中山大学计算机学院本科生实验报告

（2025学年第1学期）

课程名称：数据结构与算法　　实验任课教师：张子臻

年级	2024级	**专业（方向）**	计算机科学与技术（人工智能与大数据）
学号	242325157	**姓名**	梁玮麟
电话	18620062182	**Email**	3371676041@qq.com
开始日期	2025.12.10	**结束日期**	2025.12.11

第一题

1、实验题目

⭐ z14-DAG?

题目描述

输入一个有向图，判断该图是否是有向无环图（Directed Acyclic Graph）。

输入描述

输入的第一行包含两个整数 n 和 m，n 是图的顶点数，m 是边数。
1 ≤ n ≤ 100，0 ≤ m ≤ 10000。
接下来的 m 行，每行是一个数对 u v，表示存在有向边 (u, v)。顶点编号从 1 开始。

输出描述

如果图是 DAG，输出 1，否则输出 0。

输入样例

```
3 3
1 2
2 3
3 1
```

输出样例

```
0
```

2、实验目的

- 理解有向图中“环”的概念。
- 掌握利用拓扑排序判断一个有向图是否为 DAG 的方法。
- 学会使用入度数组和队列实现 BFS 的拓扑排序。

3、算法设计

设计思路

1. **图的构建**
   - 使用 `vector<list<int>> g(n+1)` 存储邻接表，编号从 1 开始。
   - 每读入一条有向边 u→v，将 v 加入 g[u]，并使入度 `in[v]++`。
2. **初始化队列**
   - 将所有入度为 0 的结点加入队列 `q`。
   - 若图为 DAG，则最终所有节点都能被弹出。
3. **拓扑排序过程**
   - 每次从队列取出一个节点 `cur`。
   - 对所有 `cur` 的后继结点 `next` 执行 `--in[next]`，若其入度变为 0，则入队。
   - 用一个 `result` 链表记录输出顺序（本题最后不输出顺序，只判断是否拓扑成功）。

4. **判断是否有环**
    - 若拓扑排序中输出的节点数 `< n`，说明存在至少一部分节点入度始终 >0，即存在环。
    - 输出 1 表示无环（DAG），输出 0 表示有环。

流程图

```
Start
  |
  |--> 输入 n, m
  |--> 初始化邻接表 g 和入度数组 in
  |--> 读入所有边 (u, v):
  |        g[u].push_back(v)
  |        in[v]++
  |
  |--> 建立队列 q
  |--> 将所有 in[i] = 0 的结点入队
  |
  |--> while q 非空:
  |        cur = q.front(); q.pop()
  |        对于每个 next ∈ g[cur]:
  |              in[next]--
  |              若 in[next] == 0:
  |                    q.push(next)
  |        将 cur 加入 result
  |
  |--> 若 result.size() == n:
  |        输出 1
  |    否则:
  |        输出 0
  |
End
```

复杂度分析

1. **建图**
    - 邻接表初始化：O(n)
    - 插入所有 m 条边：O(m)
2. **拓扑排序**
    - 每个点最多入队一次：O(n)
    - 每条边最多被访问一次：O(m)

总体时间复杂度：**O(n + m)**
总体空间复杂度：**O(n + m)**（邻接表 + 入度数组 + 队列）

细节注意

- 利用拓扑排序中无法进入环状结构的特性。

具体实现

```
#include <iostream>
#include <vector>
#include <queue>
#include <list>
using namespace std;
int main() {
    int n, m;
    cin >> n >> m;
    vector<list<int>> g(n + 1);
    vector<int> in(n + 1, 0);
    queue<int> q;
    list<int> result;
```

```
    for (int i = 0; i < m; i++) {
        int u, v;
        cin >> u >> v;
        in[v]++;
        g[u].push_back(v);
    }
    for (int i = 1; i < n + 1; i++) {
        if (!in[i])
            q.push(i);
    }
    while (!q.empty()) {
        int cur = q.front();
        q.pop();
        for (auto next : g[cur]) {
            if (--in[next] == 0)
                q.push(next);
        }
        result.push_back(cur);
    }
    cout << (result.size() == n ? 1 : 0) << endl;
}
```

4、程序运行与测试

测试样例一

- 标准输入：

```
4 6
1 2
2 3
3 1
1 3
1 4
4 2
```

- 实际输出：

```
0
```

- 期望输出：

```
0
```

测试样例二

- 标准输入：

```
4 5
1 2
2 3
1 3
1 4
4 2
```

- 实际输出：

```
1
```

- 期望输出：

```
1
```

5、实验总结与心得

- 深刻理解了拓扑排序的本质是不断删除入度为 0 的顶点。
- 在拓扑排序中无法处理所有节点时，即可证明有向图中存在环。
- 通过本题熟练掌握了入度数组与队列在有向图判环中的作用。

第二题

1、实验题目

⭐ z14-Euler Euler

Description

Given an undirected graph G, decide whether G has an Euler Circuit, Euler Path, or neither.

Input

There are multiple test cases. The first line contains an integer T, indicating the number of test cases.
Each test case begins with a line containing two integers, 1 ≤ n ≤ 1000 and
1 ≤ m ≤ n × (n − 1) / 2,
where n is the number of nodes (numbered from 1 to n) and m is the number of edges.
In the following m lines, each line has two integers indicating two adjacent nodes of an edge.

Output

For each test case, print a string `Euler Circuit`, `Euler Path`, or `Neither`.

Sample Input

```
3
3 3
1 2
1 3
2 3
3 2
1 2
2 3
4 3
1 2
1 3
2 3
```

Sample Output

```
Euler Circuit
Euler Path
Neither
```

2、实验目的

- 熟悉无向图中点度与欧拉路径（回路）之间的关系。
- 掌握并查集用于判断图是否连通（除孤立点外）。
- 综合使用点度计数与连通性判断完成欧拉性质分类。

3、算法设计

设计思路

1. **读入并初始化图结构**
   - 每个顶点建一个独立集合 `root[i] = i`。
   - 维护点度数组 `d[i]`。

2. **读入所有边并合并集合**
   - 每读到一条无向边 (u, v)：
     - `d[u]++`，`d[v]++`
     - 使用 find() 找到根并 union 两集合。
3. **连通性判断**
   - 统计根节点个数 `TreeNum`。
   - 若包含多个连通块，则无法存在欧拉路径或回路（除特殊情况但本题保证无自回复杂结构），输出 `Neither`。
4. **欧拉性质判定规则**
   - 若所有点度均为偶数：Euler Circuit
   - 若恰有两个点度为奇数：Euler Path
   - 否则：Neither

流程图

```
Start
  |
  |--> 输入 T
  |--> while T--:
  |         输入 n, m
  |         初始化 root[i] = i, d[i] = 0
  |
  |         读入 m 条边:
  |               d[u]++, d[v]++
  |               合并 find(u) 与 find(v)
  |
  |         统计 TreeNum = 连通块数量
  |         若 TreeNum != 1:
  |               输出 "Neither"
  |         否则:
  |               cnt = 所有 d[i] 为奇数的顶点个数
  |               若 cnt == 0:
```

```
  |                   输出 "Euler Circuit"
  |               若 cnt == 2:
  |                   输出 "Euler Path"
  |               否则:
  |                   输出 "Neither"
  |
End
```

复杂度分析

1. **建图与并查集**
   - 每条边执行两次 find + 一次 union：O(m α(n))
   - α(n) 为反 Ackermann 函数，趋近常数。
2. **点度扫描 & 连通块统计**
   - 遍历 n 次：O(n)

总体时间复杂度：**O(n + m α(n))**
总体空间复杂度：**O(n)**

细节注意

- 注意需要判断是否连通图，需要使用并查集检索。
- 欧拉路径和欧拉环路对于无向图的判定略有不同，环路需要每个点的度都是偶数，路径有两个点的度是奇数。

具体实现

```cpp
#include<iostream>
#include<vector>
using namespace std;
```

```
int find(int v, vector<int>& root){
    return v==root[v]? v : root[v]=find(root[v],root);
}
int main(){
    int T;
    cin>>T;
    while(T--){
        int n,m;
        cin>>n>>m;
        vector<int> d(n+1,0);
        vector<int> root(n+1,0);
        for(int i=0;i<n+1;i++){
            root[i]=i;
        }
        for(int i=0;i<m;i++){
            int u,v;
            cin>>u>>v;
            d[v]++;
            d[u]++;
            int ru,rv;
            ru=find(u,root);
            rv=find(v,root);
            if(ru!=rv){
                root[ru]=rv;
            }
        }
        int cnt=0;
        int TreeNum=0;
        for(int i=1;i<n+1;i++){
            cnt+=d[i]%2;
            if(root[i]==i){
                TreeNum++;
            }
        }
        if(TreeNum!=1){
            cout<<"Neither"<<endl;
        }
        else if(cnt==0){
            cout<<"Euler Circuit"<<endl;
```

```
        }else if(cnt==2){
            cout<<"Euler Path"<<endl;
        }else{
            cout<<"Neither"<<endl;
        }
    }
}
```

4、程序运行与测试

测试样例一

- 标准输入：

```
3
3 3
1 2
1 3
2 3
3 2
1 2
2 3
4 3
1 2
1 3
2 3
```

- 实际输出：

```
Euler Circuit
Euler Path
Neither
```

- 期望输出：

```
Euler Circuit
Euler Path
Neither
```

测试样例二

- 标准输入：

```
1
6 5
1 2
2 3
3 4
4 1
5 6
```

- 实际输出：

```
Neither
```

- 期望输出：

```
Neither
```

5、实验总结与心得

- 本题强化了欧拉路径与欧拉回路的判定条件记忆。
- 通过并查集检查连通性，使得判定过程更严谨。
- 进一步体会到图论问题往往需要点度与连通性联合作用才能得出完整结论。

第三题

1、实验题目

⭐ z14-Ordering Tasks

题目描述

John has n tasks to do. Unfortunately, the tasks are not independent and the execution of one task is only possible if other tasks have already been executed.

输入描述

There are multiple test cases. The first line contains an integer T, indicating the number of test cases.
Each test case begins with a line containing two integers, 1 ≤ n ≤ 100000 and
1 ≤ m ≤ 100000。
n is the number of tasks (numbered from 1 to n) and m is the number of direct precedence relations between tasks.
After this, there will be m lines with two integers i and j, representing the fact that task i must be executed before task j.
It is guaranteed that no task needs to be executed before itself either directly or indirectly.

输出描述

For each test case, print a line with n integers representing the tasks in a possible order of execution.
If multiple solutions exist, output the smallest one by lexical order.

输入样例

```
1
5 5
3 4
4 1
3 2
2 4
5 3
```

输出样例

```
5 3 2 4 1
```

2、实验目的

- 熟悉拓扑排序在任务调度与依赖关系问题中的应用。
- 理解使用优先队列保证“字典序最小”拓扑序的技巧。
- 进一步掌握 BFS 型拓扑排序的细节与实现方式。

3、算法设计

设计思路

1. **构建图与入度数组**
   - 使用邻接表 `g[i]` 保存所有 i→j 的依赖。
   - 对每条边 (i, j)，执行 `g[i].push_back(j)`，并使 `in[j]++`。
2. **建立最小堆（优先队列）**
   - 为满足“字典序最小”，使用 `priority_queue<int, vector<int>, greater<int>>`。
   - 将所有 `in[i] == 0` 的节点压入堆中。
3. **拓扑排序过程**
   - 每次取出堆顶最小的编号 `c`：
     - 加入结果列表。

    - 遍历 g[c] 中的所有后继 `next`：
        - `in[next]--`
        - 若降为 0，则入堆。
4. **输出结果**
    - 因为题意保证不存在环，所以最终会输出 n 个任务。
    - 输出结果按空格分隔。

流程图

```
Start
  |
  |--> 输入 T
  |--> while T--:
  |         输入 n, m
  |         初始化 g 与 in
  |
  |         读入所有边 (u, v):
  |               g[u].push_back(v)
  |               in[v]++
  |
  |         将所有 in[i] == 0 的结点加入最小堆 q
  |
  |         while q 非空:
  |               c = q.top(); q.pop()
  |               对每个 next ∈ g[c]:
  |                     in[next]--
  |                     若 in[next] == 0:
  |                           q.push(next)
  |               rst.push_back(c)
  |
  |         输出 rst
  |
End
```

复杂度分析

1. **建图**
   - 插入所有 m 条边：O(m)
2. **拓扑排序**
   - 每个点最多一次入堆/出堆：O(n log n)
   - 每条边最多被访问一次：O(m)

总体时间复杂度：**O(n log n + m)**
总体空间复杂度：**O(n + m)**

细节注意

- 优先队列要用在遍历中的queue中，而不是构建邻接表时使用优先队列。具体思想是，在遍历中的queue中的节点，代表当前可以直接输出。所以应该找到优先级最高的输出，尽管他可能是后入队的。

具体实现

```
#include<iostream>
#include<vector>
#include<queue>
using namespace std;
int main(){
    int T;
    cin>>T;
    while(T--){
        int n,m;
        cin>>n>>m;
        vector<vector<int>> g(n+1);
        vector<int> in(n+1,0);
        priority_queue<int,vector<int>,greater<int>> q;
        vector<int> rst;
```

```
        for(int i=0;i<m;i++){
            int u,v;
            cin>>u>>v;
            g[u].push_back(v);
            in[v]++;
        }
        for(int i=1;i<n+1;i++){
            if(in[i]==0){
                q.push(i);
            }
        }
        while(!q.empty()){
            int c=q.top();
            q.pop();
            for(auto next:g[c]){
                in[next]--;
                if(in[next]==0){
                    q.push(next);
                }
            }
            rst.push_back(c);
        }
        for(int i=0;i<n;i++){
            cout<<rst[i]<<" ";
        }
        cout<<endl;
    }
    return 0;
}
```

4、程序运行与测试

测试样例

- 标准输入：

```
1
5 5
3 4
4 1
3 2
2 4
5 3
```

- 实际输出：

```
5 3 2 4 1
```

- 期望输出：

```
5 3 2 4 1
```

5、实验总结与心得

- 通过本题进一步掌握了拓扑排序的实际应用场景。
- 使用优先队列得到“字典序最小拓扑序”是一种常见且重要的技巧。
- 对大规模数据使用拓扑排序时，应特别注意复杂度控制与数据结构选择。
