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	2025.12.03	**结束日期**	2025.12.06

第一题

1、实验题目

⭐ z13-Can I Post the letter

题目描述

I am a traveler. I want to post a letter to Merlin. But because there are so many roads I can walk through, and maybe I can’t go to Merlin’s house following these roads, I must judge whether I can post the letter to Merlin before starting my travel. Suppose the cities are numbered from 0 to N − 1, I am at city 0, and Merlin is at city N − 1. And there are M roads I can walk through, each of which connects two cities. Please note that each road is direct, i.e. a road from A to B does not indicate a road from B to A. Please help me to find out whether I could go to Merlin’s house or not.

输入描述

There are multiple input cases. For one case, first are two lines of two integers N and M, (N <=

200, M <= N * N / 2), that means the number of citys and the number of roads. And Merlin stands at city N − 1. After that, there are M lines. Each line contains two integers i and j, what means that there is a road from city i to city j. The input is terminated by N = 0.

输出描述

For each test case, if I can post the letter print `I can post the letter` in one line, otherwise print `I can't post the letter`.

输入样例

```
3
2
0 1
1 2
3
1
0 1
0
```

输出样例

```
I can post the letter
I can't post the letter
```

2、实验目的

- 理解**有向图的基本概念和邻接表存储方式。**
- 掌握利用 **DFS** 搜索判断两点之间是否可达的思想。
- 训练对多组测试数据的输入与处理。

3、算法设计

设计思路

1. **读取输入并构建图**
   - 对于每个测试用例，读入 `n`（城市数）和 `m`（道路数）。
   - 使用大小为 `n` 的邻接表 `g` 存储图，对每条有向边 `u v`，在 `g[u]` 中加入 `v`。
   - 按题意城市编号为 `0..n-1`，起点为 `0`，终点为 `n-1`。
2. **深度优先搜索判断可达性**
   - 准备一个长度为 `n` 的布尔数组 `visited`，初始均为 `false`。
   - 从起点 `0` 调用 DFS，将所有可达节点标记为已访问。
   - DFS 过程：访问当前节点 `u` 时，遍历 `g[u]` 中的每个邻接点 `v`，若 `!visited[v]`，递归调用 DFS。
3. **根据访问结果输出答案**
   - DFS 完成后，若 `visited[n-1] == true`，说明可以从 `0` 到达 `n-1`，输出：
     - `I can post the letter`
   - 否则输出：
     - `I can't post the letter`
4. **多测试用例处理**
   - 按题意测试用例可能以 `N = 0` 结束，实际代码采用“读到 EOF 为止”的方式循环处理输入中的所有用例。

流程图

```
Start
  |
  |--> while 读入 n, m 成功:
  |        |
  |        |--> 创建 n 个空链表构成邻接表 g
  |        |--> 循环 m 次读入 u, v:
  |        |        g[u].push_back(v)
  |        |
  |        |--> visited[0..n-1] 全部置为 false
  |        |--> 从结点 0 调用 DFS(g, visited, 0)
  |        |
  |        |--> if visited[n-1] 为 true
  |        |          输出 "I can post the letter"
  |        |    else
  |        |          输出 "I can't post the letter"
  |
End
```

复杂度分析

1. **建图过程**
    - 邻接表初始化：O(n)。
    - 读入并加入所有边：每条边插入一次，总共 O(m)。
2. **DFS 搜索过程**
    - 每个结点最多被访问一次，每条有向边最多被遍历一次。
    - 因此一次 DFS 的时间复杂度为 O(n + m)。
3. **总体复杂度**
    - 对于一个测试用例，建图加 DFS 的时间复杂度为 **O(n + m)**。
    - 邻接表和访问数组的额外空间为 **O(n + m)**。

总时间复杂度： O(n + m)
总空间复杂度： O(n + m)

细节注意

- 在邻接表进行遍历时，考虑到**list**的特性，不能用遍历索引的方式进行，要for(auto a: g[u])的方式遍历。

具体实现

```
#include <iostream>
#include <vector>
#include <list>
using namespace std;
void dfs(vector<list<int>>& g, vector<bool>& visited, int u){
    visited[u]=true;
    for(int v: g[u]){
        if(!visited[v]){
            dfs(g,visited,v);
        }
    }
}
int main() {
    int n, m;
    while (cin >> n >> m) {
        vector<list<int>> g(n);
        for(int i=0;i<m;i++){
            int u,v;
            cin>>u>>v;
            g[u].push_back(v);
        }
        vector<bool> visited(n,false);
        dfs(g,visited,0);
        if(visited[n-1]){
            cout<<"I can post the letter"<<endl;
        }else{
            cout<<"I can't post the letter"<<endl;
        }
    }
}
```

4、程序运行与测试

测试样例

- 标准输入：

```
2
1
0 1
10
1
1 9
50
140
0 8
0 13
...
```

- 实际输出：

```
I can post the letter
I can't post the letter
I can post the letter
I can't post the letter
I can post the letter
I can post the letter
I can post the letter
```

- 期望输出：

```
I can post the letter
I can't post the letter
I can post the letter
I can't post the letter
I can post the letter
I can post the letter
I can post the letter
```

5、实验总结与心得

- 通过本题，熟悉了使用**邻接表表示图**，并在其上执行 DFS 搜索。
- 理解了“可达性判断”的本质就是**从起点进行一次遍历，看终点是否被访问到。**
- 对多组数据的处理流程更加熟练，为后续图论题目打下基础。

第二题

1、实验题目

⭐ z13-Compute Active time Interval during DFS

Problem

Given an undirected graph, for every vertex, compute the starting time and finishing time during a DFS.

Input

The first line is the number of test cases.

For every test case, the first line is the number of node n, meaning nodes are 1, 2, ..., n.

The next line is the number of edges m, then m lines are followed, where each line is in the form of `u v`, meaning (u, v) is an undirected edge.

Output

For each test case, print the starting and finishing time for every vertex, suppose we always starting a DFS from the smallest numbered vertex. For example, the output might look like

```
1:1-2
2:3-4
3:5-6
---
```

1. Suppose we always start a DFS from the smallest numbered vertex, and choose the smallest numbered adjacent vertex.
2. The starting time is 1.
3. Print a dash-line after each test case.

Sample Input

```
2

3
1
3 2

3
3
2 1
3 1
3 2
```

Sample Output

```
1:1-2
2:3-6
3:4-5
---
1:1-6
2:2-5
3:3-4
---
```

Note

1. You **must not** write only the main function.
2. You **must not** use global variables.
3. To match the output, make sure your adjacent list is **ordered**.

2、实验目的

- 理解 DFS 过程中“进入时间（start time）”与“离开时间（finish time）”的含义。
- 学会在遍历过程中通过计时器记录每个结点的时间区间。
- 体会按编号顺序、按邻接点序顺访问对输出结果的影响。

3、算法设计

设计思路

1. **图的读入与存储**
   - 输入测试用例个数 `num`。
   - 对每个测试用例，读入结点数 `n`、边数 `m`。
   - 建立 `n + 1` 大小的邻接表 `g`（从 1 开始编号），对**每条无向边** `(u, v)`：
     - **在 `g[u]` 中加入 `v`**；
     - **在 `g[v]` 中加入 `u`**。
   - 为了满足“优先访问编号较小的相邻结点”的要求，对每个 `g[i]` 调用 `sort`，按从小到

大排序。

2. **时间数组与 DFS 函数**
   - 准备两个数组：
     - `enter[i]`：结点 `i` 的**进入**时间（第一次被访问时的时间戳），初始值为 `-1`。
     - `left[i]`：结点 `i` 的**离开**时间（所有子结点访问完后的时间），初始值为 `-1`。
   - 维护一个整型变量 `Alltime` 作为全局时间计数器，通过引用参数传入 `dfs` 函数。
   - `dfs(g, enter, left, u, time)` 的逻辑：
     a. 将 `enter[u]` 设为当前的 `time` 值；
     b. 依次遍历 `g[u]` 中按升序排好的所有邻接结点 `next`；
     c. 若 `enter[next] == -1`（还未访问过），先自增 `time`，再递归调用 `dfs` 访问 `next`；
     d. 当前结点所有邻接点遍历结束后，自增 `time`，并将该值写入 `left[u]`。
3. **按编号顺序启动 DFS**
   - 为了保证从**最小编号**的未访问结点开始 DFS，在主程序中：
     - 从 `i = 1` 到 `n` 遍历所有结点；
     - 若 `left[i] == -1`，说明该结点尚未被访问，对其调用 `dfs`，并在调用前先 `++Alltime`。
   - 这样既保证了“从最小编号的结点开始”，也能正确处理图不连通的情况。
4. **输出结果**
   - 对当前测试用例，从 `i = 1` 到 `n` 依次输出：
     - `i:enter[i]-left[i]`
   - 每个测试用例结束后输出一行：
     - `---`

流程图

```
Start
  |
  |--> 读入测试用例个数 num
  |
  |--> 循环 num 次:
  |       读入 n, m
  |       建立邻接表 g[1..n]
  |       读入 m 条无向边 (u, v):
  |            g[u].push_back(v)
  |            g[v].push_back(u)
  |       对每个 i = 1..n:
  |            sort(g[i])  // 邻接点升序
  |
  |       初始化 enter[1..n] = -1, left[1..n] = -1
  |       Alltime = 0
  |
  |       for i = 1..n:
  |            if left[i] == -1:
  |                 ++Alltime
  |                 dfs(g, enter, left, i, Alltime)
  |
  |       for i = 1..n:
  |            输出 "i:enter[i]-left[i]"
  |       输出 "---"
  |
End
```

复杂度分析

1. **建图与排序**
    - 邻接表插入所有边：O(m)。
    - 对每个结点 `i` 的邻接表排序，复杂度约为$\sum_{i=1}^{n} \deg(i)\log\deg(i) \le m\log n$。
    - 因此建图与排序整体为 O(m log n)。
2. **DFS 过程**

   - 每个结点被访问一次，每条无向边被遍历两次（两个方向），总计 O(n + m)。
3. **输出过程**
   - 打印每个结点的时间区间：O(n)。
4. **总体复杂度**
   - 主导部分为排序 + DFS：**O(m log n + n + m) ≈ O(m log n + n)**。
   - 存储图和时间数组需要 **O(n + m)** 空间。

总时间复杂度： O(m log n + n)
总空间复杂度： O(n + m)

细节注意

- 关键在于用两个数组记录进入和退出时间。单纯递归的话很难实现，而且用两个数组也能起到判断节点是否被访问过的作用。
- 可以给函数传引用，相当于传一个全局变量来记录时间。

具体实现

```
#include<iostream>
#include<vector>
#include<algorithm>
using namespace std;
void dfs(vector<vector<int>>& g, vector<int>&enter, vector<int>& left, int u,int& time){
    enter[u]=time;
    for(int i=0;i<g[u].size();i++){
        int next=g[u][i];
        if(enter[next]==-1){
            dfs(g,enter,left,next,++time);
        }
    }
    left[u]=++time;
    // return seektime+2;
}
```

```
int main(){
    int num;
    cin>>num;
    while(num--){
        int n,m;
        cin>>n>>m;
        vector<vector<int>> g(n+1);
        for(int i=0;i<m;i++){
            int u,v;
            cin>>u>>v;
            g[u].push_back(v);
            g[v].push_back(u);
        }
        for(int i=1;i<n+1;i++){
            sort(g[i].begin(),g[i].end(),less<int>());
        }
        vector<int> enter(n+1,-1);
        vector<int> left(n+1,-1);
        int Alltime=0;
        for(int i=1;i<n+1;i++){
            if(left[i]==-1){
                dfs(g,enter,left,i,++Alltime);
            }
        }
        for(int i=1;i<n+1;i++){
            cout<<i<<":"<<enter[i]<<"-"<<left[i]<<endl;
        }
        cout<<"---"<<endl;
    }
    return 0;
}
```

4、程序运行与测试

测试样例一

- 标准输入：

```
5
3
1
3 1
3
1
3 2
...
```

- 实际输出：

```
1:1-4
2:5-6
3:2-3
---
1:1-2
2:3-6
3:4-5
---
1:1-6
2:2-5
3:3-4
---
1:1-6
2:2-3
3:4-5
---
1:1-2
2:3-4
3:5-6
---
```

- 期望输出：

```
（同上）
```

测试样例二

- 标准输入：

```
1
8
16
2 1
3 1
3 2
4 3
5 1
5 2
5 3
5 4
6 1
...
```

- 实际输出：

```
1:1-16
2:2-15
3:3-14
4:4-13
5:5-12
6:6-11
7:8-9
8:7-10
---
```

- 期望输出：

```
(同上)
```

5、实验总结与心得

- 通过本题理解了 DFS 中“时间戳”的意义，掌握了如何在遍历过程中记录每个结点的进入和离开时刻。
- 体会到“访问顺序”（按结点编号、邻接点排序）会直接影响输出结果。
- 对于多连通分量的图，也能通过从每个未访问的最小编号结点重新启动 DFS 来完整覆盖。

第三题

1、实验题目

⭐ z13-Connect components in undirected graph

题目描述

输入一个简单无向图，求出图中连通块的数目。

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 \leq n \leq 1000,\ 0 \leq m \leq 10000)。
以下 m 行，每行是一个数对 `v y`，表示存在边 ((v, y))。顶点编号从 1 开始。

输出描述

单独一行输出连通块的数目。

输入样例

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

输出样例

```
2
```

2、实验目的

- 复习无向图中“连通分量（连通块）”的概念。
- 掌握使用**并查集**（Disjoint Set Union / Union-Find）统计连通块数目。
- 理解路径压缩等优化对并查集效率的影响。

3、算法设计

设计思路

1. **初始化并查集**
   - 读入 `n` 和 `m`。
   - 创建数组 `root[0..n]`，将每个结点的父亲初始化为自身，即 `root[i] = i`。
   - 由于顶点编号从 1 开始，`root[0]` 可以保留不用。
2. **查找函数 `find`**
   - `find(root, v)` 返回结点 `v` 所在集合的代表元（根节点）。
   - 若 `root[v] == v`，则 `v` 为当前集合的根，返回 `v`；
   - 否则递归查找 `root[v]` 的根，并将 `root[v]` 直接压缩为该根，实现路径压缩：
     - `root[v] = find(root, root[v])`。
3. **合并所有边对应的集合**
   - 对每条边 `(v, y)`：
     - 分别计算 `rv = find(root, v)` 与 `ry = find(root, y)`；
     - 若 `rv != ry`，说明这两个结点原本属于不同连通块，将 `root[ry] = rv`，合并两个集合。
4. **统计连通块数量**
   - 合并完所有边后，再遍历 `i = 1..n`：
     - 若 **`root[i] == i`**，说明 `i` 是一个集合的代表元，即有一个连通块。

- 用计数器 `count` 统计这样的根的个数，输出 `count` 即为连通块数目。

流程图

```
Start
  |
  |--> 读入 n, m
  |--> 初始化 root[0..n]:
  |         for i = 0..n: root[i] = i
  |
  |--> 循环 m 次读入边 (v, y):
  |         rv = find(root, v)
  |         ry = find(root, y)
  |         if rv != ry:
  |               root[ry] = rv   // 合并集合
  |
  |--> count = 0
  |--> for i = 1..n:
  |         if root[i] == i:
  |               ++count
  |
  |--> 输出 count
  |
End
```

复杂度分析

1. **初始化并查集**
   - 设置 `root[i] = i`：O(n)。
2. **合并所有边**
   - 对于每条边执行两次 `find`，一次合并。
   - 采用路径压缩优化后，单次 `find` 的均摊时间复杂度接近常数，可记为 O(α(n))。
   - 因此处理所有边的总复杂度约为 O(m α(n))。
3. **统计根节点**

   - 遍历 `1..n` 检查 `root[i] == i`：O(n)。
4. **总体复杂度**
   - 时间复杂度为 O(n + m α(n))，其中 α(n) 为反 Ackermann 函数，增长极慢，近似常数。
   - 只需一个父亲数组 `root`，空间复杂度为 O(n)。

总时间复杂度： O(n + m α(n))
总空间复杂度： O(n)

细节注意

- 实际上还可以进行按秩合并，但是仅采用路径压缩页足够了。
- 也可以使用深搜或者广搜，但是并查集写法更省时间，时间复杂度也更小。

具体实现

```
#include<iostream>
#include<vector>
using namespace std;
int find(vector<int>& root, int v){
    return root[v]==v? v: root[v]=find(root, root[v]);
}
int main(){
    int n,m;
    cin>>n>>m;
    vector<int> root(n+1);
    for(int i=0;i<n+1;i++){
        root[i]=i;
    }
    for(int i=0;i<m;i++){
        int v,y;
        cin>>v>>y;
        int rv=find(root, v);
        int ry=find(root, y);
        if(rv!=ry){
            root[ry]=rv;
        }
    }
    int count=0;
    for(int i=1;i<n+1;i++){
        if(root[i]==i){
            count++;
        }
    }
    cout<<count<<endl;
}
```

4、程序运行与测试

测试样例一

- 标准输入：

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

- 实际输出：

```
2
```

- 期望输出：

```
2
```

测试样例二

- 标准输入：

```
12 13
1 3
1 6
1 7
2 3
2 4
2 5
2 7
3 5
```

```
3 6
4 5
5 6
8 9
12 9
```

- 实际输出：

```
4
```

- 期望输出：

```
4
```

5、实验总结与心得

- 本题通过并查集的视角重新理解了“连通块”的概念。
- 实际编码中体会到，路径压缩可以显著降低多次查找的开销。
- 进一步熟练掌握了并查集的三个核心操作：初始化、查找、合并，在今后解决最小生成树、动态连通性等问题时会非常有用。
