C++ 算法：图论（二）- 图的存储结构 讲义

一、 引言

1.1 问题引入

我们已经知道，一张校园地图可以被抽象为一个由顶点（建筑）和边（道路）组成的图。但计算机不认识“点”和“边”，我们必须找到一种方式，用代码来描述“哪个顶点和哪个顶点相连？”以及“连接它们的道路有多长？”这个问题。

换言之，我们需要一个**数据结构**，来充当理论模型与算法实现之间的桥梁。

而存储图有多种办法，不同的方法会影响后续图算法的**时间复杂度**和**空间复杂度**。

二、 存储方法一：邻接矩阵 (Adjacency Matrix)

2.1 核心思想：关系地图

邻接矩阵：我们创建一个二维数组，用它来记录**每一对顶点之间**是否存在边。

例如我们有一张城市间的航班距离表：

	北京	上海	广州
北京	0	1200km	2100km
上海	1200km	0	1400km
广州	2100km	1400km	0

这个表格就是一个邻接矩阵。行和列都代表图的顶点，表格中的`[行，列]`位置的数值，就代表了对应两个顶点间的关系（边的信息）。

2.2 实现细节

在C++中，我们通常使用一个二维数组 `G[N][N]` 来表示邻接矩阵，其中 `N` 是顶点的最大数量。`G[u][v]` 的值代表了从顶点 `u` 到顶点 `v` 的边的信息。

- **对于无权图**：
  - `G[u][v] = 1` 表示从 `u` 到 `v` 存在一条边。
  - `G[u][v] = 0` 表示从 `u` 到 `v` 不存在边。
- **对于带权图**：
  - `G[u][v] = w` 表示从 `u` 到 `v` 存在一条边，且其权重为 `w`。
  - 如果边不存在，我们可以用一个特殊值来表示，通常是 **0，-1 或 无穷大**。如果用一个很大的整数（可以用 `0x3f3f3f3f`）来模拟无穷大，这样如果需要进行一些计算的时候的时候不易溢出。
- **对于无向图**：
  - 图中的边是双向的，所以矩阵是**对称**的。即 `G[u][v]` 和 `G[v][u]` 的值总是相等。我们在存边时，需要同时设置这两个值。

2.3 代码实现

我们实现储存一个顶点数量为 n , 边的数量为 m 的一个有向带权图(权重定义为顶点之间的距离)。

```
#include <bits/stdc++.h>
using namespace std;

int main() {
    int n, m; // n个顶点, m条边
    cin >> n >> m;

    // vector的大小为 n x n，初始全部设置为 0，代表不存在任何边
    vector<vector<int> > G(n, vecotr<int>(n, 0));

    for (int i = 0; i < m; i++) {
        int u, v, w;
        cin >> u >> v >> w; // 读入一条从u到v，权重为w的边

        // 如果是无向图，需要存两条边
        G[u][v] = w;
        G[v][u] = w; // 若是有向图，则没有这一行
    }

    // 查询从顶点2到顶点3的距离(权值)
    if (G[2][3] == 0) {
        cout << "顶点2和3没有边相连" << endl;
    } else {
        cout << "顶点2到3的距离是: " << G[2][3] << endl;
    }

    return 0;
}
```

2.4 优劣分析

- **优点**:
  - **实现简单**：非常直观，易于理解和编写。
  - **查询效率高**：判断两个顶点 `u` 和 `v` 是否有边只需访问 `G[u][v]` 即可，时间复杂度为 **O(1)**。
- **缺点**:
  - **空间复杂度高**：无论图中有多少条边，都需要一个 `N*N` 的空间，空间复杂度为 **O(N²)**。
  - **对于稀疏图（边数远小于顶点数的平方）来说，空间浪费巨大。**例如，1万个点但只有100条边的图，矩阵中绝大部分空间都是储存的 0，代表没有边相连，被浪费了。
  - **遍历邻居效率低**：要找到一个顶点的所有邻居，必须遍历一整行，时间复杂度为 **O(N)**。

三、 存储方法二：邻接表 (Adjacency List)

3.1 核心思想：记录我的邻居是谁

邻接矩阵像一张全国地图，记录了所有城市两两之间的关系，信息很全但很占地方。而邻接表则更像每个人的通讯录，只记录与自己有直接关系的人。

具体来说，我们为图中的**每一个顶点**，都维护一个**列表**，这个列表里只存放与该顶点**直接相连**的其他顶点（以及边的权重，如果是带权图）。

3.2 通过`struct`封装实现邻接表

为了更好地组织和管理图的数据，我们可以使用一个自定义的结构体`Graph`来表示和存储一个图。这种方式将图的所有关键信息打包在一起，让代码结构更清晰。

1. 定义图的结构体

```
// 定义图的结构体
struct Graph {
    int V = -1, E = -1; // V:顶点数量, E:边的数量
    vector<vector<int>> adjList; // 邻接表
};
```

- `V`和`E`分别存储图的顶点数和边数。
- `adjList`是一个`vector`的`vector`。一个动态的二维数组。`adjList[u]`本身就是一个`vector`，它将作为顶点`u`的“邻居通讯录”，里面存放所有与`u`直接相连的顶点编号。

2. 初始化图的结构

在使用这个图之前，我们需要一个初始化函数来设置其规模，并准备好存放邻居列表的空间。

```
struct Graph {
    int V = -1, E = -1; // V:顶点数量, E:边的数量
    vector<vector<int>> adjList; // 邻接表

    // 初始化函数，设置顶点和边的数量
    void setCount(int new_V, int new_E) {
        V = new_V;  E = new_E;
        // 一共有 V 个顶点，故需要 V 个对应顶点的通讯录来储存直接相连的顶点编号
        adjList.resize(new_V);
    }
};
```

- `setCount`函数接收顶点的数量`new_V`，然后调用`resize`方法，将`adjList`变成一个包含`new_V`个空`vector`。

3. 添加边

图的结构准备好后，我们需要一个方法来添加边，以描述顶点间的连通情况。

```
struct Graph {
    // 省略之前的结构体内容...以下为添加边的函数
    void addEdge(int src, int dest) {
        if (V == -1) return; // 确保图已被初始化

        // 代表存在一条边src->sest, 在起点 src 的相连顶点数组中添加顶点 dest
        // 注意如果是无向图，还需要添加从 dest 到 src 的反向边
        adjList[src].push_back(dest);
    }
};
```

- `addEdge`函数就是在起点`src`的相连顶点列表`adjList[src]`的末尾，添加上终点`dest`。

- **注意**：对于**无向图**，由于边是双向的，添加一条 `(u, v)` 的边，意味着 `v` 是 `u` 的邻居，同时 `u` 也是 `v` 的邻居。因此需要**双向添加**。

3.3 代码实现：使用`Graph`结构体

下面我们通过一个完整的例子，演示如何使用这个 `Graph` 结构体来存储一个**有向无权图**。

```
int main() {
    // 创建一个图对象
    Graph G;
    int V, E;
    cin >> V >> E;

    // 调用初始化函数，准备好邻接表的结构
    G.setCount(V, E);

    // 读取 E 条边，并添加到图中
    for (int i = 0; i < E; ++i) {
        int src, dest;
        cin >> src >> dest;
        G.addEdge(src, dest);
    }

    // 此时，图G中已经存好了所有的顶点和边信息，可以进行操作，例：打印顶点3的所有邻居
    for (int neighbor : G.adjList[3]) {
        cout << neighbor << " ";
    }
    return 0;
}
```

3.4 优劣分析

- **优点**：
  - **空间效率高**：邻接表只存储实际存在的边，空间复杂度为 **O(N + M)**（N为顶点数，M为边数）。对于使用邻接矩阵的的 **O(N²)**，节省了大量空间。
  - **遍历邻居效率高**：邻接表要找到一个顶点的所有邻居，只需遍历其对应的 `vector` 即可，时间复杂度为 **O(出度(u))**，其中 `出度(u)` 是顶点u的度。优于邻接矩阵的遍历邻居时间复杂度 **O(N)** 。
- **缺点**：
  - **查询效率较低**：邻接表判断两个顶点 `u` 和 `v` 是否有边，需要遍历 `adj[u]` 这个列表，时间复杂度为 **O(出度(u))**，不如邻接矩阵直接查询的 O(1) 快。

四、 总结

- 我们学习了两种主流的图存储方法：**邻接矩阵**和**邻接表**。
- **邻接矩阵**是一个二维数组，用 `G[u][v]` 存储边的信息，它像一张**完整的地图**，优点是查询快，缺点是浪费空间。
- **邻接表**是一个 `vector` 数组，`adj[u]` 存储了 `u` 的所有邻居，它像一本**通讯录**，优点是节省空间，缺点是查询稍慢。选择主要在于分析图的**疏密程度**：**稠密用矩阵，稀疏用表。**
- **在信息学竞赛和实际应用中，由于图的稀疏性，其实邻接表的使用频率会稍高于邻接矩阵。**