1.0 Dijkstra算法

解决问题：

图中一个点到其它点的最短路距离。

算法思想简述：

主要是贪心策略，以及性质挖掘。

1. 当前轮中，找到与start距离最小的点。
2. 当前点的最短路已经计算完成。
3. 基于当前轮确定最短路的点，进行拓展。更新其它点的已知最短距离。
4. 开始新的一轮，直到所有点的最短路都确定。

注意细节

1. 要求图中不存在负边。
    1. 如果图中存在负边，且存在负环。那么不可能存在最短路。
    2. 从已知最短路更新出来的路不是单调递增的。此时不能从当前轮已知最短路距离简单的找出一个点并入已知最短路的集合中。

code

1. 对于矩阵的初始化：

如果边不存在，关于边权的存储那么两个节点之间的边权设为无穷大。

2. 对于d[i]已知最短距离。d[0] = 0; 其他d[i] = INF;

```
//const int oo = 0x0fffffff;//无穷大。一般自己的主程序模板自带。

int n;//边的编号。（小心变量重复。）

int g[N][N];//表示两个节点之间的边权。graph
bool v[N];  //表示节点当前最短路是否已知。

void dijkstra()
{
    fill(v , v + 1 + n, false);
    for(int i = 0; i < n; i++)d[i] = (i == 0 ? 0 : oo);
    for(int i = 0; i < n; i++)
    {
        int x, m = oo;
        for(int y = 0; j < n; j++)if(!v[y] && d[y] < m) m = d[x = y];
        v[x] = 1;
        for(int y = 0; j < n; j++)d[y] = min(d[y] , d[x] + g[x][y]);
    }
}
```

生长：

1. 打印路程的问题：

    使用dijkstra算法和动态规划中的方案一致。从终点出发， 不断顺着d[i] + w[i][j] == d[j]的边返回节点i.

    另外可以用空间换时间，开一个fa[x]来记录。（这里引入了一个松弛操作的概念来描述）。

```
if(d[y] > d[x] w[x][y]){
    d[y] = d[x] + w[x][y];
    fa[y] = x;
}
```

2. 复杂度分析：显然是$O(n^2)$.

复杂度优化（由$O(n^2)->o(m*log(n))$

优化角度：

1. 集中精力优化未标号节点中的最小d值。
2. 存储结构：邻接表表示图：

封装

```
//边结构的存储
struct Edge {
    int from, to, dist;
    Edge(int u, int v, int d) : from(u), to(v), dist(d) {}
};

const int oo = 1E9 + 10;
const int maxn = 2E5 + 10;//最大节点个数。
struct Dijkstra {
    int n, m;
    vector<Edge> edges;
    vector<int> G[maxn]; //保存了
    bool done[maxn];
    int d[maxn];         //s到各个点的距离
    int p[maxn];         //记录上一条弧

    void init(int n)
    {
        this->n = n;
        for (int i = 0; i < n; i++)G[i].clear();
        edges.clear();
    }

    void AddEdge(int from, int to, int dist)
    {
        edges.push_back(Edge(from, to, dist));
        m = edges.size();
        G[from].push_back(m - 1);
    }

    //优先队列使用的结构。当然可以使用pair来储存。
    struct HeapNode {
        int u;
        int d;
        //优先最小堆
        //这里怎么那么奇怪？发现一些问题。
        //抛开习惯。大于小于不过只是函数名称。
        bool operator <(const HeapNode& rhs)const {
            return d > rhs.d;
        }
    };

    void dijkstra(int s)
    {
        priority_queue<HeapNode>que;
        for (int i = 0; i < n; i++)d[i] = oo;
        fill(done, done + n, false);
        que.push({ s , 0 });
        d[s] = 0;
        while (que.empty() == false)
        {
            int u = que.top().u;que.pop();
            if (done[u])continue;
```

```
            done[u] = true;
            //检查所有边进行更新
            for (auto i : G[u])
            {
                int v = edges[i].to;
                if (d[v] > d[u] + edges[i].dist)
                {
                    d[v] = d[u] + edges[i].dist;
                    p[v] = i;
                    que.push({ v , d[v] });
                }
            }
        }
    }

}dij;

/*
* 1.不要在局部函数中定义对象。
* 2.注意数据范围。考虑将int，改为ll.
* 3.改模板节点管理：节点的下标从0开始。注意是否输入统一。
*/
```

测试问题：https://www.luogu.com.cn/problem/P4779

Bellman-ford算法

介绍

解决带负权的最短路问题：

原理

1. 认识到一个事实：如果最短路存在，一定存在一个不含环的最短路。
2. 不含环则最短路最多只经过n-1个节点，可以通过n - 1轮松弛操作得到。

有种动态规划的味道在里头。

在第k轮，对于d[i]记录的是只经历0...k-1个节点的所有路径中的最短路。

code初版

```
for(int i = 0; i < n; i++)d[i] = oo;
d[0] = 0;
for(int k = 0; k < n - 1; i++)
    for(int i = 0; i < m; i++)
    {
        int x = u[i] , y = v[i];
        if(d[x] < oo) d[y] = min(d[y] , d[x] + w[i]);
    }
```

封装模板

```
//边结构的存储
struct Edge {
    int from, to, dist;
    Edge(int u, int v, int d) : from(u), to(v), dist(d) {}
};

const int oo = 0x7fffffff;
const int maxn = 5E5 + 10;//最大节点个数。
```

```cpp
struct BellmanFord {
    int n, m;
    vector<Edge> edges;
    vector<int> G[maxn]; //保存了
    bool inq[maxn];
    int cnt[maxn];
    ll d[maxn];         //s到各个点的距离
    int p[maxn];         //记录上一条弧

    void init(int n)
    {
        this->n = n;
        for (int i = 0; i < n; i++)G[i].clear();
        edges.clear();
    }

    void AddEdge(int from, int to, int dist)
    {
        edges.push_back(Edge(from, to, dist));
        m = edges.size();
        G[from].push_back(m - 1);
    }

    bool bellman_ford(int s)
    {
        queue<int>que;
        fill(inq, inq + n, false);
        fill(cnt, cnt + n, 0);
        for (int i = 0; i < n; i++)d[i] = oo;
        d[s] = 0;
        inq[s] = true;
        que.push(s);
        while (que.empty() == false)
        {
            int u = que.front(); que.pop();
            inq[u] = false ;
            for (int i = 0; i < G[u].size(); i++)
            {
                Edge& e = edges[G[u][i]];
                if (d[u] < oo && d[e.to] > d[u] + e.dist)
                {
                    d[e.to] = d[u] + e.dist;
                    p[e.to] = G[u][i];
                    if (!inq[e.to]) { que.push(e.to); inq[e.to] = true; }
                    if (++cnt[e.to] > n) return  false;
                }
            }
        }
        return true;
    }
}bellman;

/*
* 1.不要在局部函数中定义对象。
* 2.注意数据范围。考虑将int，改为ll.
* 3.改模板节点管理：节点的下标从0开始。注意是否输入统一。
* 4.记得init();
*/
```

Floyd

解决问题：

处理多源最短路径问题；

原理

动态规划：

定义$d_{i,\ j}$表示只使用0...i的节点i,到j之间的最短距离。
状态转移方程为

$$d_{k,i,j} = min(d_{k-1,i,j}, d_{k-1,i,k} + d_{k-1,k,j}); \tag{1}$$

滚动数组优化后：

$$d_{i,j} = min(d_{i,k} + d_{k,j}) \tag{2}$$

证明它的可行性：

1. 最初始的状态下：初始化就是正确的。显然是正确的。
2. 第k轮的时候。

$d_{k,i,j} = min(d_{k-1,i,j}, d_{k-1,i,k} + d_{k-1,k,j})$

现有包含k的路径上。只需要关注i—>k，k->j的最短路。根据定义，假设使用了其它路径。替换任意一个都会更短。

code

```
/*
*初始化:  d[i][i] = 0;
*无边:   d[i][j] = oo;
*ij有边：d[i][j] = w[i][j]
*/
for(int k =0; k < n; k++)
    for(int  i = 0; i < n; i++)
        for(int j = 0; j < n; j++)
            d[i][j] = min(d[i][j], d[i][k]+d[k][j]);
```

多源最短路的理解问题：

E - Souvenir.md

最短路算法

kruskal算法。

这个算法比较简单。并且可以处理生成森林的问题。所以优先使用该算法。

对边进行一些排序，其副产品可以创造出一些价值：

算法步骤

1. 给所有边排序。
2. 从小到大，考查每条边。如果加上生成树中不成环，就加上去。

反证法。如果不加上去这条边，得到了一个最优的生成树（森林）。加上这条边之后一定会出现一个环。

然后在这个环中减去任何一条边从新变成森林。至少有一条边的权值大于等于（u , v）的权值。

code

```
const int N = 10010;
const int M = 1E6+10;
int p[N];
int id[M]; // 用来作为替身，定位大小；

int find(int x){return p[x] == x ? x : p[x] = find(p[x]);}

void unit(int x , int y)
{
    x = find (x);
    y = find (y);
    p[x] = y;
}

struct node{
    int u;
    int v;
    int w;
}e[M];

//当前已经处理好各条边；
ll kruskal()
{
    ll ans = 0;
    iota(p , p + n ,0);
    iota(id , id + m, 0);

    sort(id , id + m , [&](int x , int y){
        return e[x].w < e[y].w;
    });

    for(int i = 0; i < m; i++)
    {
        int now = id[i];
        int x = find(e[now].u);int y = find(e[now].v);
        if(x != y){ ans += e[now].w; p[x] = y;}
    }
    return ans;
}
```

入门问题：

1. 聪明的猴子：

https://www.luogu.com.cn/problem/P2504

一个性质。考察生成树的最大值的最小，出现在最小生成树中。

拓扑排序

- **解决问题**
  - 判断有向无环图是否成环
  - 对一个有向无环图，建立给每一个节点建立一个序。满足，若从a可达b。有$order(a) < or(b)$.可根据实现，自由定义，大于或者小于的问题。
- 算法实现
  - 从入度为0的节点开始递归。对于一个节点（假设有父节点），要想经过该节点，必须经过其父节点，开始父节点的线程时，就已经把最高可分配序分配给父节点。保证定义中的关系。

- 另外一种巧妙的算法：
  - 从0到n节点开始dfs
  - 对于一个点，只有把它的所有子节点分配序之后，再进行分配。保证$or(father) > or(son)$
  - 给每一个节点分好一个序之后。$reverse(or.brgin, or.end)$。这样保证了$or(father) < or(son)$.

算法一的实现：

```
1
```

算法二的实现

```
vector<int> order;
vector<vector<int>> g;
bool passed[maxn];
int pos[maxn];

void dfs(int u){
    passed[u] = true;
    for (auto i : g[u])
        if (!passed[i])
            dfs(i);
    order.push_back(u);
}

void topo()
{
    for(int i=0;i<n;i++)
    if(!passed[i])dfs(i);
    reverse(order.begin(), order.end()); //倒转。
}
```

生长思考

1. 另外一种方法：
    1. 每一次不断的选取度数为0的节点：从图中删除一个节点，并且更新其它节点的度。这样不断地迭代，直到把所有地点求出来。

通过这个角度，更新是广度的，可以的关注图中的更多结构细节。

code

```
    vector<int> topo;
    auto topo_sort = [&]()->void{
        queue<int> que;
        for (int i = 0; i < n; i++)if (deg[i] == 0) {
                que.push(i);
            }
        while (que.empty() == false) {
            int u = que.front(); que.pop();
            topo.push_back(u);
            for (auto v : g[u]) {
                deg[v]--;
                if (deg[v] == 0)que.push(v);
            }
        }
    };
    topo_sort();
    reverse(topo.begin() , topo.end());
```

st. E - Find Permutation .md

建好图之后：（假设用了vector来表示邻接表。）：

转化成有根树，其实就是明确相邻节点之间地父子关系。

```
int p[N];
void dfs(int now , int fa)
{
    int d = G[u].size();
    for(int i = 0; i < d; i++ )
    {
        int v = G[u][i];
        if(v != fa) dfs(v, p[v] = u);
    }
}
```

树哈希

解决问题：

1. 快速判断树是否同构的问题：
    1. 同构的概念：对一棵树， 进行对同父亲的子树进行互换。进行若干次操作后，两颗树可以相等。那么称这棵树是同构的。

参考：一种好写且卡不掉的树哈希 - 博客 - peehs_moorhsum的博客 (uoj.ac)

算法简介

关注树结构的属性： 定义一个哈希函数。
科学的定义一个哈希函数。

1. 根哈希函数，代表了当前子树的结构情况。
2. 哈希函数， 和子树的哈希函数联系。
   最终就是要降不同的子树结构不会落入同一个哈希值的概率。
   这里直接找一些大佬的哈希函数设计。

入门问题

1. (https://codeforces.com/contest/1800/problem/G)

板子

邓老师

```
using ll = long long;
using Ull = unsigned long long;
mt19937_64 rnd(chrono::steady_clock::now().time_since_epoch().count());
Ull bas = rnd();
//Ull bas = (Ull)1E18 + 7;
void work(int testNo)
{
    int n;  cin >> n;
    vector<vector<int>> e(n);
    for (int i = 1; i < n; i++) {
        int u, v; cin >> u >> v;
        u--; v--;
        e[u].push_back(v);
        e[v].push_back(u);
```

```
    }

    vector<Ull> h(n), f(n);
    function<Ull(Ull)> H = [&](Ull x) {
        return x * x * x * 19890535 + 19260817;
    };

    function<Ull(Ull)> F = [&](Ull x) {
        return H(x & (((1ll << 32) - 1))) + H(x >> 32);
    };

    function<void(int, int)> dfs = [&](int u, int par) {
        h[u] = bas;
//
        for (auto v : e[u]) if (v != par) {
                dfs(v, u);
                h[u] += F(h[v]);
//              rec[h[v]].push_back(v);
            }
        //cout << "no is " << u << " " << rec.size() << '\n';
        //通过节点的hash值情况进行一些哈希。
    };
}
```

树的存储

- 关于树的几个关键概念。
  - 树根。
  - 儿子
  - 树的边数（等于节点树减一）
  - 子树。
- 树的存储。
  - 邻接矩阵。
  - 邻接表。

着重处理邻接表的储存管理问题。

$first$ ——利用stl中的$vector$来实现邻接表。

```
#include<vector>
//声明。
vector<vector<type>> g;
//初始化。
g.resize(n+1);
//g[i].存储了和节点i相关的边。里面的元素，和i节点有连接的关系，
```

$second$ ——自己建立一个数据结构邻接表。相关行为要简洁，常数小。

- 一般的应用背景
  - 给出总边数。节点数知道了，总边数自然就知道了。
  - 某个节点的儿子数不知道。
- 最简单的写法。
  - 直接定义一个指针数组。数组上的每一个指针都代表着，管理相关节点的儿子族的头指针。也就是代表了一个链表。
  - 优化方法，再次定义一个数组，来存储每一链表的尾部地址。可以实现$O(1)$的插入。
  - 缺点：

- ~~不够装逼。~~
- *new*运算，常数贡献比较大。

- 其它写法，完全不沾边指针的概念。
  - *tips*
    - 用一个数组来存储节点。节点的地址，可以是当前的索引。该种关注角度也可以完成一个节点的元素的寻址问题。
    - 采用头部接入，就可以不关注尾部。
      （事实上，笔者总是忽略从头部接入。反过来想一想，可以发是否可行。可以有突破性的发现。逆向思维。）

code

```
const int maxn=1e6+10;
struct edge
{
    int t, next;
} e[maxn << 1];
int tot, head[maxn];
void add(int x, int y)
{
    e[++tot].t = y;
    e[tot].next = head[x];
    head[x] = tot;
}
/////定义方式。
//使用细节
//遍历
//fa: now的父节点。
//now:节点。现在的任务是遍历当前的节点
for(int i=head[x];i;i=e[i].next)
if(i!=fa)//表示的当前就是子节点。
{
    //stament
}
```

problem

给定一个表达式， 建立一个表达式图。

code

```
const int maxn = 1000;
int lch[maxn] , rch[maxn];char op[maxn];
int nc;

//返回建立的子表达式的树根
int build_tree(char* ss, int x, int y)
{
    int i, c1 = -1, c2 = -1 , p = 0;
    int u;
    if(y - x == 1){
        u = ++nc;
        lch[u] = rch[u] = 0; op[u] = s[x];
        return u;
    }
```

```
15
16      for(int  i = x; i < y; i++){
17          switch(s[i]){
18              case '(': p++; break;
19              case ')': p--; break;
20              case '+': case '-': if(!p) c1 = i; break;
21              case '*': case '/': if(!p) c2 = i; break;
22          }
23      }
24
25      if(c1 < 0) c1 = c2;
26      if(c1 < 0) return build_tree(s , x + 1 , y - 1);
27      u = ++nc;
28      lch[u] = build_tree(s, x, c1);
29      rch[u] = build_tree(s, c1 + 1, y);
30      op[u] = s[c1];
31      return u;
32  }
```

回顾tips

这个程序讨论了表达式子的特种情况。利用p，c1 ,c2收集了相关的信息。利用正确的方式处理了所有的情况。

- p: 遍历过程中。记录了2前面的阔后出现的情况。
- c1记录括号外的加减号的位置。
- c2记录括号外的乘除号的位置。

综上：思路就是寻找最后计算的运算符号。

如果有多个根据计算机程序计算的结合性就是计算最右边的符号。所以扫描了一遍就能满足了需求。

1. 实现能力：

在扫描过程中记录关键信息， 处理信息；就是扫描线思想。在扫描一遍的过程中，我可以什么信息处理。认识结构?

例题

公共表达式消除

该问题的解决涉及了树哈希，以及表达式建树等问题。

这里主要解决公共表达式建树问题：

abstract

DFS 树上的边

1. Tree Edge(树边)
   就是树上的边
2. Back Edge(返祖边)
   子孙连向祖先的边。（不作为树边，是引入的边。）
3. Forward Edge (前向边)
   就是祖先直接到了子孙上的边，忽略了中间的一些点。
4. Cross Edges（横插边）
   除了1 2 3 的边

细节，边的形成和dfs方式有关。dfs树并不唯一确定。基于问题来研究。可能这不同的方式对解决问题有相同的效力。

scc

弱连通分量: (基于无向图)

强连通分量: (strongly connected component) (基于有相图)
u存在一条路径到达v .v存在一条路径到达u 则称u与v是强连通的.

- 性质:
    1. u->v , v->u.有环.相关环

把图分成若干块(分量) , 块之间的节点是强连通的.

求图的强连通分量:

(几级)理解证明 -- 大致观念 -- 黑盒

$first$ Tarjan 算法 (塔洋算法)

关注现象以及方法

1. dfs过程中,如果找到了返祖边,前向边横插边.可以得到和对强连通分量相关的什么信息?
2. 处理方案.如果发现一个块不可能会成为其它强连通分量的一员,就及时切掉.

总结实现

1.存储图:(一般除了网络流使用链表存储.其它的直接用vector即可.)
注意建图基于单向边.

```
vector<vector<int>> g;
void solve()
{
    int n , m;
    cin >> n >> m;
    for(int i = 1 ; i <= m; i ++)
    {
        int u , v;
        cin >> u >> v;
        g[u].push_back(v);
    }
}
```

2. 使用一些结构,遍历过程中关注点的一些属性:

1. `stack<int> stk` : 用来保存遍历dfs过程中依次经过的节点。
2. `int dfn[N]` : dfs树。节点的dfs序为当前第几个被遍历到。
3. `vector<vector<int>> ssc` 每一个vector保存连通分量 。
4. `int low[N]` 记录每一个节点的可达dfn最低点。
5. `ins[N]` 记录节点是否在栈中。

在遍历过程中，可能碰到一些边返祖（不返组就只能作为儿子。）一旦返祖，就会形成一个环。与该环上某一个的节点强连通的节点。都会成为与环上的点强连通。

1. 用low的第一个记录返祖边信息
   那么怎么将这种环套环的节点整理放在一块呢？在dfs的过程中，如果一个点没有边返祖，并且已经拓展到了最大深度，那么这个点就作为连通分量。直接将它并入ssc即可。于是
2. 利用low记录圈上最顶层的点。
   考察多种情况:
   当遍历在回到这些点时，stk的top一大段连续的部分，都是强连通块上的点。

板题

强连通分量 - 题目 - Daimayuan Online Judge

板code

```cpp
#include<bits/stdc++.h>
using namespace std;
using ll = long long;

const int N = 1E6 + 10;

vector<vector<int>>g;
int dfn[N] , low[N] , bel[N], ins[N] , id;

vector<vector<int>> ssc;
stack<int>stk;

void dfs(int u) {

    dfn[u] = low[u] = ++id;
    ins[u] = true;
    stk.push(u);

    for (auto v : g[u]) {
        if (!dfn[v]) dfs(v);
        if (ins[v]) low[u] = min(low[v] , low[u]);
    }
    //说明已经没有收割的空间了：
    if (low[u] == dfn[u])
    {
        vector<int> s;
        while (true)
        {
            int v = stk.top();
            stk.pop();
            s.push_back(v);
            ins[v] = false;
            bel[v] = ssc.size();
            if (v == u)break;
        }
        sort(s.begin(), s.end());
        ssc.push_back(s);
    }
}

int main()
{
    ios::sync_with_stdio(false);
    cin.tie(0);

    int n , m;
    cin >> n >> m;
    g.resize(n + 1);
    for (int i = 0; i < m; i ++)
    {
        int u , v;
        cin >> u >> v;
        g[u].push_back(v);
    }

    for (int  i = 1; i <= n; i++) {
        if (dfn[i] == 0)
            dfs(i);
```

```
    }
    sort(ssc.begin(), ssc.end());

    for (auto i : ssc) {
        for (auto j : i) {
            cout << j << ' ';
        }
        cout << '\n';
    }
}

/* stuff you should look for
* int overflow, array bounds
* special cases (n=1?)
* do smth instead of nothing and stay organized
* WRITE STUFF DOWN
* DON'T GET STUCK ON ONE APPROACH
*/
```

kosaraju 算法

关注现象

1. 关注dfs出栈的序：对于DAG，出栈顺序是反图的拓扑序。
2. 把scc缩成一个点后，图变成一个有向无环图。
3. 最后一个出栈的点，其所在的连通分量，在缩完之后的树中是个源点（没有入度）。
    - 证明：反证法，假设搜了一次缩出来的一个图中。u所在的连通块有入度有入度，那么无论是从其它点开始搜还是从所在的块中的点开始搜，都是矛盾的。

综合以及方法

于是可以发现所有可以到达u(最后一个出栈的点)的点都和u是同一个连通分量的。利用反图。可以找出所有的点。于是就找到u所在的连通分量之中。
每确定一个点都进行一次dfs，就可以逐步把所有的强连通分量确定。

实现

1. 只需要一遍dfs即可，对dfs栈重利用。最后的复杂度也是$O(n)$。最丑的实现是，每次确定一个点都打一次标记。
    1. 对于一个图而言，无论从哪个根开始遍历最后的结果不会变。（反证法，我们首先确定了每种根的方式下开始遍历，最后得到的正确结果。于是两种结果之间一一映射，否则矛盾。）
    2. 于是第一次得到的栈中，去掉处理后的源点。剩下的顺序也是从某个点深搜的dfs出栈序。

板code

```
#include<bits/stdc++.h>
using namespace std;
using ll = long long;

const int N = 1E6 + 10;

vector<vector<int>> e , erev;

vector<vector<int>> ssc;
vector<int> c;
bool vis[N];

vector<int>out;


void dfs(int u)
{
```

```
    vis[u] = true;
    for (auto v : e[u]) {
        if (vis[v])continue;
        dfs(v);
    }
    out.push_back(u);
}

void dfs2(int u) {
    vis[u] = true;
    for (auto v : erev[u]) {
        if (vis[v])continue;
        dfs2(v);
    }
    c.push_back(u);
}

int main()
{
    ios::sync_with_stdio(false);
    cin.tie(0);

    int n , m;
    cin >> n >> m;
    e.resize(n + 1);
    erev.resize(n + 1);
    for (int i = 0; i < m; i++) {
        int x, y;
        cin >> x >> y;
        e[x].push_back(y);
        erev[y].push_back(x);
    }

    for (int i = 1; i <= n; i ++)
    {
        if (vis[i])continue;
        dfs(i);
    }
    memset(vis, false, sizeof(vis));
    reverse(out.begin(), out.end());
    for (auto u : out) {
        if (vis[u])continue;
        c.clear();
        dfs2(u);
        sort(c.begin(), c.end());
        ssc.push_back(c);
    }
    sort(ssc.begin(), ssc.end());
    for (auto i : ssc) {
        for (auto j : i)
        {
            cout << j << ' ';
        }
        cout << '\n';
    }
}
```

解决这些问题，最重要的是转换的思想，关注解结构中强连通分量的情况。从而将问题转换成dag上的dp问题。

例题1.受欢饮的牛

[HAOI2006, 受欢迎的牛 - 题目 - Daimayuan Online Judge]

找出有向图中所有点可达的汇点。

solve

尝试从强连通性的角度来看待问题。

1. 对于一个已经缩点的图中，缩点内部任意可达。
2. 如果存在一个汇点，汇点都将在一个强连通分量中。
    1. 对于两个汇点 u，v。作为汇点说明它们双向可达。
3. 汇点所在的缩点中，没有出边。

综上：
如果缩完点之后，找出每一缩点的出度为0的缩点数之和。
如果图中存在两个以上出度为0的缩点。
说明不存在汇点。

code

```cpp
#include<bits/stdc++.h>
using namespace std;
using ll = long long;

const int N = 1E6 + 10;

vector<vector<int>>g;
int dfn[N] , low[N] , bel[N], ins[N] , id;
vector<vector<int>> ssc;
stack<int>stk;
int n , m;
int cnt[N];//记录每一个分量的入度出度个数。

void dfs(int u) {

    dfn[u] = low[u] = ++id;
    ins[u] = true;
    stk.push(u);

    for (auto v : g[u]) {
        if (!dfn[v]) dfs(v);
        if (ins[v]) low[u] = min(low[v] , low[u]);
    }

    //说明已经没有收割的空间了：
    if (low[u] == dfn[u]) {
        vector<int> s;
        while (true) {
            int v = stk.top();
            stk.pop();
            s.push_back(v);
            ins[v] = false;
            bel[v] = ssc.size();
            if (v == u)break;
        }
        sort(s.begin(), s.end());
        ssc.push_back(s);
    }
}

void tarjan(){
    for (int  i = 1; i <= n; i++) {
```

```
        if (dfn[i] == 0)
            dfs(i);
    }
    sort(ssc.begin(), ssc.end());
    for (int u = 1; u <= n; u++) {
        for (auto v : g[u]) {
            if (bel[u] != bel[v]) {
                cnt[bel[u]]++;
            }
        }
    }
    int ans = 0;
    int tot = 0;
    for (int i = 0; i < ssc.size(); i++ ){
        if (cnt[i] == 0) {
            tot ++;
            ans += ssc[i].size();
        }
    }
    if (tot > 1)
        cout << 0 << '\n';
    else cout << ans << '\n';
}
int main()
{
    ios::sync_with_stdio(false);
    cin.tie(0);
    cin >> n >> m;
    g.resize(n + 1);
    for (int i = 1; i <= m; i ++){
        int x, y;
        cin >> x >> y;
        g[x].push_back(y);
    }
    tarjan();
}

/* stuff you should look for
* int overflow, array bounds
* special cases (n=1?)
* do smth instead of nothing and stay organized
* WRITE STUFF DOWN
* DON'T GET STUCK ON ONE APPROACH
*/
```

例题二 最大半连通子图 - 洛谷

[P2272 ZJOI2007]最大半连通子图 - 洛谷 | 计算机科学教育新生态 (luogu.com.cn)

$solve$

1. 该问题关注了点之间的可达性。一些点之间是强连通的，那么必然在最优方案一起出现。于是考虑对他们进行缩点，将连通分量作为一个点研究。
   于是问题转换成了，在有点权的dag中求一条路使得路上的点权和最大，并且求出最优方案的方案数。
2. 转换得到的问题是dag上的一个dp问题，

   对于缩点后的图。
   f_i：从为起点向下的路的最大点权和。
   w_i：满足最大点权和的方案总数。

$$f_i = max(f_{sons}) + size[i]; \tag{3}$$

$$w_i = \sum(f_{son} == max(f_{sons})) * w_i. \tag{4}$$

```
这里的方案不同，只能由点决定。所以要注意去除重边。
```

3. 状态转移的顺序上，不用再找起点再进行遍历。
    1. ssg得到的编号数组，就是dag上拓扑序的反序。
    2. tarjan的过程中，也进行状态转移。
4. 关于实现细节上

有两种风格的实现方式，其中。

1. 第一种是dfs完之后再dp。

   因为求出强连通分量之后就得到了拓扑序的反序。

2. 第二种是一边dfs再一边dp

   tarjan算法在求出一个缩点之后，深度更大的缩点就被求出来了。意味着，小规模子问题，以及缩点的信息都已经求出。所以可以dp.

code first

```cpp
#include<bits/stdc++.h>
using namespace std;
using ll = long long;


const int N = 1E6 + 10;


int f[N], w[N], vis[N];

int n, m, mod;

vector<vector<int>>e;
int dfn[N], low[N], bel[N], ins[N], id;

vector<vector<int>> ssc;
stack<int>stk;

void dfs(int u) {

    dfn[u] = low[u] = ++id;
    ins[u] = true;
    stk.push(u);

    for (auto v : e[u]) {
        if (!dfn[v]) dfs(v);
        if (ins[v]) low[u] = min(low[v], low[u]);
    }

    //说明已经没有收割的空间了：
    if (low[u] == dfn[u]) {
        vector<int> s;
        while (true) {
            int v = stk.top();
            stk.pop();
            s.push_back(v);
            ins[v] = false;
            bel[v] = ssc.size();
            if (v == u)break;
        }
        //sort(s.begin(), s.end());
```

```
        ssc.push_back(s);
    }
}

void tarjan()
{
    for (int i = 1; i <= n; i++) {
        if (dfn[i] == 0)
            dfs(i);
    }
    int T = 0, ans = 0, we = 0;
    for (int i = 0; i < ssc.size(); i++) {
        ++T;
        f[i] = 0;
        w[i] = 1;
        vis[i] = T;
        for (auto u : ssc[i]) {
            for (auto v : e[u]) {
                if (vis[bel[v]] == T)continue;

                vis[bel[v]] = T;
                if (f[i] < f[bel[v]]) {
                    f[i] = f[bel[v]];
                    w[i] =  w[bel[v]];
                }
                else if (f[i] == f[bel[v]]) {
                    w[i] = (w[i] + w[bel[v]]) % mod;
                }
            }
        }
        f[i] += ssc[i].size();
        if (f[i] > ans)  ans = f[i], we = 0;
        if (f[i] == ans) we = (we + w[i]) % mod;
    }
    cout << ans << '\n';
    cout << we << '\n';
}

int main()
{
    ios::sync_with_stdio(false);
    cin.tie(0);
    cin >> n >> m >> mod;
    e.resize(n + 1);
    for (int i = 1; i <= m; i++) {
        int x, y;
        cin >> x >> y;
        e[x].push_back(y);
    }
    tarjan();
}

/* stuff you should look for
* int overflow, array bounds
* special cases (n=1?)
* do smth instead of nothing and stay organized
* WRITE STUFF DOWN
* DON'T GET STUCK ON ONE APPROACH
*/
```

code_second

```cpp
#include<bits/stdc++.h>
using namespace std;
using ll = long long;


const int N = 1E6 + 10;


int f[N], sum[N], vis[N];

int n, m, mod;

vector<vector<int>>e;
int dfn[N], low[N], bel[N], ins[N], id;

vector<vector<int>> ssc;
stack<int>stk;

int ans , way , T;

void dfs(int u) {

    dfn[u] = low[u] = ++id;
    ins[u] = true;
    stk.push(u);
    for (auto v : e[u]) {
        if (!dfn[v]) dfs(v);
        if (ins[v]) low[u] = min(low[v], low[u]);
    }
    //说明已经没有收割的空间了：
    if (low[u] == dfn[u]) {
        ++T;
        int now = ssc.size();
        f[now] = 0 , sum[now] = 1;
        vector<int> s;
        vis[now] = T;
        while (true) {
            int v = stk.top();
            stk.pop();
            s.push_back(v);
            ins[v] = false;
            bel[v] = now;
            for (auto to : e[v]) {
                if (ins[to] == false && vis[bel[to]] != T)
                {
                    vis[bel[to]] = T;
                    if (f[bel[to]] > f[now])
                        f[now] = f[bel[to]], sum[now] = 0;
                    if (f[bel[to]] == f[now])
                        sum[now] = (sum[now] + sum[bel[to]]) % mod;
                }

            }

            if (v == u)break;
        }
        f[now] += s.size();
        //sort(s.begin(), s.end());
        if (ans < f[now]) ans = f[now] , way = 0;
        if (ans == f[now]) way = (way + sum[now]) % mod;
        ssc.push_back(s);
```

```
62
63      }
64  }
65
66  void tarjan()
67  {
68      for (int i = 1; i <= n; i++) {
69          if (dfn[i] == 0)
70              dfs(i);
71      }
72
73      cout << ans << '\n';
74      cout << way << '\n';
75  }
76
77  int main()
78  {
79      ios::sync_with_stdio(false);
80      cin.tie(0);
81      cin >> n >> m >> mod;
82      e.resize(n + 1);
83      for (int i = 1; i <= m; i++) {
84          int x, y;
85          cin >> x >> y;
86          e[x].push_back(y);
87      }
88      tarjan();
89  }
90
91  /* stuff you should look for
92  * int overflow, array bounds
93  * special cases (n=1?)
94  * do smth instead of nothing and stay organized
95  * WRITE STUFF DOWN
96  * DON'T GET STUCK ON ONE APPROACH
97  */
```

生长思考

1. 总是忽视在收缩后的图上进行处理。
2. 判断重边，这里使用了T变量的技巧。每一轮，对访问过的点，都复一个当前轮独特的标记。就是跟随迭代的变量T

例题三 ATM

APIO2009, ATM - 题目 - Daimayuan Online Judge

简介

给定有向图，求出一条单向路径（可以重复经过同一个点）规定终点为$x_1....x_n$中的一点。

求出点权和最大的方案。

thinking

1. 考察众多解，最优策略总是将强连通量内的点作为整体考虑。于是利用缩点，就见将问题转换成了一个dag上的dp问题。
2. 集中精力解决dp问题。

dp：表示从该点单向到达某一个酒吧的最短距离。

对于最小规模的子问题，如果点（缩点）内部没有酒吧。那么dp = -oo

如果本身就是酒吧，就设置为 val(该路口上的银行的钱。)

2. **1 怎么统一转移?**

初始化 dp = -oo;

先考察儿子的方案。$dp = max(dp, dp[son].)$

最后再考察自己作为终点的方案即

$dp = max(dp, 0)$

code

```cpp
#include<bits/stdc++.h>
using namespace std;
using ll = long long;


const int N = 1E6 + 10;
const int inf = 0x0ffffff;


ll dp[N], val[N];

int n, m, s, p;
bool bar[N];

vector<vector<int>>e;
int dfn[N], low[N], bel[N], ins[N], id, now;

stack<int>stk;

void dfs(int u) {

    dfn[u] = low[u] = ++id;
    ins[u] = true;
    stk.push(u);

    for (auto v : e[u]) {
        if (!dfn[v]) dfs(v);
        if (ins[v]) low[u] = min(low[v], low[u]);
    }

    //说明已经没有收割的空间了:
    if (low[u] == dfn[u]) {
        vector<int> s;

        ll sum = 0;
        ++now;
        dp[now] = -inf;
        bool havebar = false;

        while (true) {
            int v = stk.top();
            stk.pop();
            s.push_back(v);
            ins[v] = false;
            bel[v] = now;

            sum += val[v];
            havebar |= bar[v];
            for (auto w : e[v]) {
                if (ins[w] == false && bel[w] != now)
                    dp[now] = max(dp[now], dp[bel[w]]);
            }
            if (v == u)break;
```

```
        }
        //sort(s.begin(), s.end());

        if (havebar) {
            dp[now] = max(0LL, dp[now]);
        }
        dp[now] += sum;
    }
}

void tarjan()
{
    for (int i = 1; i <= n; i++) {
        if (dfn[i] == 0)
            dfs(i);
    }
    cout << dp[bel[s]] << '\n';
}

int main()
{
    ios::sync_with_stdio(false);
    cin.tie(0);
    cin >> n >> m;
    e.resize(n + 1);
    for (int i = 1; i <= m; i++) {
        int x, y;
        cin >> x >> y;
        e[x].push_back(y);
    }
    for (int i = 1; i <= n; i++)cin >> val[i];
    cin >> s >> p;
    for (int i = 1; i <= p; i++) {
        int x;
        cin >> x;
        bar[x] = true;
    }

    tarjan();
}

/* stuff you should look for
* int overflow, array bounds
* special cases (n=1?)
* do smth instead of nothing and stay organized
* WRITE STUFF DOWN
* DON'T GET STUCK ON ONE APPROACH
*/
```

例题四 所驼门王的宝藏

[P2403 SDOI2010]所驼门王的宝藏 - 洛谷 | 计算机科学教育新生态 (luogu.com.cn)

依然是转换称dag然后求取最长路的问题。

solve

建图技巧：

1. 第一种联系，引入一个辅助点。这样用来减少边的个数，并且减少建边的花费。同一行，并且是第一种的点可以利用这个辅助点，实现资源共享，从而增加空间的利用率。
2. 第二种联系，同上，为每一列之间建立一个辅助点。

3. 第三种，由于最多7个，所以搜索建立边即可。

于是建立了一个原图点之间连通性等效的图。

dp

比较简单的dp类型。

但是注意统计的时候，要去除自己建立的辅助点的影响。

code

```cpp
#include<bits/stdc++.h>
using namespace std;
using ll = long long;

const int N = 1E6 + 10;
int x[N], y[N], ty[N], cunt;

int ans = 0;


map<pair<int, int>, int> rec;
int idx[N], idy[N];
vector<vector<int>>e;

vector<int> c[N] , r[N];

stack<int> stk;
int bel[N * 2 + 10], low[N * 2 + 10], dfn[N * 2 + 10], ins[N * 2 + 10], cuntssc, id;

int dp[N * 2 + 10];
int n, R, C;

void dfs(int u) {
    dfn[u] = low[u] = ++id;
    ins[u] = true;
    stk.push(u);
    for (auto v : e[u]) {
        if (dfn[v] == 0) dfs(v);
        if (ins[v]) low[u] = min(low[u], low[v]);
    }

    if (low[u] == dfn[u]) {
        cuntssc++;
        dp[cuntssc] = 0;
        int sum = 0;
        while (true) {
            int v = stk.top(); stk.pop();
            sum += (v <= n);
            bel[v] = cuntssc;
            ins[v] = false;
            for (auto w : e[v]) {
                if (ins[w] || bel[w] == cunt)continue;
                dp[cuntssc] = max(dp[bel[w]], dp[cuntssc]);
            }
            if (u == v) break;
        }
        dp[cuntssc] += sum;
        ans = max(dp[cuntssc], ans);
    }

}


```

```cpp
void tarjan() {
    for (int i = 1; i <= cunt; i++) {
        if (dfn[i] == 0) dfs(i);
    }
    cout << ans << '\n';
}


int main()
{
    ios::sync_with_stdio(false);
    cin.tie(0);
    cin >> n >> R >> C;
    for (int i = 1; i <= n; i++)
    {
        cin >> x[i] >> y[i] >> ty[i];
        rec.insert({ {x[i] , y[i]}, i });
        r[x[i]].push_back(i);
        c[y[i]].push_back(i);
    }
    cunt = n;

    e.resize(n * 2 + 10);
    for (int i = 1; i <= n; i++)
        if (ty[i] == 1) {
            int now = idx[x[i]];
            if (now == 0) {
                idx[x[i]] = now = ++cunt;
                for (auto v : r[x[i]])
                    e[cunt].push_back(v);
            }
            e[i].push_back(now);
        }
        else if (ty[i] == 2) {
            int now = idy[y[i]];
            if (now == 0) {
                idy[y[i]] = now = ++cunt;
                for (auto v : c[y[i]])
                    e[cunt].push_back(v);
            }
            e[i].push_back(now);
        }
        else {
            for (int dx = -1; dx <= 1; dx++)
                for (int dy = -1; dy <= 1; dy++) {
                    if (dx == 0 && dy == 0)continue;
                    if (rec[ {x[i] + dx, y[i] + dy}]) {
                        e[i].push_back(rec[ {x[i] + dx, y[i] + dy}]);
                    }
                }
        }

    tarjan();
}

/* stuff you should look for
* int overflow, array bounds
* special cases (n=1?)
* do smth instead of nothing and stay organized
* WRITE STUFF DOWN
* DON'T GET STUCK ON ONE APPROACH
*/
```

概念简介

割点

对于无向连通图，去掉一个点，连通图图不再连通。该点就是割点。

割边

去掉一条边，连通图不再连通。该边就是割边。

双连通分量

1. 点双连通
   图没有割点。
2. 边双连通
   图没有割边
3. 双连通性质：
    1. 点双连通中：任意两点有两条以上简单路径（不经过重复点的两条路径。）
    2. 边双连通中：任意两点之间有两条以上不经过重复边的简单路径。

找到一个极大的点集合，满足导出子图双连通，是双连通分量。
将每一个割边去掉后，剩下的不相交图集合，是边双连通分量。
同理推广到点连通分量的定义中。

双连通分量缩点

1. 边双连通分量：
   去掉所有边即可。
2. 去掉割点，然后再对割点进行一些操作。(一个割点可能在不同的点双连通分块里)

双连通分量的缩图。

1. 边双收缩完之后依然是树
2. 点双缩完之后是block tree 圆方树。
   给割点建立一个辅助点。

求双连通分量

无向图上的边

无向图上的dfs树只有返祖边和树边。

Tarjan算法

现象

dfs树中，某子树中返祖边和割边的关系。子树下的点不再有边跳到dfs序更小的点。于是根和父亲的点就是割边。

割边

[割边 - 题目 - Daimayuan Online Judge]

1. 注意对两种特殊边的处理。自环没有影响。但是重边会影响一个点是否作为割边。
   重边的处理方法是，给这些边标记一些标号。
2. 算法步骤简述：
    1. 通过存储图的方式中：将每一条边打上标记。
    2. 用类似于tarjan的算法，关注返祖边。关注每一个点的子孙能够返祖到达的最低的dfn位置。
    3. 如果将一个点遍历完之后，发现子孙返祖的最大边就是自己。说明这个子树边连通。它与父亲的边就是割边。

自己的写法

```cpp
#include<bits/stdc++.h>
using namespace std;
using ll = long long;

const int N = 1E6 + 10;

vector<pair<int , int>> g[N];
int dfn[N] , low[N] , tot;
//无向图不可能存在横插边，所以不用开一个inv数组

vector<int> ans;
int n , m;


void dfs(int u , int id) {
    dfn[u] = low [u] = ++tot;
    for (auto v : g[u]) {
        if (dfn[v.first] == 0) dfs(v.first , v.second);
        if (id != v.second) low[u] = min(low[u] , low[v.first]);
    }

    if (dfn[u] == low[u] && id != -1) {
        ans.push_back(id);
    }
}


void tarjan() {
    for (int i = 1; i <= n; i ++) {
        if (!dfn[i]) dfs(i , -1);
    }
    int size = ans.size();
    sort(ans.begin() , ans.end());
    cout << size << '\n';
    for (int i = 0; i < size; i++) {
        cout << ans[i] << " \n"[i == n];
    }
}

int main()
{
    ios::sync_with_stdio(false);
    cin.tie(0);


    cin >> n >> m;
    for (int i = 1; i <= m; i++) {
        int x , y;
        cin >> x >> y;

        g[x].push_back({y , i});
        g[y].push_back({x , i});
    }
    tarjan();
}

/* stuff you should look for
* int overflow, array bounds
* special cases (n=1?)
* do smth instead of nothing and stay organized
* WRITE STUFF DOWN
```

```
* DON'T GET STUCK ON ONE APPROACH
*/
```

dls的写法

```cpp
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;

const int N = 101000;
vector<pair<int, int>> e[N];

int dfn[N], low[N], idx, n, m;
vector<int> bridge;

void dfs(int u, int id) {
    dfn[u] = low[u] = ++idx;
    for (auto [v, id2] : e[u]) {
        if (!dfn[v]) {
            dfs(v, id2);
            low[u] = min(low[u], low[v]);
            if (dfn[v] == low[v]) bridge.push_back(id2 + 1);
        } else if (id != id2) {
            low[u] = min(low[u], dfn[v]);
        }
    }
}

int main() {
    scanf("%d%d", &n, &m);
    for (int i = 0; i < m; i++) {
        int u, v;
        scanf("%d%d", &u, &v);
        e[u].push_back({v, i});
        e[v].push_back({u, i});
    }
    dfs(1, -1);
    sort(bridge.begin(), bridge.end());
    printf("%d\n", (int)bridge.size());
    for (auto x : bridge) printf("%d ",x);
    puts("");
}

```

生长思考：

对比两个代码。主要差别是， low的定义方式以及，判断桥记录桥的环节。

1. 关于low的定义方式：

   自己写的一份。沿袭了ssc的代码板子。对于灭一个节点的low的意义是，可以通过路径返祖到达的最小dfs序的祖先。

   dls的代码，代表的是对于每一个点如果low小于dfn[u]， 则代表了可以返祖。且是对于当前自己的子树上的返祖方式而言的。并且只用到了一次返祖边。

2. 因为这种和dfn有关的定义方式，和割点的求法相关。同时效果相同，所以追求统一。两份代码都使用了同一种写法。

关于桥的记录细节。对于一个点，它什么时候出现，并且已经计算完了low值。

1. 第一， 作为dfs(u , id)中的u。
2. 第二， dfs树种的父亲访问子节点的时候，对它进行dfs序后。

在第二个时间，不用特判，根处id不存在。

割点

关注现象

对于一颗子树，如果子树下的祖先边，不通过割点的返祖边的条件下（因为割点被割后，该点的边就不能用了。）不可以跳出子树外的祖先。那么子树的根就是割点。

- low， 一次跳可以达到的最大距离。

特殊情形下的特判：对于根，如果根自由一个儿子，那么不作为割点。

code

```cpp
#include<bits/stdc++.h>
using namespace std;
using ll = long long;

const int N = 1E6 + 10;
int n , m;

vector<int> g[N];
int cut[N]; // 记录是否为割点。
int dfn[N], low[N] , tot;

int size = 0;


void dfs(int u , int par) {
    dfn[u] = low[u] = ++tot;
    int sons = 0;
    for (auto v : g[u]) {
        if (dfn[v] == 0) {
            sons ++ ;
            dfs(v , u);
            low[u] = min(low[u] , low[v]);
            if (low[v] >= dfn[u])cut[u] = 1;
        }
        //为什么要判断重边?
        else {
            low[u] = min(low[u] , dfn[v]);
        }
    }

    if (u == 1 && sons <= 1)cut[u] = 0;
    size += cut[u];
}

void tarjan() {
    for (int i = 1; i <= n; i++) {
        if (dfn[i] == 0)
            dfs(i , 0);
    }
    cout << size << '\n';
    for (int i = 1; i <= n; i++)
        if (cut[i])cout << i << ' ';
    cout << '\n';
}


int main()
{
    ios::sync_with_stdio(false);
    cin.tie(0);
```

```
    cin >> n >> m;
    for (int i = 1; i <= m; i++) {
        int x , y;
        cin >> x >> y;
        g[x].push_back(y);
        g[y].push_back(x);
    }
    tarjan();
}

/* stuff you should look for
* int overflow, array bounds
* special cases (n=1?)
* do smth instead of nothing and stay organized
* WRITE STUFF DOWN
* DON'T GET STUCK ON ONE APPROACH
*/
```

生长思考：

- 是否要判断重复边?

  重边的计算，符合low的定义。没有影响。

- dfs下，每优化一个语句。都是非常大的常数优化。

几个例题，是将问题转换成缩点后的图（用双连通分量缩点）上的问题。

$first$ USACO 2006 Jan, Redundant Paths

USACO 2006 Jan, Redundant Paths - 题目 - Daimayuan Online Judge

solve

1. 发现在边双连通分量内部建边是没有意义的。

   在此启发之下，将图缩成一个，边双连通分量之间的树。

2. 于是转换成一个图上构造问题。分析得到每一个叶子（度数为1）都要搭一条边，融入边连通变量。发现最优的构造方法是，叶子叶子之间建立边。于是最优的操作是cunt_dag=1/2向上取整。

实现

1. 缩图。
2. 计算度数为1的点。

code

```
#include<bits/stdc++.h>
using namespace std;
using ll = long long;

const int N = 1E6 + 10;

vector<pair<int , int>> g[N];
int dfn[N] , low[N] , tot , bel[N];
//无向图不可能存在横插边，所以不用开一个inv数组
stack<int> stk;

vector<int> e;
int n , m;
vector<int> cc[N];
int cnt;

```

```cpp

void dfs(int u , int id) {
    dfn[u] = low [u] = ++tot;
    stk.push(u);

    for (auto temp : g[u]) {
        int v = temp.first;
        int id2 = temp.second;
        if (!dfn[v]) dfs(v , id2) , low[u] = min(low[u] , low[v]);
        else if (id != id2) low[u] = min(low[u] , dfn[v]);
    }

    if (low[u] == dfn[u]) {
        ++cnt;
        while (true) {
            int v = stk.top(); stk.pop();
            bel[v] = cnt;
            cc[cnt].push_back(v);
            if (v == u)break;
        }
    }
}


void tarjan() {
    for (int i = 1; i <= n; i ++) {
        if (!dfn[i]) dfs(i , -1);
    }
    int ans = 0;
    for (int i = 1; i <= cnt; i++) {
        int cunte = 0;
        for (auto u : cc[i])
            for (auto temp : g[u]) {
                int v = temp.first;
                if (bel[u] != bel[v])cunte ++;
            }
        ans += (cunte == 1);
    }
    cout << (ans + 1) / 2 << '\n';
}

int main()
{
    ios::sync_with_stdio(false);
    cin.tie(0);


    cin >> n >> m;
    for (int i = 1; i <= m; i++) {
        int x , y;
        cin >> x >> y;

        g[x].push_back({y , i});
        g[y].push_back({x , i});
    }
    tarjan();
}

/* stuff you should look for
* int overflow, array bounds
* special cases (n=1?)
* do smth instead of nothing and stay organized
* WRITE STUFF DOWN
* DON'T GET STUCK ON ONE APPROACH
```

```
*/
```

生长

- 不需要引入inv变量。 inv变量记录是否在栈种。来防止横插边。

second BLO

POI 2008, BLO - 题目 - Daimayuan Online Judge

solve

关注图中的两种点： 割点和普通点。利用点双连通分量将图收缩。发现如果点作为子树的割点。那么当把点割掉之后，子树中的所有点都不再和其它点连通，此时可以计算出部分的贡献。
如果是一般点，贡献的改变只在于删除点。

code

```cpp
#include<bits/stdc++.h>
using namespace std;
using ll = long long;

const int N = 1E6 + 10;

int n , m;

vector<int>e[N];
int dfn[N] , low[N] ,  id;
long long ans[N];
int sz[N];

void dfs(int u , int par) {
    dfn[u] = low[u] = ++id;
    sz[u] = 1;
    ans[u] = n - 1;//割舍走的点和其它的点匹配的贡献。
    int cut = n - 1;

    for (auto v : e[u]) {
        if (!dfn[v]) {
            dfs(v, u);
            low[u] = min(low[u] , low[v]);
            sz[u] += sz[v];
            if (low[v] >= dfn[u]) {
                //说明这个点对应于当前子树下，是一个割点。
                ans[u] += 1LL * sz[v] * (n - sz[v]);
                cut -= sz[v];
            }
        }
        else if (v != par) low[u] = min(low[u] , dfn[v]);
    }

    ans[u] += 1LL * cut * (n - cut);
}

void tarjan()
{
    dfs(1 , 0);
    for (int i = 1; i <= n; i++)
        cout << ans[i] << '\n';
}

int main()
{
    ios::sync_with_stdio(false);
```

```
    cin.tie(0);

    cin >> n >> m;
    for (int i = 1; i <= m; i++) {
        int x, y;
        cin >> x >> y;
        e[x].push_back(y);
        e[y].push_back(x);
    }
    tarjan();
}

/* stuff you should look for
* int overflow, array bounds
* special cases (n=1?)
* do smth instead of nothing and stay organized
* WRITE STUFF DOWN
* DON'T GET STUCK ON ONE APPROACH
*/
```

$Third$: HNOI2012, 矿场搭建

利用点双连通分量，将图收缩。

挖掘性质。

- 如果不存在割点。那么要在图间设置两个入口。方案数用组合计算即可。
- 如果存在割点，缩点后，入度为1的节点都设置一个点（不能设置在割点处。）其实就是当前分量中只有一个节点。

```
#include<bits/stdc++.h>
using namespace std;
typedef long long ll;

const int oo = 0x0fffffff;
int n = 1E3 + 10;

bool work(int testNo)
{
    int T;
    scanf("%d" , &T);
    if (T == 0)
        return true;
    vector<vector<int>> g(n + 1);
    for (int i = 0; i < T; i++) {
        int x, y;
        scanf("%d%d", &x , &y);
        g[x].push_back(y);
        g[y].push_back(x);
    }
    vector<int> low(n + 1, 0) , dfn(n + 1 , 0) ;
    //记录点双连通分量。割点除外的点双连通分量。
    vector<vector<int>> cc;
    int id = 0;
    vector<int> cut(n + 1 , 0);
    stack<int> stk;

    function<void(int , int)> dfs = [&](int u , int par)->void{
        dfn[u]  = low[u] = ++id;

        stk.push(u);
        int ch = 0;

        for (auto v : g[u]) {
            if (!dfn[v]) {
```

```
                ch ++;
                dfs(v , u);
                low[u] = min(low[u] , low[v]);
                if (low[v] >= dfn[u]) {
                    cut[u] = 1;
                    vector<int>c;
                    c.push_back(u);
                    while (true) {
                        int w = stk.top(); stk.pop();
                        c.push_back(w);
                        if (w == v)break;
                    }
                    cc.push_back(c);
                }
            }
            else if (v != par)
                low[u] = min(low[u] , dfn[v]);
        }

        if (par == 0 && ch <= 1)cut[u] = 0;
    };

    for (int i = 1; i <= n; i++)if (!dfn[i])
            dfs(i , 0);

    //然后分类讨论统计答案。
    //如果只有一个点双连通分量。那么就是2.
    //如果有多个就是每一个连通块都要放置一个连通分量。
    unsigned long long ans1 = 0 , ans2 = 0;
    if (cc.size() == 1) {
        ans1 = 2;
        int nn = cc[0].size();
        ans2 = 1ULL * (nn) * (nn - 1) / 2;
    }
    else {
        ans1 = 0;
        ans2 = 1;
        for (auto c : cc) {
            int ncut = 0;
            for (auto u : c) {
                ncut += cut[u];
            }
            if (ncut == 1) {
                ans1++;
                ans2 *= (1ULL * c.size() - 1);
            }
        }
    }
    printf("Case %d: %lld %lld\n" , testNo , ans1 , ans2);
    return false;
}


int main()
{
    int id = 0;
    while (true) {
        if (work(++id))break;
    }
}

/* stuff you should look for
* int overflow, array bounds
* special cases (n=1?)
```

```
* do smth instead of nothing and stay organized
* WRITE STUFF DOWN
* DON'T GET STUCK ON ONE APPROACH
*/

```

概念

差分约束系统

一种特殊的n元不等式组。包含n个变量和m个约束条件，其中每一个约束条件由两个变量的差组成。

对于$x_1....x_n$
形如$x_i - x_j <= c$不等式限制约束下，符合条件的解。或者判断不存在解。

定义上参见：

差分约束 - OI Wiki (oi-wiki.org)

差分约束系统 - 题目 - Daimayuan Online Judge

solve

1. 将不等式转换成：

$$x_i <= c + x_j \tag{5}$$

2. 一种方法如下：预设， 所有的$d_i = 0$。然后遍历上述限制。

   `x_i = min(x_i , c + x_j)`

   不断调整直到符合题意。

3. 发现上述过程等效于bellman算法。
    1. 每一个具体的问题，都可以有一个相关的图匹配。
    2. 唯一的差别是， 初始化上bell_man的初始化为inf。

理解等效性：

1. 将上述关系建立成一个图。d[i]初始化为0，
2. 相当于x_1到所有当前的已知点的最短距离为0.然后进行更新。
3. 如果存在负数圈，找不到解。（将圈上相关关系的不等式，通过消元，得到0<= 负数）
4. 没有负数圈，一定可以找到相关的解？
    1. 如果是一条链状。必然有解。
    2. 如果是一个非负环。按照边的递增方式最终必然会出现一个解。
    3. 如果不再更新说明对于任意点满足$x_i <= x_j + c$;

回到问题本身

1. 求非负数解：
   使用技巧：引入一点$x_0 = 0$，引入关系$x_i - x_0 >= 0$
2. 求最小解。
   事实上，数与数之间，由于边的建立为c。两点之间如果通过该点更新，将会是差为c事实上，差距可能可以更小。
   逆向思维：将最小值问题转化成最大值问题。
   调整不等式为：

$$\begin{gathered} x_i - x_j <= c \\ -(-x_i) + (-x_j) <= c \\ 令x_i \= -x_i, x_j \= -x_j \end{gathered} \tag{6}$$

初始化时，为了保证，任意两点之间的差距等于边。所以初始化$x_i = oo$

然后得到了变化后的变量的最大值，等效于找到了目标的最小值。

然后跑一遍bellman_ford即可。如果数据太大考虑spfa(加强版的)

code

```cpp
#include<bits/stdc++.h>
using namespace std;
using ll = long long;

const int N = 1E6 + 10;

vector<array<int , 3>> e;
int d[N];
int dif = 1E9;

int main()
{
    ios::sync_with_stdio(false);
    cin.tie(0);

    int n , m;
    cin >> n >> m;
    for (int i = 0; i < m; i++) {
        int u , v , w;
        //v - > u ,w --> u -> v w
        cin >> u >> v >> w;
        e.push_back({v , u , w});//反过来。
    }
    for (int i = 1; i <= n; i++) {
        // i -> 0  w = > 0 -> i  w
        e.push_back({ i , 0 , 0});
    }
    for (int i = 1; i <= n; i++)
        d[i] = dif;
    for (int i = 1; i <= n; i++)
        for (int j = 0; j < (int)e.size(); j++) {
            int u = e[j][0] , v = e[j][1] , w = e[j][2];
            d[u] = min(d[u] , d[v] + w);
        }
    for (int i = 0; i <= m; i++) {
        int u = e[i][0] , v = e[i][1] , w = e[i][2];
        if (d[u] > d[v] + w) {
            cout << -1 << '\n';
            return 0;
        }
    }
    for (int i = 1; i <= n; i++) {
        cout << d[0] -  d[i] << " \n"[i == n];
    }

}

/* stuff you should look for
* int overflow, array bounds
* special cases (n=1?)
* do smth instead of nothing and stay organized
* WRITE STUFF DOWN
* DON'T GET STUCK ON ONE APPROACH
*/
```

first 问题一

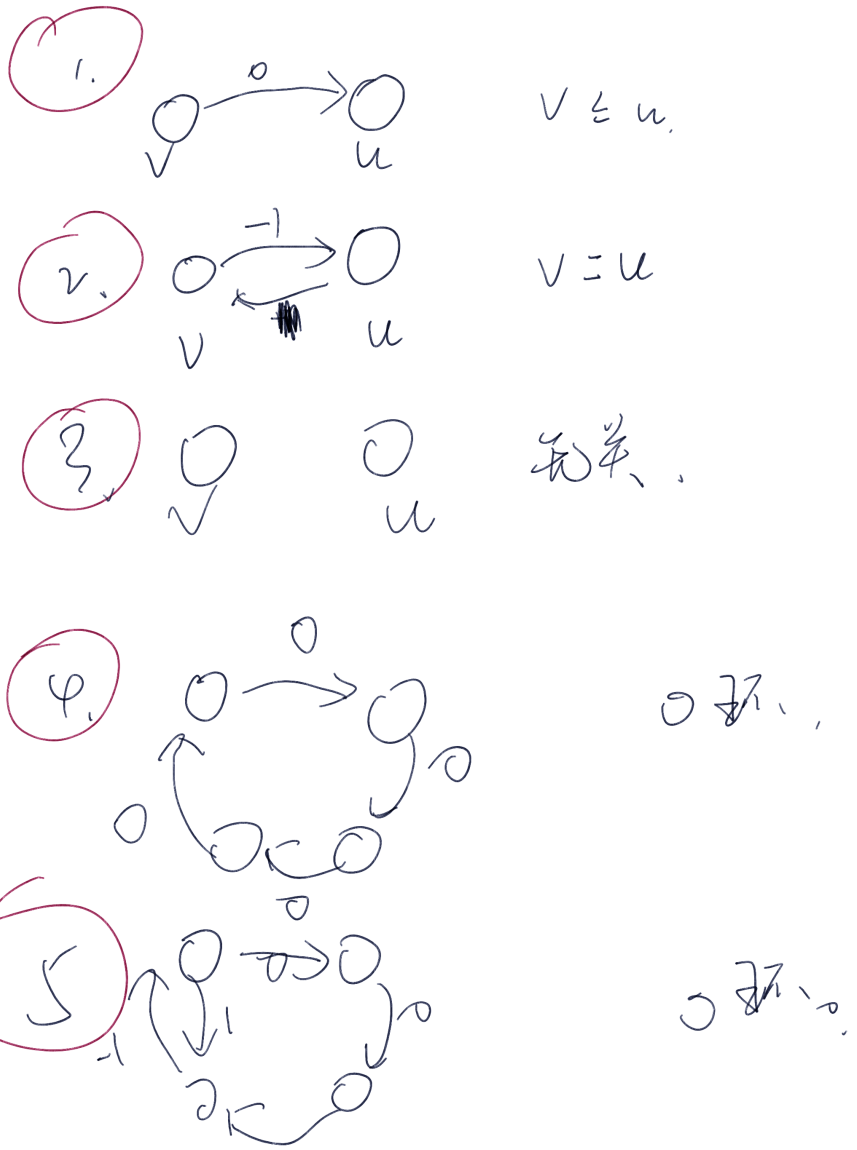
POI2012, Festival - 题目 - Daimayuan Online Judge

solve

1. 很容易对题中给出的约束条件，建立差分约束系统。
2. 问题是，追求数的种类尽量多。

观察差分约束建立的图和问题之间的关系：

图中有如下几种环结构



图中有如上5种环结构。
对于第①种，可以使它们不相交
对于互相可达的两点，它们之间的差有一定的关系。

几个结论

1. 利用强连通分量对图缩点。非强连通的两个分量，可以控制不相交：
    1. 比方说，可以将一个分量中的点放到非常大，无穷大。另外一个分量中的点与它的差也无穷大。这样在满足前提条件下，使他们并不相交。
2. 对于缩点内部，点的最大种数，是内部点之间的最长路径。找出$max - min$。利用最短路求出，
    1. 强连通分量内部的点之间的最短路，必然只由强连通分量内部的边构成。
    2. 每条边的可能是 0， 1， -1.所以各种最最大与最小值之间的整数路径长度都存在。

3. 如果环内部出现0这种约束。除非全0否则无解。
4. 对于只有-1 ， 1的强连通分量内部。追求每种边都不一样。
    1. 找到这种连通量中的点数。
    2. 直接做最短路计算。（forad）顺便判断负环无解情况。

code(dls)

反正都是模仿dls来写的。

```cpp
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;

const int N = 610;
const int inf = 1 << 29;

int n, m1, m2;
int g[N][N];
bool vis[N];

int main() {
    scanf("%d%d%d", &n, &m1, &m2);
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= n; j++) {
            g[i][j] = (i == j) ? 0 : inf;
        }
    }
    for (int i = 1; i <= m1; i++) {
        int a, b;
        scanf("%d%d", &a, &b);
        g[b][a] = min(g[b][a], -1);
        g[a][b] = min(g[a][b], 1);
    }

    for (int i = 1; i <= m2; i++) {
        int a, b;
        scanf("%d%d", &a, &b);
        g[b][a] = min(g[b][a], 0);
    }
    for (int k = 1; k <= n; k++)
        for (int i = 1; i <= n; i++)
            for (int j = 1; j <= n; j++)
                g[i][j] = min(g[i][j], g[i][k] + g[k][j]);
    for (int i = 1; i <= n; i++)
        if (g[i][i] < 0) {
            puts("NIE");
            return 0;
        }
    int ans = 0;
    for (int i = 1; i <= n; i++) if (!vis[i]) {
        vector<int> v;
        for (int j = 1; j <= n; j++)
            if (g[i][j] <= n && g[j][i] <= n) {
                v.push_back(j);
                vis[j] = true;
            }
        int d = 0;
        for (auto p : v)for (auto q : v)
            d = max(d, g[p][q]);
        ans += d + 1;
    }
    printf("%d\n", ans);
}
```

概念简介

借鉴：

2-SAT - OI Wiki (oi-wiki.org)

背景

SAT是适用性的简称。一般形式为k-SAT.但是当k>2时。建模较麻烦，且复杂度比较高。

简述

给定N个布尔变量。若干个或表达式约束。
给每一表达式中的一对变量分配真假。求符合约束的一组解。

2-SAT，简单的说就是给出 n 个集合，每个集合有两个元素，已知若干个 <a,b>，表示 a 与 b 矛盾（其中 a 与 b 属于不同的集合）。然后从每个集合选择一个元素，判断能否一共选 n 个两两不矛盾的元素。显然可能有多种选择方案，一般题中只需要求出一种即可。

问题背景往往是n个集合中，每个集合选一个。当然可以选两个。但是如果选两个必然存在每个集合选一个的一种方案。（贪心）

建模方法

1. 二元布尔表达式都可以转换成或表达式（离散数学中的内容。）
2. 建图：
    1. 枚举假设，变量的情况。比方说$x_1 = 1$，$x_2 = 0$至少一个成立
    2. 关注两种确定的组合：

$$x_1 = 0 => x_2 = 1 \tag{7}$$

$$x_2 = 1 => x_1 = 1 \tag{8}$$

        1. 为什么要关注这两种假设？两个都成立的假设不考虑？
        2. 因为我们使得变量为真，有数量上的限制。我们追求用尽量少变量为真的数量，得到一组解。
    3. 根据上述描述建立边。。对每一个变量拆成两个点。
3. 图上性质。
    1. 选择了一个点作为方案结构。不能再选择与之矛盾的解。
    2. 选择了一个点作为方案结构。其可达的点都要选择。

求解方法

1. 判断是否有解。

    根据规则：只要对每一个变量分配了真假。并且不出现矛盾。那么就是有解。可知当且仅当，不存在相矛盾的两个点（x和！x）在同一个连通分量中。

2. 具体构造出一组解。
    1. 先缩点。然后判断是否可以构造出一组解。
    2. 在缩完点的dag上，任选一个拓扑序。从拓扑序靠后的开始选择。
    3. 关于1，2的证明。
        1. 首先关注，几种出现矛盾的可能情况。选择了$x_i, x_i => .... => !x_i$同时选了两个。但是我们优先选择拓扑序小的。就不会先选到x_i
        2. 从拓扑序小的开始选择。每一条边都有一个反边。也就是具有了一种对称性。缩点都有其对称的内容。就是缩点内部涉及了相同类型的变量。所以考察一个缩点。
            1. 如果对称的缩点已经选择了。那么这个缩点就不必再选了。
            2. 如果对称的缩点还没有选择。可以保证子孙都已经被选择，没有空隙。观察图的结构。可以发现它是具有点的关于边的对称性的。如果fa子孙中同时出现了出现了$x_i, !x$。那么也必然出现了$!fa$。矛

盾。

例题1 满汉全席

[P4171 JSOI2010] 满汉全席 - 洛谷 | 计算机科学教育新生态 (luogu.com.cn)

solve

问题就是给定若干对。求一个解，使得其中的元素至少存在一个。典型的2—SAT问题。

1. 约束定义
   对于每一个变量有状态就是为h和为m。类比true 和 false. 定义 m为 0 状态。 h为1状态。
2. 建模

选择一个取反操作。然后指向另一个即可。

3. 缩点

tarjan。一步到位。因为不用求一个具体的结构。所以不需要记录拓扑序。（或者在tarjan过程中直接求解。）

4. 利用bel数组判断有无解即可。

code

```cpp
#include<bits/stdc++.h>
using namespace std;
typedef long long ll;

const int oo = 0x0fffffff;
const int N = 1E6 + 10;

void work(int testNo)
{
    int n , m;
    cin >> n >> m;

    vector<vector<int>> e(2 * n);

    for (int i = 0; i < m; i ++) {
        char a , b;
        int u , v;
        cin >> a; cin >> u; cin >> b; cin >> v;
        u--; v--;

        u = u * 2 + (a == 'h');
        v = v * 2 + (b == 'h');
        e[u ^ 1].push_back(v);
        e[v ^ 1].push_back(u);
    }

    vector<int> dfn(2 * n) , low(2 * n) , bel(2 * n) , ins(2 * n);
    int id = 0;
    stack<int> stk;
    int cnt = 0;

    function<void(int)> dfs = [&](int u) {
        dfn[u] = low[u] = ++id;
        ins[u] = true;
        stk.push(u);

        for (auto v : e[u]) {
            if (!dfn[v]) dfs(v);
            if (ins[v]) low[u] = min(low[v] , low[u]);
        }

        if (low[u] == dfn[u]) {
```

```
            ++cnt;
            while (true) {
                int v = stk.top();
                stk.pop();
                ins[v] = false;
                bel[v] = cnt;
                if (v == u) break;
            }
        }
    };

    for (int i = 0; i < 2 * n; i++) {
        if (dfn[i] == 0) dfs(i);
    }

    for (int i = 0; i < 2 * n; i++) {
        if (bel[i ^ 1] == bel[i]) {
            cout << "BAD\n";
            return;
        }
    }

    cout << "GOOD\n";
}


int main()
{
    ios::sync_with_stdio(false);
    cin.tie(0);

    int t; cin >> t;
    for (int i = 1; i <= t; i++)work(i);
}

/* stuff you should look for
* int overflow, array bounds
* special cases (n=1?)
* do smth instead of nothing and stay organized
* WRITE STUFF DOWN
* DON'T GET STUCK ON ONE APPROACH
*/

```

生长思考。

1. 拆点技巧。乘2。利用数字的第一位信息区分。
2. 用lambda表达式，使码风更简洁，清晰。
    1. 可能传引用调用，局部调用速度更快。
    2. 更利于封装。模板优化。

例题二 奶牛议会。

每个奶牛给出两个方案的决定。要求两个变量至少一个成立才为可行解。

问法变化：

1. 考察所有可行解。研究，一个议案的情况：所有可行方案都通过 。所有方案不通过。部分方案通过，部分方案不通过。三种情况。

solve

1. 是否存在解的问题，求强连通分量即可。
2. 基于某一个提案，分析所有方案结构
    1. 假设只能为真，看是否有解。 即给定一条!x->x
    2. 假设只能为假，是否有解。 同一引入 x->!x
3. 分析出结论，如果x可达!x，那么不存在x的方案。所以此时为'N'反之如果!x可达x. ， 那么不存在！x的提案，

关于实现

1. 对于第3点。求出每一个点的可达情况。不可以使用树形dp。因为状态与状态间会互相影响。最多通过bitset来做一个常数优化。类似的问题：清楚姐姐学排序.md

code

```cpp
#include<bits/stdc++.h>
using namespace std;
using ll = long long;

const int N = 1E6 + 10;


void tarjan()
{
    int n , m;
    cin >> n >> m;

    //建图
    //情景不同数组可能开更大。注意
    vector<vector<int>> e(n * 2);

    for (int i = 0; i < m; i ++) {
        /*处理边输入*/
        int a[2];
        char s[2];
        cin >> a[0] >> s[0] >> a[1] >> s[1];
        a[0] --;
        a[1] --;
        a[0] = a[0] * 2 + (s[0] == 'N');
        a[1] = a[1] * 2 + (s[1] == 'N');
        e[a[0] ^ 1].push_back(a[1]);
        e[a[1] ^ 1].push_back(a[0]);
    }

    vector<int> dfn(n * 2) , low(n * 2) , bel(n * 2) , ins(n * 2);
    int id = 0;
    stack<int> stk;
    int cnt = 0;

    vector<vector<int>> ssc(n * 2);

    function<void(int)> dfs = [&](int u) {
        dfn[u] = low[u] = ++id;
        ins[u] = true;
        stk.push(u);

        for (auto v : e[u]) {
            if (!dfn[v]) dfs(v);
            if (ins[v]) low[u] = min(low[v] , low[u]);
        }

        if (low[u] == dfn[u]) {
```

```
            ++cnt;
            vector<int> c;
            while (true) {
                int v = stk.top();
                stk.pop();
                ins[v] = false;
                bel[v] = cnt;
                if (v == u) break;
            }
            ssc.push_back(c);
        }
    };

    for (int i = 0; i < 2 * n; i++) {
        if (dfn[i] == 0) dfs(i);
    }

    for (int i = 0; i < n; i++) {
        if (bel[i * 2 + 1] != bel[i * 2]) {
            cout << "IMPOSSIBLE\n";
            return ;
        }
    }
    vector<int> vis(n * 2);
    function<void (int)> dfs2 = [&](int u) {
        vis[u] = true;
        for (auto v : e[u]) {
            if (!vis[v])dfs(v);
        }
    };
    string ans;
    ans.resize(n + 1);
    for (int i = 0; i < n; i++) {
        fill(vis.begin() , vis.end(), 0);
        dfs2(i * 2);
        ans[i] = '?';
        if (vis[i * 2 + 1]) {
            ans[i] = 'N';
        }
        fill(vis.begin(), vis.end() , 0);
        dfs2(i * 2 + 1);
        if (vis[i * 2]) {
            ans[i] = 'Y';
        }
    }
    cout << ans << '\n';
}

int main()
{
    ios::sync_with_stdio(false);
    cin.tie(0);

    tarjan();
}

/* stuff you should look for
* int overflow, array bounds
* special cases (n=1?)
* do smth instead of nothing and stay organized
* WRITE STUFF DOWN
* DON'T GET STUCK ON ONE APPROACH
*/
```

kosaraju

```cpp
const int N = 1E6 + 10;

vector<vector<int>> e , erev;

vector<vector<int>> ssc;
vector<int> c;
bool vis[N];

vector<int>out;

int n , m;

void dfs(int u)
{
    vis[u] = true;
    for (auto v : e[u]) {
        if (vis[v])continue;
        dfs(v);
    }
    out.push_back(u);
}

void dfs2(int u) {
    vis[u] = true;
    for (auto v : erev[u]) {
        if (vis[v])continue;
        dfs2(v);
    }
    c.push_back(u);
}

void kosaraju() {
    for (int i = 1; i <= n; i ++)
    {
        if (vis[i])continue;
        dfs(i);
    }
    memset(vis, false, sizeof(vis));
    reverse(out.begin(), out.end());
    for (auto u : out) {
        if (vis[u])continue;
        c.clear();
        dfs2(u);
//      sort(c.begin(), c.end());
        ssc.push_back(c);
    }
}

/*
*  1.注意建一个反图
*  2.图的初始化没有完成。
*  3.main上记得写下函数
*/
```

tarjan

```cpp
const int N = 1E6 + 10;

vector<int> e[N];
int dfn[N] , low[N] , bel[N], ins[N] , id;

vector<vector<int>> ssc;
stack<int>stk;

void dfs(int u) {

    dfn[u] = low[u] = ++id;
    ins[u] = true;
    stk.push(u);

    for (auto v : e[u]) {
        if (!dfn[v]) dfs(v);
        if (ins[v]) low[u] = min(low[v] , low[u]);
    }

    //说明已经没有收割的空间了：
    if (low[u] == dfn[u]) {
        vector<int> s;
        while (true) {
            int v = stk.top();
            stk.pop();
            s.push_back(v);
            ins[v] = false;
            bel[v] = ssc.size();
            if (v == u)break;
        }
        //sort(s.begin(), s.end());
        ssc.push_back(s);
    }
}

void tarjan()
{
    for (int  i = 1; i <= n; i++) {
        if (dfn[i] == 0)
            dfs(i);
    }
}
```

lambda封装版本

```cpp
void tarjan()
{
    int n , m;
    cin >> n >> m;

    //建图
    //情景不同数组可能开更大。注意
    vector<vector<int>> e(n + 1);

    for (int i = 0; i < m; i ++) {
        /*处理边输入*/
    }

    vector<int> dfn(n + 1) , low(n + 1) , bel(n + 1) , ins(n + 1);
    int id = 0;
```

```
    stack<int> stk;
    int cnt = 0;

    vector<vector<int>> ssc(n + 1);

    function<void(int)> dfs = [&](int u) {
        dfn[u] = low[u] = ++id;
        ins[u] = true;
        stk.push(u);

        for (auto v : e[u]) {
            if (!dfn[v]) dfs(v);
            if (ins[v]) low[u] = min(low[v] , low[u]);
        }

        if (low[u] == dfn[u]) {
            ++cnt;
            vector<int> c;
            while (true) {
                int v = stk.top();
                stk.pop();
                ins[v] = false;
                bel[v] = cnt;
                if (v == u) break;
            }
            ssc.push_back(c);
        }
    };

    for (int i = 0; i < 2 * n; i++) {
        if (dfn[i] == 0) dfs(i);
    }
}
```

tarjan 求边双连通分量缩图

```
const int N = 1E6 + 10;

vector<pair<int , int>> g[N];
int dfn[N] , low[N] , tot , bel[N];
//无向图不可能存在横插边，所以不用开一个inv数组
stack<int> stk;

vector<int> e;
int n , m;
vector<int> cc[N];
int cnt;


void dfs(int u , int id) {
    dfn[u] = low [u] = ++tot;
    stk.push(u);

    for (auto temp : g[u]) {
        int v = temp.first;
        int id2 = temp.second;
        if (!dfn[v]) dfs(v , id2) , low[u] = min(low[u] , low[v]);
        else if (id != id2) low[u] = min(low[u] , dfn[v]);
    }

    if (low[u] == dfn[u]) {
```

```
        ++cnt;
        while (true) {
            int v = stk.top(); stk.pop();
            bel[v] = cnt;
            cc[cnt].push_back(v);
            if (v == u)break;
        }
    }
}


void tarjan() {
    for (int i = 1; i <= n; i ++) {
        if (!dfn[i]) dfs(i , -1);
    }
}
```

一个明智地追求快乐的人，除了培养生活赖以支撑的主要兴趣之外，总得设法培养其他许多闲情逸致。