ACM模版

//组合数C(n,m)打表

//公式C[n][m]=C[n-1][m]+C[n-1][m-1]

C[1][0]=1;

C[1][1]=1;

for(int i=2;i<n;i++)

{

C[i][0]=1;

for(int j=1;j<m;j++)

C[i][j]=(C[i-1][j]+C[i-1][j-1])%mod;

}

//快速幂模版

long long quick\_pow(long long n,long long k)

{

n=n%mod; //WA 大数进来要先mod

long long ans=1;

while(k>0)

{

if(k&1)

ans=ans\*n%mod;

n=n\*n%mod;

k>>=1;

}

return ans;

}

//筛选1-n的质数,存在prime[]数组里,下标从0开始

void filterPrime(int n)

{

bool isPrime[MAX];

primeCnt=0;

memset(isPrime,true,sizeof(isPrime));

isPrime[0]=false;

isPrime[1]=false;

for(int i=2;i<=n;i++)

{

if(isPrime[i]==true)

{

prime[primeCnt++]=i;

for(int j=i\*2;j<=n;j=j+i)

isPrime[j]=false;

}

}

}

//rmq

//rmq[i][j]表示从第j个数开始,长度为2^i的最大值

for(int i=1;i<=N;i++)

scanf("%d",&rmq[0][i]);

for(int i=1;(1<<i)<=N;i++)

for(int j=1;j+(1<<i)-1<=N;j++)

rmq[i][j]=min(rmq[i-1][j],rmq[i-1][j+(1<<(i-1))]);

int rmqQuery(int l,int r)

{

int k=0;

while((1<<(k+1))<=r-l+1)

k++;

return min(rmq[k][l],rmq[k][r-(1<<k)+1]);

}

int gcd(int a,int b)

{

return b==0?a:gcd(b,a%b);

}

求LCM的时候注意溢出

先做除法

//裸LIS O(nlogn)

int A[MAX];

int B[MAX];

int main(void)

{

int T;

scanf("%d",&T);

while(T--)

{

int n;

scanf("%d",&n);

for(int i=0;i<n;i++)

scanf("%d",&A[i]);

int ans=0;

B[ans]=A[0];

for(int i=1;i<n;i++)

{

if(A[i]>B[ans]) //没有等号是严格LIS

B[++ans]=A[i];

else

{

int idx=(int)(upper\_bound(B,B+ans,A[i])-B);

B[idx]=A[i];

}

}

printf("%d\n",ans+1); //因为ans从0开始

}

return 0;

}

//B[i]表示 长度为i+1的LIS末尾元素最小为多少,末尾越小当然越优

//遍历A数组,维护B数组,最后B数组元素个数即为答案

//由于B数组单调递增,可二分查找

//

//upper\_bound(A,A+n,x)返回第一次出现x的位置

//lower\_bound(A,A+n,x)返回最后一次出现x的位置的下一个

//判断a->b->c旋转方向

int ccw(const struct node &a,const struct node &b,const struct node &c)

{

double area2 = ((double)b.x-a.x)\*(c.y-a.y) - (b.y-a.y)\*(c.x-a.x);

if (area2 < 0) return -1; // clockwise

else if (area2 > 0) return 1; // counter-clockwise

else return 0; // collinear

}

//已知正方形任意两点,求另外两点公式

//已知(x1,y1),(x2,y2)

//p1(x1+(y1-y2),y1-(x1-x2)) p2(x2+(y1-y2),y2-(x1-x2))

//或

//p1(x1-(y1-y2),y1+(x1-x2)) p2(x2-(y1-y2),y2+(x1-x2))

//最后答案除以4

第二类Stirling数:

s(p,k)把p元素集合划分到k个不可区分的盒子里且没有空盒子的划分个数。

递推公式：

S(p,p)=1(p≥0)

S(p,0)=0(p≥1)

S(p,k)=k∗S(p−1,k)+S(p−1,k−1)(1≤k≤p−1)

第一类Stirling数:

s(p,k)把p个对象排成k个非空循环排列的方法数。

意思是n个不同元素构成m个圆排列的方案数

递推公式：

S(p,p)=1(p≥0)

S(p,0)=0(p≥1)

S(p,k)=(p−1)∗S(p−1,k)+S(p−1,k−1)(1≤k≤p−1)

//Ramsey Theorem

//6个人中至少存在3人相互认识或者相互不认识。

//复杂度O(row\*row\*col\*log(col))

int maxSumSubmatrix(vector< vector<int> >& matrix, int k)

{

int N=(int)matrix.size();

if(N==0)

return 0;

int M=(int)matrix[0].size();

if(N>M) //矩阵转置,使复杂度减小

{

//括号里是(个数,初值) auto???

auto newMatrix=vector< vector<int> >(M,vector<int>(N,0));

for(int i=0;i<N;i++)

for(int j=0;j<M;j++)

newMatrix[j][i]=matrix[i][j];

return maxSumSubmatrix(newMatrix,k);

}

//构造列方向上的前缀和

int colSum[N+1][M];

memset(colSum,0,sizeof(colSum));

for(int i=1;i<=N;i++)

for(int j=0;j<M;j++)

colSum[i][j]=colSum[i-1][j]+matrix[i-1][j];

int ans=INT\_MIN;

//枚举任意两行,即转换问题为:一维数组上子序列不超过k的最大值

for(int i=0;i<=N;i++)

{

for(int j=i+1;j<=N;j++)

{

int array[M];

memset(array,0,sizeof(array));

//通过之前列方向上的前缀和 计算两行之间夹的部分

for(int p=0;p<M;p++)

array[p]=colSum[j][p]-colSum[i][p];

//计算夹的部分的前缀和

for(int p=1;p<M;p++)

array[p]=array[p-1]+array[p];

//再次转换问题为:对于array数组,找到两个位置i,j,使array[j]-array[i]不大于k,且使其最大

//思路:从前往后遍历j,搜索之前的i是否满足条件array[j]-array[i]不大于k,并更新答案

//使用Set,可有序的存放array[i],达到二分查找的目的

set<int> Set;

Set.insert(0); //表示array[0]前缀和没有元素

for(int p=0;p<M;p++)

{

set<int>::iterator iter=Set.lower\_bound(array[p]-k);

if(iter!=Set.end())

ans=max(ans,array[p]-\*iter);

Set.insert(array[p]);

}

}

}

return ans;

}

SPFA算法

Bellman-Ford的优化版本，不需要像Bellman-Ford算法一样把每条边都更新n次

vector模拟邻接表存储，dist初值为INF

首先将源点入队，若一个结点的dist更新了，将他进队去更新他的邻接点

判断负环：如一个结点进队超过n次则存在负环

SLF优化：deque维护

#include <iostream>

#include <queue>

#include <vector>

#include <cstring>

#define MAX 10

#define INF 0xfffffff

using namespace std;

struct node{

int u;

int w;

};

vector<vector<struct node>> G;

int dist[MAX];

bool ifInQueue[MAX];

int updateTimes[MAX]; //若大于n,则有负环

int F,N,M,W;

bool SPFA(int S);

int main(void)

{

scanf("%d",&F);

while(F--)

{

scanf("%d %d %d",&N,&M,&W);

G.clear();

G.resize(N+1);

for(int i=0;i<M;i++)

{

struct node p;

int u;

scanf("%d %d %d",&u,&p.u,&p.w);

G[u].push\_back(p);

int temp=u;

u=p.u;

p.u=temp;

G[u].push\_back(p);

}

for(int i=0;i<W;i++)

{

struct node p;

int u;

scanf("%d %d %d",&u,&p.u,&p.w);

p.w=-p.w;

G[u].push\_back(p);

}

bool ans=SPFA(1);

if(ans==true)

printf("YES\n");

else

printf("NO\n");

}

return 0;

}

bool SPFA(int S)

{

memset(ifInQueue,0,sizeof(ifInQueue));

memset(updateTimes,0,sizeof(updateTimes));

for(int i=0;i<=N;i++)

dist[i]=INF;

deque<int> que;

dist[S]=0;

ifInQueue[S]=true;

que.push\_back(S);

while(!que.empty())

{

int u=que.front();

que.pop\_front();

ifInQueue[u]=false;

for(int i=0;i<G[u].size();i++)

{

int v=G[u][i].u;

if(dist[u]+G[u][i].w<dist[v])

{

dist[v]=dist[u]+G[u][i].w;

updateTimes[v]++;

//进队超过n次则有负环

if(updateTimes[v]>=N)

return true;

//把最短路径更新的结点入队

if(ifInQueue[v]==false)

{

ifInQueue[v]=true;

//SLF优化,先更新短边

//注意万一队列为空,那访问dis[que.front]不合法

//所以que.empty()放在前面

if(que.empty()==false && dist[v]<dist[que.front()])

que.push\_front(v);

else

que.push\_back(v);

}

}

}

}

return false;

}

//316K 157MS

Dijkstra算法，堆优化

vector模拟邻接表，注意resize，dist数组初始化为INF

priority\_queue重载<，让dist小的在前

首先将源点入队，访问他的邻接点，如果这个结点未被访问且能更新变小，则更新dist数组，并将该结点入队，每次从pq中弹出一个最短的边

#include <iostream>

#include <vector>

#include <algorithm>

#include <queue>

#include <cstdio>

#define N 10

#define INF 0xfffffff

using namespace std;

struct node{

int v;

int w;

};

vector<vector<node>> v;

//vector<CNode> v[30010]; error,如果用这个，则在poj会超时。说明vector对象的初始化，也是需要可观时间的

priority\_queue<node> pq;

int dist[N];

bool vis[N];

int n,m;

bool operator < ( const node & d1, const node & d2 ) {

return d1.w > d2.w; //priority\_queue总是将最大的元素出列,即最小堆

}

//这个还是要会的!!!

void dijkstra();

int main(void)

{

scanf("%d %d",&n,&m);

//我也不知道...

v.clear();

v.resize(n+1);

for(int i=0;i<m;i++)

{

int a,b,c;

struct node p;

scanf("%d %d %d",&a,&b,&c);

p.v=b;

p.w=c;

v[a].push\_back(p);

}

memset(vis,0,sizeof(vis));

dijkstra();

printf("%d\n",dist[n]);

return 0;

}

void dijkstra()

{

struct node S; //源点

S.v=1;

S.w=0;

dist[S.v]=0;

for(int i=2;i<=n;i++)

dist[i]=INF;

pq.push(S);

while(!pq.empty())

{

struct node p=pq.top();

pq.pop();

if(vis[p.v]==true)

continue;

vis[p.v]=true;

for(int i=0,j=(int)v[p.v].size();i<j;i++)

{

int V=v[p.v][i].v;

if(vis[V]==false && dist[p.v]+v[p.v][i].w<dist[V])

{

dist[V]=dist[p.v]+v[p.v][i].w;

struct node q;

q.v=V;

q.w=dist[V];

pq.push(q);

}

}

}

}

//3284K 1485MS 一发AC

//和之前写的dijktra不同点:priority\_queue模拟堆,重载了<,最小堆

//if(vis[p.v]==true) continue;的优化

Floyd算法

G[i][j]=min(G[i][j],G[i][k]+G[k][j])，k范围为1-N

G[i][j]=INF，G[i][i]=0

负环条件：G[i][i]<0

题意：给出m对牛的相互关系，求有多少个牛排名是确定的。

用floyed求传递闭包。如果一个牛和其余的牛关系都是确定的，那么这个牛的排名就是确定的了。

void Floyd()

{

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]);

}

#include <iostream>

#include <cstdio>

#define MAX 10

#define INF 0xfffffff

using namespace std;

int G[MAX][MAX];

void Floyd();

int N,M;

int main(void)

{

scanf("%d %d",&N,&M);

for(int i=1;i<=N;i++)

for(int j=1;j<=N;j++)

{

if(i==j)

G[i][j]=0;

else

G[i][j]=INF;

}

for(int i=0;i<M;i++)

{

int u,v;

scanf("%d %d",&u,&v);

G[u][v]=1;

}

Floyd();

int ans=0;

for(int i=1;i<=N;i++)

{

int degree=0;

for(int j=1;j<=N;j++)

{

if(i==j)

continue;

if(G[i][j]!=INF || G[j][i]!=INF)

degree++;

}

if(degree==N-1)

ans++;

}

printf("%d\n",ans);

return 0;

}

void Floyd()

{

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]);

}

//228K 32MS

//求有向图的传递闭包

//求传递闭包,转变为图,即求任意两点是否连通

//考虑每个点的出度和入度,若等于N-1则可以确定

//所以如果这个点到另一个点的距离为INF,则不可以确定

DisjointSet

void Union(int p,int q)

{

int i=root(p);

int j=root(q);

if(i==j)

return;

Id[i]=j;

//加上对于根的某属性的修改

}

int root(int i)

{

//为了记录i的爸爸

int temp=Id[i];

if(temp==i)

return i;

Id[i]=root(temp);

//加上对某个结点的某个属性更新

return Id[i];

}

题意：给出n对关系，每对关系表示a，b是情侣，问其中是否有同性恋

#include <iostream>

#include <cstdio>

#include <cstring>

#define MAX 10

using namespace std;

void Union(int p,int q);

int root(int i);

int Id[MAX];

int relation[MAX]; //为1则和根异性,0则为同性

bool ans=false;

int main(void)

{

int N;

scanf("%d",&N);

int cnt=0;

while(cnt<N)

{

cnt++;

ans=false;

int n,m;

scanf("%d %d",&n,&m);

memset(relation,0,sizeof(relation));

for(int i=0;i<=n;i++)

Id[i]=i;

for(int i=0;i<m;i++)

{

int a,b;

scanf("%d %d",&a,&b);

if(root(a)==root(b)) //如果同根且relation值一样则是同性

{

if(relation[a]!=(relation[b]+1)%2) ans=true;

}

else

Union(a,b);

}

printf("Scenario #%d:\n",cnt);

if(ans==true)

printf("Suspicious bugs found!\n");

else

printf("No suspicious bugs found!\n");

printf("\n");

}

return 0;

}

void Union(int p,int q)

{

int i=root(p);

int j=root(q);

if(i==j)

return;

Id[i]=j;

relation[i]=(relation[p]-relation[q]+1)%2; //why

//根的relation一定为0

//现在i成为了j的子树,所以要更新i的relation值

//那p,q是异性,那就要修改i的relation值,i相对于j的性别就是

//!(rank[p]^rank[q]),异或相同为0

//即p,q同性那relaition[i]为1,p,q异性的话本来relaition[i]就是0

//接着就靠root来改接下来的结点的relation值了,只要保证在下一次Union之前root

}

int root(int i)

{

//和1988的t意义一样,为了记录i的爸爸

int temp=Id[i];

if(temp==i)

return i;

Id[i]=root(temp);

relation[i]=(relation[i]+relation[temp])%2; //why

//或者写成relation[i] = relation[i]^relation[temp];

//relation[temp]是i的父亲与最新的根结点的关系

//现在知道了relation[temp]和自己跟父亲的关系(和temp的关系)relation[i],

//要更新自己现在和最新的根结点的关系,因为路径压缩连到根

//可以列出来这是个异或关系

return Id[i];

}

//188K 829MS

题目大意：开始有N堆砖块，编号为1,2....N，每堆都只有一个。之后可以进行两种操作：   
（1）M X Y 将编号为X的砖块所在的那堆砖拿起来放到编号为Y的砖块所在的堆上；

（2）C X 查询编号为X的砖块所在的堆中，在砖块X下方的所有砖块的数目

#include <iostream>

#include <cstdio>

#define MAX 10

using namespace std;

int Id[MAX];

int under[MAX]; //记录该编号的方块下面有多少方块

int sum[MAX]; //若Id[i]=i,sum[i]记录该堆方块的个数

void Union(int p,int q);

int root(int i);

int main(void)

{

for(int i=0;i<MAX;i++)

{

sum[i]=1;

Id[i]=i;

under[i]=0;

}

int p;

scanf("%d",&p);

for(int i=1;i<=p;i++)

{

char op[20];

scanf("%s",op);

if(op[0]=='M')

{

int a,b;

scanf("%d %d",&a,&b);

Union(a,b); //把a放到b的上面

}

else

{

int a;

scanf("%d",&a);

root(a);

printf("%d\n",under[a]);

}

}

return 0;

}

void Union(int p,int q)

{

int i=root(p);

int j=root(q);

if(i==j)

return;

Id[i]=j;

under[i]=sum[j]; //因为i是根,永远这时候under[i]=0

sum[j]+=sum[i];

}

int root(int i)

{

// if(Id[i]==i)

// return i;

// int t=root(Id[i]);

// //变量t的意义

// //明明可以写成Id[i]=root(Id[i]];

// //先拿t保存一下,然后因为要更新under数组,如果修改了根节点,无法一路累加under

// //因为修改完了Id[i]就是根节点为0

// under[i]+=under[Id[i]];

// Id[i]=t;

int temp=Id[i];

if(temp==i)

return i;

Id[i]=root(temp);

under[i]+=under[temp];

return Id[i];

//under是一个相对值,相对于根的值

//所以对于i来说,temp是他的爸爸,under[i]本来记录的值是相对于temp(i原来的爸爸)的under值

//temp在之前更新完了,那i现在相对于现在爸爸(即根)的under值,就是under[i]+under[temp]

}

//516K 266MS

//还是有点混乱

//核心思想是under数组在堆合并和路径压缩的时候都要更新

//堆合并的时候很简单只需要加上sum

//而路径压缩的时候要加上一路到根的under之和

KMP

#include <iostream>

#include <cstdio>

#include <string>

#define MAX 20

using namespace std;

void getNext(string s);

int KMP(string s,string p);

int KMPnext[MAX];

int main(void)

{

string s;

string p;

cin>>s;

cin>>p;

getNext(p);

cout<<KMP(s,p)<<endl;

return 0;

}

//getNext思路:具体怎么还不太懂,背吧

void getNext(string p)

{

int len=(int)p.size();

KMPnext[0]=-1;

int k=-1;

int j=0;

while(j<len)

{

if(k==-1 || p[j]==p[k])

{

k++;

j++;

KMPnext[j]=k;

}

else

k=KMPnext[k]; //递归的去找

}

for(int i=0;i<len;i++)

printf("%d ",KMPnext[i]);

}

int KMP(string s,string p)

{

int i=0;

int j=0;

int sLen=(int)s.size();

int pLen=(int)p.size();

while(i<sLen && j<pLen)

{

if(j==-1 || s[i]==p[j])

{

i++;

j++;

}

else

j=KMPnext[j];

}

if(j==pLen)

return i-j;

else

return -1;

}

//KMP核心

//假设现在文本串S匹配到 i 位置，模式串P匹配到 j 位置

//如果j = -1，或者当前字符匹配成功（即S[i] == P[j]），都令i++，j++，继续匹配下一个字符；

//如果j != -1，且当前字符匹配失败（即S[i] != P[j]），则令 i 不变，j = next[j]。此举意味着失配时，模式串P相对于文本串S向右移动了j - next [j] 位。

//换言之，当匹配失败时，模式串向右移动的位数为：失配字符所在位置 - 失配字符对应的next 值，即移动的实际位数为：j - next[j]，且此值大于等于1。

//求子串匹配主串次数 next数组同上

long long KMP\_count(char\* s,char\* t)

{

int i=0,j=0;

long long ans=0;

int lens=(int)strlen(s);

int lent=(int)strlen(t);

while(i<lens)

{

if(s[i]==t[j])

{

++i,++j;

if(j==lent)

++ans;

}

else if(KMPnext[j]==-1)

++i;

else

j=KMPnext[j];

}

return ans;

}

//exKMP,extand[i]表示主串[i...len-1]匹配子串多长长度

#include<iostream>

using namespace std;

const int N = 101010;

int KMPnext[N],extand[N];

void getKMPnext(char \*T){// KMPnext[i]: 以第i位置开始的子串 与 T的公共前缀

int i,length =(int)strlen(T);

KMPnext[0] = length;

for(i = 0;i<length-1 && T[i]==T[i+1]; i++);

KMPnext[1] = i;

int a = 1;

for(int k = 2; k < length; k++)

{

int p = a+KMPnext[a]-1, L = KMPnext[k-a];

if( (k-1)+L >= p )

{

int j = (p-k+1)>0? (p-k+1) : 0;

while(k+j<length && T[k+j]==T[j]) j++;

// 枚举(p+1，length) 与(p-k+1,length) 区间比较

KMPnext[k] = j, a = k;

}

else KMPnext[k] = L;

}

}

void getextand(char \*S,char \*T)

{

memset(KMPnext,0,sizeof(KMPnext));

getKMPnext(T);

int Slen = (int)strlen(S);

int Tlen = (int)strlen(T);

int a = 0;

int MinLen = Slen>Tlen?Tlen:Slen;

while(a<MinLen && S[a]==T[a]) a++;

extand[0] = a, a = 0;

for(int k = 1; k < Slen; k++)

{

int p = a+extand[a]-1, L = KMPnext[k-a];

if( (k-1)+L >= p )

{

int j = (p-k+1)>0? (p-k+1) : 0;

while(k+j<Slen && j<Tlen && S[k+j]==T[j] ) j++;

extand[k] = j;a = k;

}

else extand[k] = L;

}

}

int main()

{

char s[N],t[N];

//T匹配S,S为主串

while(~scanf("%s %s",s,t)){

getextand(s,t);

for(int i = 0; i < strlen(t); i++)

printf("%d ",KMPnext[i]);

puts("");

for(int i = 0; i < strlen(s); i++)

printf("%d ",extand[i]);

puts("");

}

}

KMPnext数组求循环节

#include <iostream>

#include <cstdio>

#include <cstring>

#define MAX 20

using namespace std;

void getNext(char \*p);

int KMPnext[MAX];

int main(void)

{

char p[MAX];

int cnt=1;

while(1)

{

int n;

scanf("%d",&n);

if(n==0)

break;

scanf("%s",p);

getNext(p);

printf("Test case #%d\n",cnt++);

for(int i=2;i<=n;i++) //注意范围

{

if(KMPnext[i]!=0 && i%(i-KMPnext[i])==0)

{

int ans=i/(i-KMPnext[i]);

printf("%d %d\n",i,ans);

}

}

printf("\n");

}

return 0;

}

void getNext(char \*p)

{

int len=(int)strlen(p);

KMPnext[0]=-1;

int k=-1;

int j=0;

while(j<len) //根据需要多算一位next值

{

if(k==-1 || p[j]==p[k])

{

k++;

j++;

KMPnext[j]=k;

}

else

k=KMPnext[k]; //递归的去找

}

}

//5028K 188MS

//总结一下,如果对于next数组中的i,符合i%(i-next[i])==0 && next[i]!=0,则说明字符串循环

//循环节长度为:i-next[i]

//循环次数为:i/(i-next[i])

Kruskal算法

vector存所有的边

排序，每次取出最小的边，判断这条边的两个端点是否形成环，直到收入了n-1条边

#include <iostream>

#include <cstdio>

#include <algorithm>

#include <vector>

#define MAX 10

using namespace std;

struct node{

int u,v;

int w;

};

bool comp(struct node a,struct node b)

{

return a.w<b.w;

}

int Kruskal(int n);

void Union(int p,int q);

int root(int i);

bool connected(int p,int q);

vector<struct node> G;

int Id[MAX];

int main(void)

{

int N;

while(cin>>N)

{

G.clear();

for(int i=1;i<=N;i++)

{

for(int j=1;j<=N;j++)

{

struct node p;

if(i==j)

{

scanf("%d",&p.w);

continue;

}

p.u=i;

p.v=j;

scanf("%d",&p.w);

G.push\_back(p);

}

}

sort(G.begin(),G.end(),comp);

printf("%d\n",Kruskal(N));

}

return 0;

}

void Union(int p,int q)

{

int i=root(p);

int j=root(q);

if(i==j)

return;

Id[i]=j;

}

bool connected(int p,int q)

{

return root(p)==root(q);

}

int root(int i)

{

// if(Id[i]==i)

// return i;

// Id[i]=root(Id[i]);

// return Id[i];

while(Id[i]!=i)

{

Id[i]=Id[Id[i]];

i=Id[i];

}

return i;

}

int Kruskal(int n)

{

int ans=0;

int cnt=0;

for(int i=0;i<=n;i++)

Id[i]=i;

for(int i=0;i<G.size();i++)

{

struct node p=G[i];

if(connected(p.u,p.v)==false)

{

Union(p.u,p.v);

ans+=p.w;

cnt++;

}

if(cnt==n-1)

break;

}

return ans;

}

Edmonds\_Karp算法

bfs找到一条源到汇的路，并记录下路径，求出路径上最短的边，插入反向边，并渐小原来的边，大小和那条最短的边相同

最外面再套个循环，直到更新后的图没有源到汇的路，sum求和即为答案

邻接矩阵存储，注意重边

#include <iostream>

#include <cstdio>

#include <queue>

#include <cstring>

#define MAX 300

using namespace std;

int G[MAX][MAX];

bool vis[MAX];

int pre[MAX];

int m,n;

unsigned int Edmonds\_Karp();

int main(void)

{

while(scanf("%d %d",&m,&n)!=EOF)

{

memset(G,0,sizeof(G)); //WA了一小时,因为这个放在的循环外,mdzz

for(int i=1;i<=m;i++)

{

int a,b,c;

scanf("%d %d %d",&a,&b,&c);

G[a][b]+=c; //可能重边

}

int ans=0;

int temp;

while(1)

{

temp=Edmonds\_Karp();

if(temp==0)

break;

ans+=temp;

}

printf("%d\n",ans);

}

return 0;

}

//1为源,n为汇

unsigned int Edmonds\_Karp()

{

//初始化

memset(vis,0,sizeof(vis));

memset(pre,0,sizeof(pre));

//BFS

deque<int> Q;

int S=1;

Q.push\_back(S);

vis[S]=true;

pre[S]=0;

bool ifRoad=false;

while(!Q.empty())

{

int v=Q.front();

Q.pop\_front();

for(int i=1;i<=n;i++)

{

if(G[v][i]>0 && vis[i]==false)

{

vis[i]=true;

pre[i]=v;

//找到了源到汇到路

if(i==n)

{

ifRoad=true;

Q.clear(); //queue没有clear

break;

}

else

Q.push\_back(i);

}

}

}

//没有路

if(ifRoad==false)

return 0;

int minRoad=0x7ffffff;

int v=n;

while(pre[v]!=0)

{

minRoad=min(minRoad,G[pre[v]][v]);

v=pre[v];

}

//插反向边

v=n;

while(pre[v]!=0)

{

G[pre[v]][v]-=minRoad;

G[v][pre[v]]+=minRoad;

v=pre[v];

}

return minRoad;

}

Tarjan算法

vector模拟邻接表

对每个点都需要进行Tarjan算法，判断一下其是否在之前的点点Tarjan算法中已经做过了没，if(!dfn(i))，不然如果这个图并非连通，那有些点就根本没有做

dfn[i]=low[i]=idx++，将源点进栈，访问他的邻接点，是否访问过用dfn数组判断，如果没有访问过，递归调用Tarjan，并最后让v把low[v]带出来更新low[u]，如果访问过还在栈里，用low[v]更新low[u]，邻接点全访问过后，如果发现dfn[u]==low[u]，则从栈里开始弹出结点，直到弹出的是u，弹出来的这一部分就是强连通分量

题目大意：在一个牧群中，有N个奶牛，给定M对关系（A,B）表示A仰慕B，而且仰慕关系有传递性，问被所有奶牛（除了自己）仰慕的奶牛个数

技巧：求出强连通分量后进行染色，缩点，形成DAG，再利用DAG的性质

#include <iostream>

#include <cstdio>

#include <vector>

#include <stack>

#include <cstring>

#define MAX 10

using namespace std;

void Tarjan(int u); //u当前结点

vector<vector<int>> G;

stack<int> S;

int dfn[MAX];

int low[MAX];

bool vis[MAX]; //是否在栈中

int color[MAX];

int idx=1; //记录dfs的编号

int clr=0; //染色标记

int main(void)

{

int n,m;

while(scanf("%d %d",&n,&m)!=EOF)

{

G.clear();

G.resize(n+1);

for(int i=0;i<m;i++)

{

int a,b;

scanf("%d %d",&a,&b);

G[a].push\_back(b);

}

//初始化

memset(vis,0,sizeof(vis));

memset(dfn,0,sizeof(dfn));

memset(low,0,sizeof(low));

memset(color,0,sizeof(color));

clr=0;

idx=1;

//WA的教训

for(int i=1;i<=n;i++)

{

if(!dfn[i])

Tarjan(i);

}

// 看一下染色结果

// for(int i=1;i<=n;i++)

// printf("%d %d\n",i,color[i]);

//把强连通分量缩点,clr的值为强连通分量的个数

//用数组ans保存每个颜色的强连通分量的出度

//定理:有向无环图中唯一出度为0的点，一定可以由任何点出发均可达

//枚举所有的边

int ans[MAX];

memset(ans,0,sizeof(ans));

for(int i=0;i<G.size();i++)

for(int j=0;j<G[i].size();j++)

{

if(color[i]==color[G[i][j]])

continue;

else

ans[color[i]]++;

}

int cnt=0;

int idx=-1;

for(int i=1;i<=clr;i++)

{

if(ans[i]==0)

{

cnt++;

idx=i;

}

}

if(cnt==1)

{

int sum=0;

for(int i=1;i<=n;i++)

{

if(color[i]==idx)

sum++;

}

printf("%d\n",sum);

}

else

printf("0\n");

}

return 0;

}

void Tarjan(int u)

{

dfn[u]=low[u]=idx++;

S.push(u);

vis[u]=true;

for(int i=0;i<G[u].size();i++)

{

int v=G[u][i];

if(!dfn[v]) //如果没访问过

{

Tarjan(v);

low[u]=min(low[u],low[v]);

}

else if(vis[v]==true) //如果访问过还在栈里

{

low[u]=min(low[u],low[v]);

}

}

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

{

int v;

clr++;

do

{

v=S.top();

S.pop();

color[v]=clr;

vis[v]=false;

// printf("%d ",v);

}while(u!=v);

// printf("\n");

}

}

//多次Tarjan

//定理:有向无环图中唯一出度为0的点，一定可以由任何点出发均可达

//由于无环，所以从任何点出发往前走，必然终止于一个出度为0的点

//1. 求出所有强连通分量

//2. 每个强连通分量缩成一点，则形成一个有 向无环图DAG。

//3. DAG上面如果有唯一的出度为0的点,则该点 能被所有的点可达。

//那么该点所代表的连通分 量上的所有的原图中的点，都能被原图中的所有点可达，则该连通分量的点数，就是答案。

//4. DAG上面如果有不止一个出度为0的点，则这些点互相不可达，原问题无解，答案为0

TopSort

思想：维护一个入度为0的队列，依次出队，出队以后把邻接点的入度减1，如果减为0入队，直到队列空

判断有向图环：队列空后，判断每个点的入度，检查每个结点入度，如果入度有不为0的，则有环

可判断DAG是否任意两点都可到达，考虑DAG是一条长链，若存在分叉，那分叉的那两个点互相不可达到，即判断队列里是否超过两个元素。

#include <iostream>

#include <cstdio>

#include <vector>

#include <queue>

#include <cstring>

#define MAX 20

using namespace std;

vector< vector<int> > v;

int in[MAX];

void Topsort(int n,int m);

int main(void)

{

int n,m; //顶点,边数,顶点从1开始

scanf("%d %d",&n,&m);

memset(in,0,sizeof(in));

v.resize(n+1);

for(int i=1;i<=m;i++)

{

int a,b;

scanf("%d %d",&a,&b);

v[a].push\_back(b);

in[b]++;

}

Topsort(n,m);

return 0;

}

void Topsort(int n,int m)

{

queue<int> q;

for(int i=1;i<=n;i++)

{

if(in[i]==0)

q.push(i); //入度为0的结点入队

}

while(!q.empty())

{

int u=q.front();

printf("%d ",u);

q.pop();

for(int i=0;i<v[u].size();i++)

{

int a=v[u][i];

in[a]--; //修改邻接点入度

if(in[a]==0)

q.push(a);

}

}

for(int i=1;i<=n;i++) //检查每个点入度是否为0,如果不为0,则存在环

if(in[i]!=0) //判断有向图是否有环

{

printf("loop\n");

break;

}

}