O(L), 其中L为文件头之前的行数。这部分较短，因此可认为是常数级O(1)。

- **初始化邻接链表　O(V)**

```
adjList.resize(numVertices, nullptr);
```

初始化图的邻接表，`resize`需要为每个顶点设置初始值`nullptr`，时间复杂度为O(V), 其中V为顶点数。

- **多线程处理文件**:

1. **分割文件 O(T)**

```
std::streamoff partSize = static_cast<std::streamoff>(fileSize) / numThreads;
for (int i = 1; i < numThreads; ++i) {
    infile.seekg(i * partSize);
    std::string dummyLine;
    std::getline(infile, dummyLine);
    threadStarts.push_back(infile.tellg());
}
```

> `partSize`计算各个线程读取的文件大小，进而计算各个线程开始的位置并存入`threadStarts`。
> 因此该`for`耗时O(T),其中T为线程个数，为常数级；

2. **多线程处理　O(E/T)**
3. 为了防止数据竞争而存在的互斥锁，在一定程度上会影响读取速度，但是相比于O(E)以及多线程处理的影响较小。

综上：总体图的读取和构建总体时间复杂度为 **O(V+E/T)**。

4.1.2 算法处理的时间复杂度

- **最小堆**

  1. **初始化** 利用数组`distances`初始化所有的顶点`dist`为无穷大，并根据传入的`src`设置对应的初始`dist`为`0`；因此V个顶点入堆，且其中`V-1`个顶点插入到数组尾部后**不需要**执行上浮操作，该部分时间复杂度为O(V-1)；插入`src`所对应的节点时需要执行`heapify`操作，耗时O(`logV`)。因此，这部分的时间复杂度整体为O(`logV + V-1`)，约为**O(V)**。
  2. **主循环** 提取最小点并更新其邻接顶点的`dist`部分，考虑上界：提取所有的顶点并调整堆维持最小堆的性质耗时O(`VlogV`),每条边最多涉及一次临界点的`dist`更新，并伴随该更新`dist`的重新插入，单次耗时O(`logV`)，因此整体的耗时为O(`ElogV`)。

  综上分析，最小堆实现算法的总体时间复杂度为O(`V+(E+V)logV`), 近似为**O(`(E+V)logV`)**

- **左式堆**

  1. **初始化** 将所有的顶点插入左式堆耗时O(`VlogV`);
  2. 主循环 与最小堆类似，单次合并的时间复杂度为O(`logV`)，E条边最多涉及E次`dist`的更新以及合并；

  综上分析，整体近似为**O(`(E+V)logV`)**。

- **斐波那契堆**

1. **初始化** 由于在插入时，各个节点初始插入的位置都是根据指针`min`直接插入到根链表中，因此单次插入的时间复杂度为O(1)。初始化时连续插入V个节点的时间复杂度为O(V).
2. **主循环** 调用`deleteMin`和`consolidate`提取最小距离顶点，涉及根链表的连接以及相同度的树的合并，其均摊时间复杂度为O(`logV`), 至多设计V个顶点的操作，因此整体为O(`VlogV`)；更新邻接顶点的`dist`时，由于斐波那契堆的`decreaseKey`的均摊时间复杂度为O(1), 每条边至多导致一次的`dist`更新，因此该部分的时间复杂度为O(E)。

   综上分析，斐波那契堆的总体时间复杂度为O(`V+VlogV+E`)，近似为**O(`VlogV+E`)**

结合4.1.1和4.1.2分析，上述实现的整体时间复杂度为：

- 最小堆和左式堆版本：O(`VlogV+ElogV`)；
- 斐波那契堆版本：O(`VlogV+E`).

4.2 空间复杂度分析

4.2.1 读入数据集的空间复杂度

1. **邻接表存储** 图的表示需要**O(V+E)**的空间复杂度；
2. **构造多线程读取**

需要根据线程数计算各线程的文件起始位置，存储到辅助数组当中：

```
std::vector<std::streampos> threadStarts
```

因此，该部分的空间复杂度为**O(T)**，其中T为线程数。

4.2.2 算法处理的空间复杂度

三种堆结构均需要存储V个顶点、大小为V的距离数组`distance`以及前驱数组，整体为O(`V`).

> 其中，斐波那契堆需要开设一个数组来存储合并过程中临时的树根，大小为O(`logV`)。

综合4.2.1和4.2.2分析，整体的空间复杂度为**O(`E+V`).**

4.3 程序总体评价

4.3.1 读取文件方面

在读取文件方面，考虑到此次输入规模较大，我们采取多线程的方式读取图的信息。

同时，因为所有线程同时共享堆邻接表的访问，我们采取使用互斥锁的方式确保数据安全的存储，避免多线程之间的数据访问冲突。

最终，我们将读取文件所需的时间O(E)缩减为O(E/T),其中T为线程数。较为显著地降低了时间开支。

4.3.2 不同堆的实现方面

我们成功分别用最小堆、左式堆以及斐波那契堆来实现算法。结合不同堆对于不同规模的图的运行表现，我们在`3.2 结果分析`中得到了不同情况下的`the best data structure for the Dijkstra's algorithm`,高度实现了本项目的要求。

4.3.3 改进计划

- **线程分配**:
  - 进一步优化文件读取的多线程分配，确保各线程的工作量均衡。
  - 使用并行算法库来简化多线程实现，提升可维护性。
- **智能指针**
  - 使用智能指针（如`std::unique_ptr`或`std::shared_ptr`）来管理动态内存，减少手动管理的复杂性，防止内存泄漏。
- **性能测试**
  - 使用基准测试工具对不同堆实现进行详细的性能比较，找出瓶颈。

Appendix

Source Code

- `main.cpp`

```
#include "../include/graph.h"

#include <iostream>
#include <time.h>
#include <array>
#include <string>
#include<vector>

/*********************************************************************************************************************
 * 这里是main函数
 * if you cannot see simplified Chinese word, please set your environment according to readme!
 * 主要采用面向对象，只展示接口，所以main函数是阅读友好的，建议从这里开始阅读
 *
 * 下面有两个超参数可以自己设置
 * threads表示读取文件时的线程，一个好的线程数目可以提高效率，经过测试5~6效率较高
 * query_number表示在进行随机查询时查询的数据对数，题目要求1000对，如果你对自己的电脑有自信（参考readme的配置），可以试着跑1000对
 * 但是就测试而言，为了避免不必要的蓝屏，发热，爆炸等情况，建议使用50对以下
 *
 * 在main函数中的测试函数设置：
 * graph.test(query_number, filename, heaptype, Vnum, israndom, start, end);
 * 这个方法有两种调用模式：
 * 1.随机生成数据开始循环跑：我们需要设置query_number表示循环次数，filename表示文件名称，heaptype表示堆结构名称，Vnum表示顶点个数，israndom设置为1
 * 2.手动输入数据检测：我们需要设置filename表示文件名称，heaptype表示堆结构名称，Vnum表示顶点个数，israndom设置为0，a，b表示起点和终点
 * 上面没说的参数就是随便设置，建议设置为空的基本类型。
 * 返回值是一个double，你也可以对它操作（我们没有操作因为在函数内部默认显示了）
 * heaptype支持三个：fib left int 分别表示斐波那契堆，左式堆和普通最小堆

*********************************************************************************************************************/

int threads = 5;
int query_number = 10;


int main() {
    try { //尝试创建图（从文件中加载数据）
        std::array<std::string, 13> all_files = {
        "test.gr",
        "USA-road-d.BAY.gr",
        "USA-road-d.CAL.gr",
        "USA-road-d.COL.gr",
        "USA-road-d.E.gr",
        "USA-road-d.FLA.gr",
```

```
        "USA-road-d.LKS.gr",
        "USA-road-d.NE.gr",
        "USA-road-d.NW.gr",
        "USA-road-d.NY.gr",
        "USA-road-d.USA.gr",
        "USA-road-d.W.gr",
        "USA-road-d.CTR.gr",
        };//所有的文件名

        std::string filename;//具体创建图时的文件名，在循环时从上面的数组中替换

        for (int i = 0; i < 13; i++) {
            filename = all_files[i];//替换

            std::cout << "正在读取文件" << filename << "..." << std::endl;//IO操作

            clock_t start_time = clock();
            Graph graph("data/"+filename, threads);//这里实例化一个图，传入文件名，这个
类将自动创建出一个图，以后的操作都在类的实例化对象上进行
            clock_t end_time = clock();
            //这是文件读取计时
            std::cout << "文件" << filename << "读取成功" << "，用时" << end_time -
start_time << std::endl;

            //显示图的邻接列表
            //graph.displayGraph();
            //很大的图就别显示了，你可以手动设置只读取test.gr然后查看它的效果

            //测试，关于这个测试函数怎么用，请见文件开头
            //下面是我们给出的一些测试例子，你可以有选择地取消他们的注释来试一试
            //这三组是随机测试循环1000轮
            double time_fib = graph.test(query_number, "data/" + filename, "fib",
graph.getNumVertices(), 1, 0, 0);
            //double time_left = graph.test(query_number, "data/" + filename,
"left", graph.getNumVertices(), 1, 0, 0);
            //double time_min = graph.test(query_number, "data/" + filename,
"min", graph.getNumVertices(), 1, 0, 0);

            //这三组是利用test.gr显示最短路径和长度
            //double t1 = graph.test(1, "data/" + "test.gr", "fib",
graph.getNumVertices(), 0, 1, 0);
            //double t2 = graph.test(1, "data/" + "test.gr", "left",
graph.getNumVertices(), 0, 1, 0);
            //double t3 = graph.test(1, "data/" + "test.gr", "min",
graph.getNumVertices(), 0, 1, 0);

        }
    }
    catch (const std::exception& e) {
        // 捕获并处理异常，输出错误信息 在文件内部会扔出各种错误并在此处捕获
        std::cerr << "错误 " << e.what() << std::endl;//我没用英语因为注释和信息用中文
写能看得更明白一点
        return 1;  // 返回错误代码
    }

    return 0;  // 程序正常结束
```

```
}
```

`graph.cpp`：

```
#include "../include/graph.h"

#include <iostream>
#include <time.h>
#include <array>
#include <string>
#include<vector>

/*****************************************************************************************
************************************
 * 这里是main函数
 * if you cannot see simplified Chinese word, please set your environment
according to readme!
 * 主要采用面向对象，只展示接口，所以main函数是阅读友好的，建议从这里开始阅读
 *
 * 下面有两个超参数可以自己设置
 * threads表示读取文件时的线程，一个好的线程数目可以提高效率，经过测试5~6效率较高
 * query_number表示在进行随机查询时查询的数据对数，题目要求1000对，如果你对自己的电脑有自信
（参考readme的配置），可以试着跑1000对
 * 但是就测试而言，为了避免不必要的蓝屏，发热，爆炸等情况，建议使用50对以下
 *
 * 在main函数中的测试函数设置：
 * graph.test(query_number, filename, heaptype, Vnum, israndom, start, end);
 * 这个方法有两种调用模式：
 * 1.随机生成数据开始循环跑：我们需要设置query_number表示循环次数，filename表示文件名称，
heaptype表示堆结构名称，Vnum表示顶点个数，israndom设置为1
 * 2.手动输入数据检测：我们需要设置filename表示文件名称，heaptype表示堆结构名称，Vnum表示顶
点个数，israndom设置为0，a，b表示起点和终点
 * 上面没说的参数就是随便设置，建议设置为空的基本类型。
 * 返回值是一个double，你也可以对它操作（我们没有操作因为在函数内部默认显示了）
 * heaptype支持三个：fib left int 分别表示斐波那契堆，左式堆和普通最小堆

*****************************************************************************************
************************************/

int threads = 5;
int query_number = 10;


int main() {
    try { //尝试创建图（从文件中加载数据）
        std::array<std::string, 13> all_files = {
        "test.gr",
        "USA-road-d.BAY.gr",
        "USA-road-d.CAL.gr",
        "USA-road-d.COL.gr",
        "USA-road-d.E.gr",
        "USA-road-d.FLA.gr",
        "USA-road-d.LKS.gr",
        "USA-road-d.NE.gr",
```

```
        "USA-road-d.NW.gr",
        "USA-road-d.NY.gr",
        "USA-road-d.USA.gr",
        "USA-road-d.W.gr",
        "USA-road-d.CTR.gr",
        };//所有的文件名

        std::string filename;//具体创建图时的文件名，在循环时从上面的数组中替换

        for (int i = 0; i < 13; i++) {
            filename = all_files[i];//替换

            std::cout << "正在读取文件" << filename << "..." << std::endl;//IO操作

            clock_t start_time = clock();
            Graph graph("data/"+filename, threads);//这里实例化一个图，传入文件名，这个
类将自动创建出一个图，以后的操作都在类的实例化对象上进行
            clock_t end_time = clock();
            //这是文件读取计时
            std::cout << "文件" << filename << "读取成功" << "，用时" << end_time -
start_time << std::endl;

            //显示图的邻接列表
            //graph.displayGraph();
            //很大的图就别显示了，你可以手动设置只读取test.gr然后查看它的效果

            //测试，关于这个测试函数怎么用，请见文件开头
            //下面是我们给出的一些测试例子，你可以有选择地取消他们的注释来试一试
            //这三组是随机测试循环1000轮
            double time_fib = graph.test(query_number, "data/" + filename, "fib",
graph.getNumVertices(), 1, 0, 0);
            //double time_left = graph.test(query_number, "data/" + filename,
"left", graph.getNumVertices(), 1, 0, 0);
            //double time_min = graph.test(query_number, "data/" + filename,
"min", graph.getNumVertices(), 1, 0, 0);

            //这三组是利用test.gr显示最短路径和长度
            //double t1 = graph.test(1, "data/" + "test.gr", "fib",
graph.getNumVertices(), 0, 1, 0);
            //double t2 = graph.test(1, "data/" + "test.gr", "left",
graph.getNumVertices(), 0, 1, 0);
            //double t3 = graph.test(1, "data/" + "test.gr", "min",
graph.getNumVertices(), 0, 1, 0);

        }
    }
    catch (const std::exception& e) {
        // 捕获并处理异常，输出错误信息 在文件内部会扔出各种错误并在此处捕获
        std::cerr << "错误 " << e.what() << std::endl;//我没用英语因为注释和信息用中文
写能看得更明白一点
        return 1;  // 返回错误代码
    }

    return 0;  // 程序正常结束
}
```

`minheap.h`:

```
#ifndef MIN_HEAP_H
#define MIN_HEAP_H

#include <iostream>
#include <limits>
#include <stdexcept>
#include <utility>
#include <vector>
#include <unordered_map>

/*********************************************************************************************
*****************************
 * 这是普通最小堆的内部实现
 * 在本次，我们编写的数据结构都是模板类 未实例化时不指定类型名称
 * 因此无法按照传统的 .h 放名称 .cpp 放主体的方式实现
 * 为了结构完整性加了cpp文件，但它们全是空的
 * 所以在实现上会有注释“用**结构作为堆元素”
 *
 * 包含操作增删查减小键值等
 *

*********************************************************************************************
**************************/

template <typename T>
class MinHeap {
private:
    std::vector<T> heap; // 用于存储堆的元素，假设每个元素是一个 std::pair<int, int>，其中第一个 int 是顶点，第二个 int 是键值
    std::unordered_map<int, int> pos; // 用来追踪每个顶点在堆中的位置，key 是顶点，value 是该顶点在 heap 数组中的索引

    // 向上调整堆，以保持最小堆的性质
    void heapifyUp(int index) {
        while (index > 0) {
            int parentIndex = (index - 1) / 2; // 计算父节点的索引
            // 如果当前节点的键值小于父节点的键值，则交换
            if (heap[index].second < heap[parentIndex].second) {
                std::swap(heap[index], heap[parentIndex]);
                // 更新 pos 数组中对应顶点的索引
                pos[heap[index].first] = index;
                pos[heap[parentIndex].first] = parentIndex;
                // 移动到父节点继续检查
                index = parentIndex;
            }
            else {
                break; // 如果当前节点的键值不小于父节点，则堆已调整完毕
            }
        }
    }

    // 向下调整堆，以保持最小堆的性质
```

```
    void heapifyDown(int index) {
        int leftChild, rightChild, smallest;

        while (true) {
            leftChild = 2 * index + 1; // 计算左子节点的索引
            rightChild = 2 * index + 2; // 计算右子节点的索引
            smallest = index; // 假设当前节点是最小的

            // 如果左子节点存在且左子节点的键值小于当前节点，更新最小节点
            if (leftChild < heap.size() && heap[leftChild].second <
heap[smallest].second) {
                smallest = leftChild;
            }

            // 如果右子节点存在且右子节点的键值小于当前最小节点，更新最小节点
            if (rightChild < heap.size() && heap[rightChild].second <
heap[smallest].second) {
                smallest = rightChild;
            }

            // 如果最小节点不是当前节点，进行交换并继续向下调整
            if (smallest != index) {
                std::swap(heap[index], heap[smallest]);
                // 更新 pos 数组中对应顶点的索引
                pos[heap[index].first] = index;
                pos[heap[smallest].first] = smallest;
                index = smallest; // 继续向下调整
            }
            else {
                break; // 如果最小节点是当前节点，堆已调整完毕
            }
        }
    }

public:
    // 构造函数，初始化空的堆
    MinHeap() {}

    // 析构函数，这里没有需要特殊处理的资源，默认析构函数就可以了
    ~MinHeap() {}

    // 插入一个新的元素，假设元素是 std::pair<int, int>，第一个值是顶点，第二个是键值
    void insert(T value) {
        heap.push_back(value); // 将新元素添加到堆的末尾
        int index = heap.size() - 1; // 获取新元素的索引
        pos[value.first] = index; // 记录该顶点在堆中的位置
        heapifyUp(index); // 向上调整堆，以保持最小堆的性质
    }

    // 删除并返回堆中的最小元素（堆顶元素）
    std::pair<int, int> deleteMin() {
        if (heap.empty()) { // 如果堆为空，抛出异常
            throw std::runtime_error("Heap is empty");
        }

        std::pair<int, int> minValue = heap[0]; // 保存堆顶元素
```

```
        heap[0] = heap.back(); // 将最后一个元素移到堆顶
        pos[heap[0].first] = 0; // 更新该顶点在堆中的位置
        heap.pop_back(); // 移除最后一个元素
        pos.erase(minValue.first); // 从 pos 中移除已删除的最小顶点
        if (!heap.empty()) {
            heapifyDown(0); // 向下调整堆，以保持最小堆的性质
        }
        return minValue; // 返回具有最小键值的顶点和键值
    }

    // 减少某个顶点的键值
    void decreaseKey(int vertex, int newDist) {
        if (pos.find(vertex) == pos.end()) {
            // 如果顶点不在堆中，则直接插入
            insert({ vertex, newDist });
        }
        else {
            int index = pos[vertex]; // 获取顶点在堆中的索引
            if (newDist < heap[index].second) { // 只有当新键值更小时才更新
                heap[index].second = newDist; // 更新顶点的键值
                heapifyUp(index); // 向上调整堆
            }
        }
    }

    // 检查堆是否为空
    bool isEmpty() const {
        return heap.empty(); // 如果堆为空，返回 true
    }

    // 检查某个顶点是否在堆中
    bool hasVertex(int vertex) const {
        return pos.find(vertex) != pos.end(); // 如果顶点在 pos 中，返回 true
    }

    // 显示堆中的所有元素
    void display() const {
        for (const auto& element : heap) { // 遍历堆数组中的所有元素
            std::cout << "(" << element.first << ", " << element.second << ") ";
// 打印顶点和键值
        }
        std::cout << std::endl;
    }
};


#endif // MIN_HEAP_H
```

`leftistheap.h`:

```
#ifndef LEFTIST_HEAP_H
#define LEFTIST_HEAP_H

#include <iostream>
```

```cpp
#include <limits>
#include <stdexcept>
#include <utility>


/*****************************************************************************************************************************
 * 这是左式堆的内部实现
 * 在本次，我们编写的数据结构都是模板类 未实例化时不指定类型名称
 * 因此无法按照传统的 .h 放名称 .cpp 放主体的方式实现
 * 为了结构完整性加了cpp文件，但它们全是空的
 * 所以在实现上会有注释“用**结构作为堆元素”
 *
 * 包含操作增删查减小键值等
 *

*****************************************************************************************************************************/

template <typename T>
class LeftistHeap {
private:
    // 节点结构体，表示堆中的每个节点
    struct Node {
        T key; // 键值对，key 是 std::pair<int, int>，第一个 int 是顶点，第二个 int 是对应的键值
        int npl; // 零路径长度 (NPL)
        Node* left; // 指向左子树的指针
        Node* right; // 指向右子树的指针

        // 构造函数，初始化节点的键值，并设置左右子节点为空，NPL 为 0
        Node(T value) : key(value), npl(0), left(nullptr), right(nullptr) {}
    };

    Node* root; // 左式堆的根节点

    // 合并两个左式堆，返回合并后的堆根节点
    Node* merge(Node* h1, Node* h2) {
        // 如果 h1 为空，则直接返回 h2
        if (!h1) return h2;
        // 如果 h2 为空，则直接返回 h1
        if (!h2) return h1;

        // 确保 h1 的根节点的键值小于 h2 的根节点的键值
        if (h1->key.second > h2->key.second) { // 假设 key 是 std::pair<int, int>，比较第二个值（键）
            std::swap(h1, h2); // 如果 h1 的键值较大，交换 h1 和 h2
        }

        // 递归地合并 h1 的右子树和 h2，结果作为新的右子树
        h1->right = merge(h1->right, h2);

        // 确保左子树的 NPL 不小于右子树，否则交换左右子树
        if (!h1->left || (h1->right && h1->left->npl < h1->right->npl)) {
            std::swap(h1->left, h1->right);
        }
```

```
        // 更新 NPL 值（右子树的 NPL 加 1，如果右子树为空，则 NPL 为 0）
        h1->npl = (h1->right ? h1->right->npl + 1 : 0);
        return h1; // 返回合并后的根节点
    }

public:
    // 构造函数，初始化根节点为空
    LeftistHeap() : root(nullptr) {}

    // 析构函数，释放堆中的所有节点
    ~LeftistHeap() {
        clear(root); // 递归清除所有节点
    }

    // 递归清除以 node 为根的子树
    void clear(Node* node) {
        if (node) {
            clear(node->left); // 清除左子树
            clear(node->right); // 清除右子树
            delete node; // 删除当前节点
        }
    }

    // 插入一个新的元素，假设元素是 std::pair<int, int>，其中第一个值是顶点，第二个是键值
    void insert(T value) {
        Node* newNode = new Node(value); // 创建一个新节点
        root = merge(root, newNode); // 合并新节点和当前堆
    }

    // 删除最小元素并返回其键值对
    std::pair<int, int> deleteMin() {
        if (!root) { // 如果堆为空，抛出异常
            throw std::runtime_error("空的! ");
        }

        std::pair<int, int> minValue = root->key; // 保存根节点的键值对，即最小值
        Node* oldRoot = root; // 保存当前根节点
        root = merge(root->left, root->right); // 合并根节点的左右子树
        delete oldRoot; // 删除旧的根节点
        return minValue; // 返回最小键值的顶点和键值
    }

    // 获取堆中最小元素的键值对（不删除）
    std::pair<int, int> getMin() const {
        if (!root) { // 如果堆为空，抛出异常
            throw std::runtime_error("空的! ");
        }
        return root->key; // 返回根节点的键值对，即最小值
    }

    // 检查堆是否为空
    bool isEmpty() const {
        return root == nullptr; // 如果根节点为空，堆为空
    }
```

```
    // 递归中序遍历并显示堆中的元素
    void inorder(Node* node) const {
        if (node) {
            inorder(node->left); // 递归遍历左子树
            std::cout << "(" << node->key.first << ", " << node->key.second << ")
"; // 显示顶点和键值
            inorder(node->right); // 递归遍历右子树
        }
    }

    // 显示整个堆的元素（中序遍历）
    void display() const {
        inorder(root); // 从根节点开始中序遍历
        std::cout << std::endl;
    }
};


#endif // LEFTIST_HEAP_H
```

`graph.h`:

```
#ifndef GRAPH_H
#define GRAPH_H

#include <iostream>
#include <vector>
#include <string>
#include <mutex>
#include <thread>

# include "../include/fibheap.h"
# include "../include/leftistheap.h"
# include "../include/minheap.h"


/*********************************************************************************
*********
 * 这是图和算法实现的头文件，我建议详细看这一部分
 *
 * 我们在这里把数据结构都include进来，但是具体数据结构是按照传入测试/算法函数的结构名确定的
 *
 * 数据结构的选择，我们选取了三种代表性的
 * 斐波那契堆 二项队列有的斐波那契堆都有，二项队列没有的斐波那契堆也有，所以选择斐波那契堆，避免
代码过长
 * 左式堆 左式堆与斜堆类似，都实现可能导致代码过长
 * 普通最小堆，这是一个数组版本的，在FDS学的，我们用这个和高级数据结构做对比
 *

*********************************************************************************
********/


class Graph { //这是整个图的类，除了数据结构的实现，其他所有东西都在里面
```

```
private:
    // 邻接节点结构体 用于储存图的节点所连的边（我们采用了链表来实现）
    struct AdjListNode {
        int vertex; //第二节点
        int weight; //权重
        AdjListNode* next;//链表指针
        AdjListNode(int v, int w) : vertex(v), weight(w), next(nullptr) {} //初始
化函数
    };


    int numVertices;                        // 图中的顶点数量
    int numEdges;                           // 图中的边数量

    std::vector<AdjListNode*> adjList;    // 邻接表
    std::mutex graphMutex;                  // 保护图的互斥锁，用于多线程环境

    // 私有成员函数：多线程处理一部分文件
    void processFilePart(const std::string& filename, std::streampos start,
std::streampos end, int tqdm);

    // 私有成员函数：从文件中读取图
    void loadGraphFromFileMultithreaded(const std::string& filename, int
numThreads);

public:

    // 构造函数：接受文件名并从文件中读取图（多线程）
    //构造函数里面就套了一个loadGraphFromFileMultithreaded函数，用于图的读取，也就是说这玩
意是用来生成图的
    Graph(const std::string& filename, int numThreads);

    // 析构函数 释放空间
    ~Graph();

    // 显示图的邻接列表 不建议大图调用，经过测试，给定的数据集的最小的也会显示很多内容难以识别
    void displayGraph() const;

    //算法实现 之所以直接暴露在外面是因为你也可以选择直接拿这个算法当借口来测试，没有那麼多IO操
作，简单直接，此处不设置示例
    //参数包括起始位置，终点，堆类型
    std::pair<std::vector<int>, int> dijkstra(int startVertex, int endVertex,
 const std::string& heapType);

    //测试方法
    //这一部分在main里有详细讲
    double test(int queryNumber, const std::string& filename, const std::string&
heapType, int num, int random, int a, int b);


    //两个封装起来的变量，表示顶点数目和边数目
    int getNumVertices() const {
        return numVertices;
    }

    int getNumEdges() const {
```

```
        return numEdges;
    }
};

#endif //GRAPH_H
```

`fibheap.h`:

```
#ifndef FIBONACCI_HEAP_H
#define FIBONACCI_HEAP_H

#include <iostream>
#include <cmath>
#include <vector>
#include <limits>
#include <stdexcept>


/*****************************************************************************************
*****************************
 * 这是斐波那契堆的内部实现
 * 在本次，我们编写的数据结构都是模板类 未实例化时不指定类型名称
 * 因此无法按照传统的 .h 放名称 .cpp 放主体的方式实现
 * 为了结构完整性加了cpp文件，但它们全是空的
 * 所以在实现上会有注释“用**结构作为堆元素”
 *
 * 我们在编写 .h 时，把传统的 .cpp 包含的内容分隔开，便于阅读
 *
 * 包含操作增删查减小键值等
 *

*****************************************************************************************
**************************/



template <typename T> //以T作为模板
class FibonacciHeap {
public:
    FibonacciHeap();
    ~FibonacciHeap();

    struct Node {
        T key;         // 存储的值（键）
        int val;       // 顶点的值
        int degree;    // 度数，表示有几个儿子
        Node* parent; // 父亲
        Node* child;  // 任意一个孩子
        Node* left;   // 左兄弟
        Node* right;  // 右兄弟
        bool mark;     // 是否标记
        // 构造函数 接受模板类型的k和int类型的v，v表示节点编号（在迪杰特拉斯算法中的编号
        Node(T k, int v) : key(k), val(v), degree(0), parent(nullptr),
child(nullptr), left(this), right(this), mark(false) {}
```

```
    };

    Node* minNode;
    int heapSize;

    bool isEmpty() const; //是否空
    void insert(T value, int vertex); //插入
    typename FibonacciHeap<T>::Node* getMin() const; // 修改为返回顶点的值
    void deleteMin(); //删最小
    void decreaseKey(Node* node, T newValue); //减小键值
    void deleteNode(Node* node); //删（在算法实现中没有用到）
    int size() const; // 大小

private:
    void link(Node* y, Node* x);  //链接，一个成为另一个孩子
    void consolidate(); //合并
    void cut(Node* x, Node* y); //剪切 合并的反操作
    void cascadingCut(Node* y);  //剪切 链接的反操作
};



////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////
//下面是实现



//构造函数
template <typename T>
FibonacciHeap<T>::FibonacciHeap() : minNode(nullptr), heapSize(0) {}

//析构函数
template <typename T>
FibonacciHeap<T>::~FibonacciHeap() {
    if (minNode) {
        // 释放所有节点
        std::vector<Node*> nodes;
        Node* current = minNode;
        do {
            nodes.push_back(current);
            current = current->right;
        } while (current != minNode);

        for (Node* node : nodes) {
            delete node;
        }
    }
}

template <typename T>
bool FibonacciHeap<T>::isEmpty() const {
    // 检查堆是否为空，如果最小节点为 nullptr，则堆为空
    return minNode == nullptr;
}
```

```cpp
template <typename T>
void FibonacciHeap<T>::insert(T value, int vertex) {
    // 插入一个新的节点到斐波那契堆中，节点包含一个值和对应的顶点编号
    Node* node = new Node(value, vertex); // 创建新节点
    if (!minNode) { // 如果最小节点为空，设置当前节点为最小节点
        minNode = node;
    }
    else { // 如果堆非空，将新节点插入到根列表中
        node->left = minNode; // 新节点的左指针指向最小节点
        node->right = minNode->right; // 新节点的右指针指向最小节点的右节点
        minNode->right->left = node; // 更新最小节点右侧节点的左指针
        minNode->right = node; // 更新最小节点的右指针
        if (value < minNode->key) { // 如果新插入的值比当前最小值还小，更新最小节点
            minNode = node;
        }
    }
    heapSize++; // 堆大小增加
}

template <typename T>
typename FibonacciHeap<T>::Node* FibonacciHeap<T>::getMin() const {
    // 返回最小节点，如果堆为空则抛出异常
    if (!minNode) throw std::runtime_error("空堆! ");
    return minNode; // 返回最小节点的指针
}

template <typename T>
void FibonacciHeap<T>::deleteMin() {
    // 删除堆中最小的节点
    if (!minNode) return; // 如果堆为空，直接返回

    Node* oldMin = minNode; // 保存最小节点
    if (oldMin->child) { // 如果最小节点有孩子
        Node* child = oldMin->child;
        do { // 遍历孩子链表，将孩子的 parent 指针置空
            child->parent = nullptr;
            child = child->right;
        } while (child != oldMin->child);

        // 合并孩子节点到根列表
        Node* minLeft = minNode->left;
        Node* childLeft = oldMin->child->left;

        minLeft->right = oldMin->child;
        oldMin->child->left = minLeft;

        childLeft->right = minNode->right;
        minNode->right->left = childLeft;
    }

    // 从根列表中移除最小节点
    minNode->left->right = minNode->right;
    minNode->right->left = minNode->left;

    if (minNode == minNode->right) {
        // 如果堆中只有一个节点，删除后堆为空
```

```
        minNode = nullptr;
    }
    else {
        // 否则，将最小节点设置为根列表中的下一个节点，并调用 consolidate 函数
        minNode = minNode->right;
        consolidate();
    }

    delete oldMin; // 删除旧的最小节点
    heapSize--; // 堆大小减少
}

template <typename T>
void FibonacciHeap<T>::decreaseKey(Node* node, T newValue) {
    // 减少节点的键值，如果新的值大于当前值，抛出异常
    if (newValue > node->key) {
        throw std::invalid_argument("太大! ");
    }
    node->key = newValue; // 更新节点的值
    Node* parent = node->parent;

    // 如果节点有父节点，且当前值小于父节点的值，切断节点与父节点的连接
    if (parent && node->key < parent->key) {
        cut(node, parent);
        cascadingCut(parent); // 执行级联剪切
    }

    // 如果当前值比最小值还小，更新最小节点
    if (minNode && node->key < minNode->key) {
        minNode = node;
    }
}

template <typename T>
void FibonacciHeap<T>::link(Node* y, Node* x) {
    // 将节点 y 链接到节点 x 上，y 将作为 x 的孩子
    y->left->right = y->right; // 从根列表中移除 y
    y->right->left = y->left;

    y->parent = x; // 设置 y 的父节点为 x
    if (!x->child) { // 如果 x 没有孩子，y 成为 x 的唯一孩子
        x->child = y;
        y->left = y->right = y;
    }
    else { // 如果 x 有孩子，将 y 插入到孩子链表中
        y->left = x->child;
        y->right = x->child->right;
        x->child->right->left = y;
        x->child->right = y;
    }
    x->degree++; // 更新 x 的度数
    y->mark = false; // 标记 y 为未被剪切
}

template <typename T>
void FibonacciHeap<T>::consolidate() {
```

```
    // 将根列表中的节点合并，以确保每个度数的节点只有一个
    int D = static_cast<int>(std::log2(heapSize)) + 1; // 计算所需的数组大小
    std::vector<Node*> A(D, nullptr); // 创建一个数组，用于存储各个度数的节点

    std::vector<Node*> rootList; // 创建根列表数组
    Node* current = minNode;
    if (current) { // 如果根列表不为空，遍历根列表中的所有节点
        do {
            rootList.push_back(current);
            current = current->right;
        } while (current != minNode);
    }

    // 遍历根列表，合并具有相同度数的节点
    for (Node* w : rootList) {
        Node* x = w;
        int d = x->degree;
        while (A[d]) { // 如果当前度数存在节点，合并
            Node* y = A[d];
            if (x->key > y->key) std::swap(x, y); // 保证 x 的键值更小
            link(y, x); // 将 y 链接到 x 上
            A[d] = nullptr; // 清空该位置
            d++; // 增加度数
        }
        A[d] = x; // 将节点放入数组中
    }

    minNode = nullptr; // 清空最小节点
    for (Node* w : A) { // 重新构建根列表
        if (w) {
            if (!minNode) {
                minNode = w;
                w->left = w->right = w;
            }
            else {
                w->left = minNode;
                w->right = minNode->right;
                minNode->right->left = w;
                minNode->right = w;
                if (w->key < minNode->key) {
                    minNode = w;
                }
            }
        }
    }
}

template <typename T>
void FibonacciHeap<T>::cut(Node* x, Node* y) {
    // 从 y 的孩子中移除 x，并将 x 添加到根列表
    if (x->right == x) { // 如果 x 是 y 的唯一孩子
        y->child = nullptr;
    }
    else { // 否则将 x 从孩子列表中移除
        x->left->right = x->right;
        x->right->left = x->left;
```

```
        if (y->child == x) {
            y->child = x->right;
        }
    }
    y->degree--; // 减少 y 的度数

    // 将 x 插入到根列表
    x->left = minNode;
    x->right = minNode->right;
    minNode->right->left = x;
    minNode->right = x;

    x->parent = nullptr; // 设置 x 的父节点为空
    x->mark = false; // 标记 x 为未被剪切
}

template <typename T>
void FibonacciHeap<T>::cascadingCut(Node* y) {
    // 级联剪切，如果父节点 y 已被标记，则递归剪切
    Node* z = y->parent;
    if (z) {
        if (!y->mark) { // 如果 y 没有被标记，则标记它
            y->mark = true;
        }
        else { // 如果 y 已被标记，剪切 y，并继续递归剪切其父节点 z
            cut(y, z);
            cascadingCut(z);
        }
    }
}

template <typename T>
void FibonacciHeap<T>::deleteNode(Node* node) {
    // 删除给定的节点，首先将其键值减少到负无穷，然后删除最小节点
    decreaseKey(node, std::numeric_limits<T>::min()); // 将键值减少到负无穷
    deleteMin(); // 删除最小节点
}

template <typename T>
int FibonacciHeap<T>::size() const {
    // 返回堆的大小
    return heapSize;
}

#endif // FIBONACCI_HEAP_H
```

References

1. 数据集

Author List

> filtered due to academic honesty

Declaration

We hereby declare that all the work done in this project titled "**Shortest Path Algorithm with Heaps**" is of our independent effort as a group.

Signatures

> filtered due to academic honesty