PAT_A1013. Battle Over Cities (25)

1.Abstraction

1.1 Algorithm and idea

```
1.图的遍历，深度优先遍历或者广度优先遍历都可以。
2.需要修建的路径条数实际上就是指联通的块的数目。
```

1.2 Notice

```
1.注意被“敌人占领的节点”实际上是指不可访问的节点。可以提前标记为已访问节点。
```

2.Problem: Battle Over Cities (25)

It is vitally important to have all the cities connected by highways in a war. If a city is occupied by the enemy, all the highways from/toward that city are closed. We must know immediately if we need to repair any other highways to keep the rest of the cities connected. Given the map of cities which have all the remaining highways marked, you are supposed to tell the number of highways need to be repaired, quickly.

For example, if we have 3 cities and 2 highways connecting city1-city2 and city1-city3. Then if city1 is occupied by the enemy, we must have 1 highway repaired, that is the highway city2-city3.

Input

Each input file contains one test case. Each case starts with a line containing 3 numbers N (<1000), M and K, which are the total number of cities, the number of remaining highways, and the number of cities to be checked, respectively. Then M lines follow, each describes a highway by 2 integers, which are the numbers of the cities the highway connects. The cities are numbered from 1 to N. Finally there is a line containing K numbers, which represent the cities we concern.

Output

For each of the K cities, output in a line the number of highways need to be repaired if that city is lost.

Sample Input

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

Sample Output

```
1
0
0
```

3.Algorithm note: 深度优先遍历

1.深度优先遍历就是以*深度*为首要搜索条件的搜索算法。

2.深度优先遍历可以使用*递归*或者*栈*的形式来实现。一般如果算法较为复杂可以考虑使用虚拟栈。

递归实现

```
//递归实现
void dfs(Node *original_graph,int point){
    hasVisited[point]=true;
    int n=original_graph[point].link.size();
    for(int i=0;i<n;++i){
        int j=original_graph[point].link[i];
        if(hasVisited[j]==false){
            dfs(original_graph,j);
        }
    }
}
```

虚拟栈实现

```
#include <stack>
using std::stack;
void dfs(Node *original_graph,int point){
    stack<int> s;
    int top;
    s.push(point);
    while(!s.empty()){
        top=s.top();
        s.pop();
        if(hasVisited[top]==false){
            hasVisited[top]=true;
            int n=original_graph[top].link.size();
            for(int i=0;i<n;++i){
                s.push(original_graph[point].link[i]);
            }
        }
    }
}
```

4. Code

4.1 Edit 0:

4.1.1 Algorithm abstraction

4.1.2 Notice

4.1.3 Code Block

```
#include <cstdio>
#include <cstring>
#include <vector>
#include <queue>

using std::vector;
using std::queue;
const int maxn=1010;

struct Node{
    vector<int> link;
};
Node origin_graph[maxn];
bool hasVisited[maxn];

void init_hasVisited(){
    memset(hasVisited,false,sizeof(hasVisited));
}
```

```
void dfs(Node (&new_graph)[maxn],int index,int delete_index){
    if(index==delete_index) return;
    for(int i=0;i<(int)new_graph[index].link.size();++i){
        if(hasVisited[new_graph[index].link[i]]==false){
            hasVisited[new_graph[index].link[i]]=true;
            dfs(new_graph,new_graph[index].link[i],delete_index);
        }
    }
}


int main(){
    int city_num,remaining_highways,check_num;
    int id1,id2;
    scanf("%d%d%d",&city_num,&remaining_highways,&check_num);
    for(int i=0;i<remaining_highways;++i){
        scanf("%d %d",&id1,&id2);
        origin_graph[id1].link.push_back(id2);
        origin_graph[id2].link.push_back(id1);
    }
    for(int i=0;i<check_num;++i){
        int num=0;
        int id;
        scanf("%d",&id);
        init_hasVisited();
        for(int j=1;j<=city_num;++j){
            if(hasVisited[j]==false&&j!=id){
                ++num;
                dfs(origin_graph,j,id);
            }
        }
        printf("%d\n",num-1);
    }
    return 0;
}
```

4.2 Edit 1:

4.2.1 Algorithm abstraction

4.2.2 Notice

4.2.3 Code Block

```
#include <vector>
#include <iostream>

using std::vector;

using std::cin;
```

```
using std::cout;
using std::endl;

const int maxn=1010;
struct Node{
    vector<int> link;
}original_graph[maxn];

bool hasVisited[maxn];
void init_hasVisited(){
    for(int i=0;i<maxn;++i){
        hasVisited[i]=false;
    }
}

void dfs(Node *original_graph,int point){
    hasVisited[point]=true;
    int n=original_graph[point].link.size();
    for(int i=0;i<n;++i){
        int j=original_graph[point].link[i];
        if(hasVisited[j]==false){
            dfs(original_graph,j);
        }
    }
}

int main(){
    int point_count,link_count,query_count;
    cin>>point_count>>link_count>>query_count;
    int point1,point2;
    for(int i=0;i<link_count;++i){
        cin>>point1>>point2;
        original_graph[point1].link.push_back(point2);
        original_graph[point2].link.push_back(point1);
    }
    for(int i=0;i<query_count;++i){
        cin>>point1;
        int num=0;
        init_hasVisited();
        hasVisited[point1]=true;
        for(int j=1;j<=point_count;++j){
            if(hasVisited[j]==false){
                dfs(original_graph,j);
                ++num;
            }
        }
        cout<<num-1<<endl;
    }

    return 0;
}
```

4.3 Edit 2:

4.3.1 Algorithm abstraction

4.3.2 Notice

4.3.3 Code block:

```
#include <vector>
#include <iostream>
#include <stack>
using std::stack;
using std::vector;
using std::cin;
using std::cout;
using std::endl;

const int maxn=1010;
struct Node{
    vector<int> link;
}original_graph[maxn];

bool hasVisited[maxn];
void init_hasVisited(){
    for(int i=0;i<maxn;++i){
        hasVisited[i]=false;
    }
}

void dfs(Node *original_graph,int point){
    stack<int> s;
    int top;
    s.push(point);
    while(!s.empty()){
        top=s.top();
        s.pop();
        if(hasVisited[top]==false){
            hasVisited[top]=true;
            int n=original_graph[top].link.size();
            for(int i=0;i<n;++i){
                s.push(original_graph[point].link[i]);
            }
        }
    }
}

int main(){
    int point_count,link_count,query_count;
    cin>>point_count>>link_count>>query_count;
    int point1,point2;
    for(int i=0;i<link_count;++i){
        cin>>point1>>point2;
        original_graph[point1].link.push_back(point2);
        original_graph[point2].link.push_back(point1);
    }
```

```
    for(int i=0;i<query_count;++i){
        cin>>point1;
        int num=0;
        init_hasVisited();
        hasVisited[point1]=true;
        for(int j=1;j<=point_count;++j){
            if(hasVisited[j]==false){
                dfs(original_graph,j);
                ++num;
            }
        }
        cout<<num-1<<endl;
    }

    return 0;
}
```

5. Summary

深度优先遍历算法的递归实现是常用算法，要能熟练掌握并且灵活使用。