C++ 算法：图论（四）- 反向图

一、 引言

1.1 问题引入：寻找可达的最大顶点

【例18-4】 图的遍历

给出一个 N 个点、M 条边的有向图。对于图中的每一个顶点 v，我们需要求出 A(v)，它表示**从点 v 出发，沿着有向边能到达的所有顶点中，编号最大的那一个。**

例如，在一个图中，如果从顶点3开始出发只能到达 {3, 5, 8, 2}，那么 A(3) 的值就是8。我们需要为图中所有的点（从1到N）都计算出这个值。

二、 解题思路

2.1 思路一：暴力遍历

面对这个问题，我们的第一反应通常是：对每一个点 `u` （从1到N），都从它出发做一次完整的 DFS 或 BFS，记录下所有能访问到的顶点，然后从中找出一个最大值。

- **复杂度分析**：
  - 一次完整的图遍历，复杂度为 O(N+M)。
  - 我们需要对 N 个点都执行一次。
  - 总时间复杂度 = N * O(N+M) = $O(N^2 + N*M)$。
- **结论**：题目给出的数据范围 N, M 高达 10^5。N^2 级别的复杂度会达到 10^{10}，程序必然超时。**暴力方法虽然思路正确，但效率过低，不是可行解。**

2.2 思路二：记忆化搜索

既然暴力不行，我们来优化。分析 A(u) 的答案构成：
`A(u) = max(u, A(v₁), A(v₂), ...)`，其中 `v₁`，`v₂`... 是 `u` 的所有邻居。

这是一个递归的结构，但纯粹的递归会产生海量的**重复计算**。例如，计算A(0)时可能需要算A(2)，而计算A(1)时可能也需要算A(2)。A(2)会被重复计算多次。

优化方案：记忆化搜索
我们创建一个`Answer`数组，`Answer[i]` 用来存储 A(i) 的计算结果。

```
int A(int u) {
    if (Answer[u] != 0) { // 如果算过了
        return Answer[u]; // 直接返回记好的答案
    }
    int ans = u;
    for (int v : adj[u]) {
        ans = max(ans, A(v));
    }
    Answer[u] = ans; // 算出来后，记在笔记本上
    return ans;
}
```

2.3 缺陷：有环图中的循环依赖

记忆化搜索看似能解决问题，但在有向有环图中，它会陷入死循环。
考虑这样一个环：`2 -> 4 -> 3 -> 2`

- 为了计算 `A(2)`，程序需要先知道 `A(4)`。
- 为了计算 `A(4)`，程序需要先知道 `A(3)`。
- 为了计算 `A(3)`，程序需要先知道 `A(2)`。

这就形成了一个逻辑上的循环：一个尚未完成计算的函数，被它自己的调用链反向依赖。就像老板问经理要项目结果，经理问员工，员工却反过来问老板结果是什么。程序会因无限递归而崩溃。

三、 解决方案：反向图

既然正向求解（我能到哪？）的思路在有环时碰壁了，我们不妨**逆转思维**。

3.1 从“我去哪”到“谁来找我”

- **正向思考：我（顶点`u`）出发，能到达的最大编号是多少？
- **反向思考：对于一个编号很大的顶点 `k`，有哪些点可以到达它？对于所有能到达`k`的点`u`来说，`A(u)`的候选答案里就必然包含了`k`。

3.2 反向图的定义与构建

为了实现“谁能到我”的查询，我们引入**反向图**。

- **定义**：对于一个有向图 G，其反向图 G_rev 拥有和 G 完全相同的顶点。对于 G 中的任意一条边 `u -> v`，在 G_rev 中都对应一条方向相反的边 `v -> u`。
- **构建**：在读入原图的边 `u -> v` 时，我们不执行 `adj[u].push_back(v)`，而是执行 `rev_adj[v].push_back(u)`，即在**终点v**的邻接表中加入**起点u**。
- **功能**：在**反向图**上从顶点 `k` 开始遍历，所有能被访问到的点，就是在**原图**中所有能到达 `k` 的点。

3.3 结合贪心思想

我们的目标是求最大值，一个很自然的**贪心**想法是：让编号越大的顶点，越有优先权去宣告自己是答案。

算法流程：

1. **反向建图**：读入 `m` 条边 `u -> v`，构建反向图（添加边 `v -> u`）。
2. **初始化**：创建一个 `ans` 数组，全部初始化为-1，表示所有点的答案待定。
3. **倒序遍历与广播**：
    - 我们从大到小遍历所有顶点 `i` (从 `N` 到 `1`)。
    - 对于当前的顶点 `i`，检查 `ans[i]`。
    - **如果 `ans[i]` 仍为-1**：这说明 `i` 没有被任何比它更大的顶点“覆盖”过。它就是它所在“可达区域”的最大顶点。此时，我们从 `i` 出发，在**反向图**上进行一次遍历（DFS或BFS），将所有能访问到且`ans`值仍为-1的顶点`u`，都标记上答案：`ans[u] = i`。
    - **如果 `ans[i]` 不为-1**：这说明 `i` 已经被一个比它更大的顶点（比如 `j > i`）“覆盖”了，那么 `i` 就不可能是它自己的最终答案。我们直接跳过，什么也不做。
4. **输出结果**：倒序循环结束后，`ans`数组中就存储了所有点的最终答案。

四、 代码实现

```
vector<vector<int> > rev_adj; // 存储反向图的邻接表
vector<int> ans;              // 存储最终答案

// 在反向图上进行DFS，将答案i“广播”出去
void dfs(int u, int max_node_val) {
    ans[u] = max_node_val; // 标记当前点的答案

    for (int v : rev_adj[u]) {
        if (ans[v] == 0) { // 如果邻居还未被广播过
            dfs(v, max_node_val);
        }
    }
}

int main() {
    int n, m;
    cin >> n >> m;
    rev_adj.resize(n + 1);
    ans.resize(n + 1);
    // 反向建图
    for (int i = 0; i < m; i++) {
        int u, v;
        cin >> u >> v;
        rev_adj[v].push_back(u); // 建立 v -> u 的反向边
    }

    // 倒序遍历与广播
    for (int i = n; i >= 1; i--) {
        if (ans[i] == 0) { // ans[i]为0代表未被访问过
            // i没有被更大的节点覆盖，它就是自己区域的最大顶点
            // 开始以i为答案，在反向图上传播
            dfs(i, i);
        }
    }

    for (int i = 1; i <= n; i++) {
        cout << ans[i] << " ";
    }

    return 0;
}
```

五、 反向图的应用与总结

5.1 什么时候应该用反向图?

反向图本质上是一种**视角转换工具**。当遇到以下情况时，可以考虑使用它：

1. **反向可达性查询**：当问题关心的不是“从A能到哪”，而是“谁能到A”时。
2. **正向求解遇阻**：当正向的递归或递推因为“循环依赖”或需要“未来状态”而陷入困境时，可以尝试从确定的终点或最优解出发，反向地去影响和更新前端节点。

5.2 总结

- 面对一个复杂的图问题，直接的暴力解法往往因为效率问题而不可行。
- **记忆化搜索**是优化递归、避免重复计算的强大技巧，但要警惕它在**有环图**中可能导致的**循环依赖**问题。
- **反向图**提供了一种全新的解题视角，能将“A能到哪”的问题转化为“谁能到A”，从而将一个复杂的依赖问题转化为标准的图遍历问题。