C++ 算法：图论（三）- 图的遍历 讲义

一、 引言

1.1 问题引入

我们已经熟悉了数组和树的遍历。数组的遍历是一条直线，从头到尾即可。树的遍历虽有岔路，但结构清晰，从父到子，不会回头。

但是，图的遍历要复杂得多。我们面临着两个新的问题：

1. **连接的顶点多**：在树中，一个节点最多连接少数几个子节点。但在图中，一个顶点可能连接着任意数量的其他顶点，我们该如何选择前进的方向？
2. **存在“环”**：在树中，我们绝不会走回祖先节点。但在图中，从A出发，经过B、C，完全可能再回到A，形成一个“环”。如果程序处理不当，就会陷入无限循环。

让我们通过书本中的**【例18-3 查找文献】**案例，来学习如何解决这些问题。

小K正在阅读一篇编号为1的博客文章。这篇文章引用了若干其他文章作为参考文献，而这些参考文献又各自引用了更多的文章，形成了一个庞杂的引用网络。小K求知欲旺盛，他决定：**只要读了一篇文章，就必须把它引用的所有没读过的文章都读一遍。**

现在，我们拥有这个引用网络的完整数据（`n`篇文章，`m`条引用关系）。问题是：我们如何设计一个策略，能帮助小K**不重复、不遗漏**地看完所有他从文章1出发能接触到的文章？

我们可以对问题进行**抽象**为图论问题：

- 每一篇**文章**，都可以看作一个**顶点**。
- 每一条从文章X到文章Y的**引用关系**，都可以看作一条从X指向Y的**有向边**。

因此，这个问题被成功地转化为了一个图论问题：**从图的某个起始顶点出发，遍历所有可达的顶点。**

二、 核心思想：如何做到“不重复、不遗漏”？

2.1 用`visited`数组避免重复

为了解决图遍历中可能出现的“环”和“重复访问”问题，我们引入一个关键工具：`visited`**布尔数组。**

- **定义**：`bool visited[N];`
- **作用**：用于记录对应的顶点我们是否已经访问过。`visited[i] = true` 表示顶点 `i` 我们已经访问过这个顶点了。
- **规则**：
    1. 每当我们**首次**访问一个顶点时，立刻在“笔记”上记录下来：`visited[i] = true`。
    2. 在尝试访问任何一个顶点之前，检查：`if (!visited[i])`。我们只访问还没有到过的顶点。

这个`visited`数组是保证遍历**不重复**的关键，是DFS和BFS都必须使用的基础工具。

2.2 DFS 与 BFS 遍历图

解决了不重复的问题，我们如何保证不遗漏呢？有两种策略：

- **策略一：深度优先（DFS）- “一条路走到底”**

- 当你读了文章A，发现它引用了B和C。你选择立刻深入进去，先把B以及B的所有参考文献、B的参考文献的参考文献……全部看完，直到这条链条的尽头。然后才“回溯”过来看文章C。
- 一条路走到黑，不撞南墙不回头。

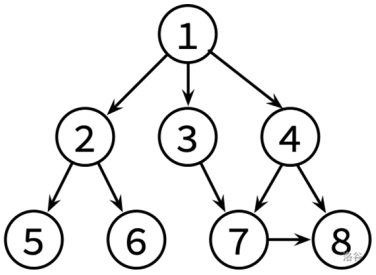
- **策略二：广度优先（BFS）- “逐层拓展”**
  - 当你读完文章A，发现它引用了B和C。你不会立刻深入，而是先把同一层级的B和C都看完。然后，再去看B和C共同引用的下一层级的所有文章。
  - 一层一层地看，像水波一样由近及远地扩散。

三、 深度优先遍历 (DFS)

3.1 算法精讲

DFS体现的是一条路走到底的策略。其过程符合**递归**的定义：要访问顶点`u`的所有可达点，我们先访问`u`，然后对`u`的每一个邻居`v`，再去递归地执行“访问`v`的所有可达点”这一相同的任务。它的两个重要动作是**深入探索**和**回溯**。

3.2 步骤拆解（以查找文献的样例数据为例子）



DFS的遍历过程为：

1. **从1出发**：访问**1**，标记`visited[1]=true`。1的邻居有2, 3, 4。
2. **深入2**：选择邻居2，递归调用DFS(2)。访问**2**，标记`visited[2]=true`。2的邻居有5, 6。
3. **深入5**：选择邻居5，递归调用DFS(5)。访问**5**，标记`visited[5]=true`。5没有邻居，已到达尽头。
4. **回溯到2**：从5的调用中返回。检查2的下一个邻居6。
5. **深入6**：选择邻居6，递归调用DFS(6)。访问**6**，标记`visited[6]=true`。6没有邻居。
6. **回溯到2**：从6返回。2的所有邻居都已探索完毕。
7. **回溯到1**：从2返回。检查1的下一个邻居3。
8. **深入3 -> 7 -> 8**：重复上述过程，一路深入到8，再层层回溯到1。
9. **探索4**：检查1的下一个邻居4，递归调用DFS(4)。访问**4**，标记`visited[4]=true`。
10. **检查4的邻居**：尝试访问7和8，但检查发现`visited[7]`和`visited[8]`都已为`true`，故跳过。
11. **回溯**：从4返回1。1的所有邻居探索完毕。遍历结束。

3.3 代码实现

```
// DFS函数，v是当前访问的顶点，adj: 邻接表，visited: 访问数组
void DFS(int v) {
    visited[v] = true;
    cout << v << " ";        // 访问当前节点

    // 遍历v的所有邻居
```

```
    for (int neighbor : adj[v]) {
        // 如果邻居未被访问过，则递归深入
        if (!visited[neighbor]) {
            DFS(neighbor);
        }
    }
}

// 题目要求中邻居按小编号优先，所以在注意在主函数中对邻接表排序
for (int i = 1; i <= n; i++) {
    sort(adj[i].begin(), adj[i].end());
}
```

四、 广度优先遍历 (BFS)

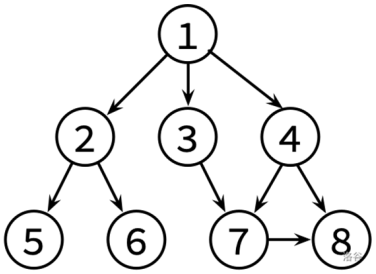
4.1 算法思路

BFS是“逐层扩散”的思路。它必须先处理完最早发现的节点，再处理新发现的节点。这种“先进先出”的规则，契合了**队列（Queue）**的数据结构。

算法流程：

1. 创建一个队列，并将起始顶点放入队列。标记起始顶点为已访问。
2. 当队列不为空时，循环执行以下操作：
   a. 从**队头**取出一个顶点`u`。
   b. 访问`u`。
   c. 遍历`u`的所有邻居`v`，如果`v`未被访问过，则标记`v`并将其放入**队尾**。

4.2 步骤拆解（以查找文献的样例数据为例子）



步骤	操作	队列状态 (Queue)	已访问 (Visited)	输出
1	初始状态，1入队	`[1]`	{1}	
2	1出队，访问1，其邻居2,3,4入队	`[2, 3, 4]`	{1,2,3,4}	1
3	2出队，访问2，其邻居5,6入队	`[3, 4, 5, 6]`	{1,2,3,4,5,6}	2
4	3出队，访问3，其邻居7入队	`[4, 5, 6, 7]`	{1,2,3,4,5,6,7}	3
5	4出队，访问4，其邻居8入队	`[5, 6, 7, 8]`	{1,2,3,4,5,6,7,8}	4
...	...	...	...	...
10	队列为空，结束	-	-	

最终访问顺序：1 -> 2 -> 3 -> 4 -> 5 -> 6 -> 7 -> 8

4.3 代码实现

```
// DFS函数，v是当前访问的顶点，adj: 邻接表，visited: 访问数组
void BFS(int v) {
    queue<int> q;

    q.push(v);          // 起始点入队
    visited[v] = true;

    while (!q.empty()) {
        int u = q.front();  q.pop();   // 取出队头

        cout << u << " ";              // 访问

        // 根据邻接表遍历所有邻居
        for (int v : adj[u]) {
            if (!visited[v]) {
                visited[v] = true;
                q.push(v);          // 新发现的顶点入队
            }
        }
    }
}
// main函数与DFS版本只是调用函数不同而已
```

五、 总结与应用

5.1 如何将现实问题抽象为图?

图的表达能力极强，世界上绝大多数涉及“物体”和“连接”的问题，都可以抽象为图。只需问自己两个问题：

1. **物体是什么?** -> 它们成为图的**顶点**。
2. **连接是什么?** -> 它们成为图的**边**。

- **社交网络**：用户是顶点，好友关系是边。找“朋友的朋友”就是图的遍历。
- **城市导航**：路口是顶点，道路是边。找“最短路线”就是图的最短路径搜索。

5.2 拓展：非连通图的遍历

在本题中，我们只需从顶点1开始遍历。但如果一个图由多个互不相连的部分组成（非连通图），我们该如何遍历所有顶点?

解决办法：在 `main` 函数中使用一个 `for` 循环，从 1 到 n 检查每一个顶点 `i`。如果发现 `!visited[i]`，就说明我们发现了一个新的、独立的连通块，此时再以 `i` 为起点，调用DFS或BFS来探索这个新部分。

```
for (int i = 1; i <= n; i++) {
    if (!visited[i]) {   // 如果还没有被访问过，从点i开始遍历这个新的连通图
        DFS(i); // 或 BFS(i)
    }
}
```