1. 最短路算法

1.dijkstra算法(未优化O(V^2))

#include <cstdio>

#include <queue>

using namespace std;

const int maxn = 1000;

const int INF = 0x3fffffff;

int graph[maxn][maxn],dist[maxn],flag[maxn];

//dijkstra算法，求单源最短路，0(n^2),不能处理负边权

//s为起始点,n为节点个数

void dijkstra(int s,int n)

{

fill(dist,dist+maxn,INF);

fill(flag,flag+maxn,0);

dist[s] = 0;

flag[s] = 1;

int sm,pre = s;

//找出另外n-1个节点到s的最短距离

for( int i= 0 ; i < n-1 ; ++i ){

sm = INF;

//更新最短距离,节点编号从1~n

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

if( !flag[i] ) dist[i] = min(dist[i],graph[i][pre] );

}

//找到最短距离的点

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

if( !flag[i] && sm > dist[i] ){

sm = dist[i];

pre = i;

}

}

flag[pre] = 1;

}

}

2.dijkstra算法(优先队列优化O(ElogV))

#include <cstdio>

#include <queue>

using namespace std;

const int maxn = 1000;

const int INF = 0x3fffffff;

//dijkstra算法优先队列优化,0(ElogV)

struct Edge{

int node,len;

Edge\*next;

}m\_edge[maxn];

Edge\*head[maxn];

int Ecnt,dist[maxn],flag[maxn];

void init()

{

Ecnt = 0;

fill( head , head+maxn , (Edge\*)0 );

}

void mkEdge( int a , int b , int c )

{

m\_edge[Ecnt].node = b;

m\_edge[Ecnt].len = c;

m\_edge[Ecnt].next = head[a];

head[a] = m\_edge+Ecnt++;

}

struct HeapNode{

int d,u;

HeapNode(int \_d,int \_u){

d = \_d; u = \_u;

}

bool operator < (const HeapNode&a) const{

return d > a.d;

}

};

void dijkstra(int s)

{

priority\_queue<HeapNode>Q;

fill( dist , dist+maxn , INF );

fill( flag , flag+maxn , 0 );;

dist[s] = 0;

Q.push(HeapNode(0,s));

while( !Q.empty() ){

HeapNode x = Q.top();

Q.pop();

int u = x.u;

if(flag[u]) continue;

flag[u] = 1;

for( Edge\*p = head[u] ; p ; p = p->next ){

int v = p->node;

if( dist[v] > dist[u]+p->len ){

dist[v] = dist[u]+p->len;

Q.push(HeapNode(dist[v],v));

}

}

}

}

3.Bellman\_Ford算法(O(VE))

#include <cstdio>

#include <queue>

using namespace std;

const int N = 1000;

const int INF = 0x3fffffff;

struct Edge{

int u,v,w;

}e[N\*N];

int dist[N],pre[N],Ecnt;

//Bellman\_Ford算法，可以处理负权边，但不能处理负环，0(EV)

void mkEdge( int a , int b , int c )

{

e[Ecnt].u = a;

e[Ecnt].v = b;

e[Ecnt].w = c;

++Ecnt;

}

bool Bellman\_Ford(int s,int n)

{

fill( dist , dist+N , INF );

dist[s] = 0;

//松弛n-1次

for( int i = 0 ; i < n-1 ; ++i ){

for( int j = 0 ; j < Ecnt ; ++j ){

int u = e[j].u;

int v = e[j].v;

int w = e[j].w;

if( dist[v] < dist[u]+w ){

dist[v] = dist[u]+w;

//记录前驱节点

pre[v] = u;

}

}

}

//再松弛一次，若能产生更小的值，则存在负环

for( int j = 0 ; j < Ecnt ; ++j ){

int u = e[j].u;

int v = e[j].v;

int w = e[j].w;

if( dist[v] < dist[u]+w) return false;

}

return true;

}

4.Bellman\_Ford(队列优化)

#include <cstdio>

#include <queue>

using namespace std;

const int N = 1000;

const int M = 1000;

const int INF = 0x3fffffff;

//最坏的时间复杂度未0(VE)

struct Edge{

int node,len;

Edge\*next;

}m\_edge[M\*2];

Edge\*head[N];

int Ecnt,dist[N],vis[N];

void init()

{

Ecnt = 0;

fill( head , head+N , (Edge\*)0 );

}

void mkEdge( int a , int b , int c )

{

m\_edge[Ecnt].node = b;

m\_edge[Ecnt].len = c;

m\_edge[Ecnt].next = head[a];

head[a] = m\_edge+Ecnt++;

}

void spfa( int n )

{

fill( dist , dist+N , INF );

fill( vis , vis+N , 0 );

queue<int>point;

point.push(n);

vis[n] = 1;

dist[n] = 0;

while( !point.empty() ){

int s = point.front();

point.pop();

vis[s] = 0;

for( Edge\*p = head[s] ; p ; p = p->next ){

int t = p->node;

if( dist[t] > dist[s]+p->len ){

dist[t] = dist[s]+p->len;

if( !vis[t] ){

point.push(t);

vis[t] = 1;

}

}

}

}

}

5.floyed算法(O(V^3))

#include <cstdio>

#include <queue>

using namespace std;

const int N = 1000;

const int M = 1000;

const int INF = 0x3ffffff;

//floyed算法，求任意两点间的最短距离，0(V^3)

int dist[N][N],graph[N][N];

void floyed( int n )

{

//dist[i][i]初始化为0,dist[i][j]初始化为INF

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

for( int j = 0 ; j < n ; ++j ){

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

//注意INF的取值，避免过大越界，过小求不出dist

dist[i][j] = min(dist[i][j],dist[i][k]+dist[k][j]);

}

}

}

1. 最小生成树
2. hdu4607—树的直径

#include <iostream>

#include <cstdio>

using namespace std;

const int N = 20090;

const int M = 100900;

//树的直径:树上的最长简单路

//以某个点u为根搜索，找到到u最长的距离的点，那么改点一定是树中某条最长路的端点

//再以该点为根节点，找到到改点最长的距离

struct Edge{

int node,len;

Edge\*next;

}m\_edge[M\*2];

Edge\*head[M];

int Ecnt,vis[M],dp[M];

void init()

{

Ecnt = 0;

fill( head , head+M , (Edge\*)0 );

}

void mkEdge(int a,int b)

{

m\_edge[Ecnt].node = b;

m\_edge[Ecnt].len = 1;

m\_edge[Ecnt].next = head[a];

head[a] = m\_edge+Ecnt++;

}

void dfs(int u)

{

vis[u] = 1;

for( Edge\*p = head[u] ; p ; p = p->next ){

int v = p->node;

if( vis[v] ) continue;

dp[v] = dp[u]+p->len;

dfs(v);

}

}

int main()

{

int T;

scanf("%d",&T);

while( T-- ){

init();

int n,m,a,b;

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

for( int i = 0 ; i < n-1 ; ++i ){

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

mkEdge(a,b);

mkEdge(b,a);

}

int node,ans = 0;

memset(dp,0,sizeof(dp));

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

dfs(1);

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

if( dp[i] > ans ){

ans = dp[i];

node = i;

}

}

memset(dp,0,sizeof(dp));

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

dfs(node);

ans = 0;

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

ans = max(ans,dp[i]);

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

scanf("%d",&a);

if( a <= ans+1 ) printf("%d\n",a-1);

else printf("%d\n",2\*a-2-ans);

}

}

return 0;

}

1. 树的重心—poj1655

#include <iostream>

#include <cstdio>

#include <algorithm>

using namespace std;

const int N = 20090;

const int M = 100000;

//树的重心:删掉这个节点后将树分成几个部分，使得这几个部分中点个数的最大值最小

//以每个点作为根节点搜一次，求得改点的子树中点数的最大值，然后求其中的最小值

//但是复杂度为0(n^2)肯定TLE

//其实只要以某个节点u为根搜一次就可以了。

//当向下搜到某个节点v时，我们可以求出节点v的子树中最大的节点数

//这时只要再和节点v的非子树节点数比较就行了，假如节点v的子树节点数为m,节点数为n

//显然节点v的非子树节点数为n-m-1，这样就相当于求出了去掉v后，子树中节点数最大的值

struct Edge{

int node;

Edge\*next;

}m\_edge[N\*2];

Edge\*head[N];

int Ecnt,vis[N],son[N];

int Num,Size; //树的重心的编号，划分后的最大值

void init()

{

Ecnt = 0; Num = 0; Size = INT\_MAX;

fill( head , head+N , (Edge\*)0 );

fill( vis , vis+N , 0 );

}

void mkEdge( int a , int b )

{

m\_edge[Ecnt].node = b;

m\_edge[Ecnt].next = head[a];

head[a] = m\_edge+Ecnt++;

}

int n; //节点个数

void dfs( int u )

{

vis[u] = 1;

son[u] = 0;

int temp = 0;

for( Edge\*p = head[u] ; p ; p = p->next ){

int v = p->node;

if( !vis[v] ){

dfs(v);

//子树中节点数的最大值

temp = max(temp,son[v]+1);

//节点u的子树树

son[u] += son[v]+1;

}

}

//与非子树节点数比较

temp = max(temp,n-son[u]-1);

if( temp < Size || temp == Size&&u < Num ){

Size = temp;

Num = u;

}

}

int main()

{

int T;

scanf("%d",&T);

while( T-- ){

init();

int a,b;

scanf("%d",&n);

for( int i = 0 ; i < n-1 ; ++i ){

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

mkEdge(a,b);

mkEdge(b,a);

}

dfs(1);

printf("%d %d\n",Num,Size);

}

return 0;

}

1. 最小生成树kruskal算法(O(ElogE))

#include <iostream>

#include <stack>

#include <vector>

using namespace std;

const int INF = 0x3fffffff;

const int N = 50030;

const int M = 100300;

typedef int MyType;

struct Edge{

int u,v;

MyType len;

bool operator < (const Edge&a)const{

return a.len > len;

}

}edge[M];

int Ecnt,parent[N];

void init()

{

Ecnt = 0;

memset(parent,0,sizeof(parent));

}

int find(int a)

{

while(a != parent[a]){

parent[a] = parent[parent[a]];

a = parent[a];

}

return a;

}

void merge(int a,int b)

{

int i = find(a);

int j = find(b);

if( i != j ) parent[i] = j;

}

MyType kruskal(int n,int m)

{

sort(edge,edge+m);

int Count = 0;

MyType len = 0;

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

int u = edge[i].u;

int v = edge[i].v;

if(find(u) != find(v)){

len += edge[i].len;

Count++;

merge(u,v);

}

if( Count == n-1 ) break;

}

return len;

}

int main()

{

return 0;

}

1. 最小树形图

#include <iostream>

#include <stack>

#include <vector>

using namespace std;

const int INF = 0x3fffffff;

const int N = 50030;

const int M = 100300;

typedef int MyType;

/\*

最小树形图:类似树的有向图

1.恰好有一个入度为0的点，为根节点

2.其他节点的入度均为1

3.可以从根节点到达所有其他接点

求解:朱刘算法

1.求最短弧集合E0(除根节点外所有点的入边边权最小的边集),若某个点的入边不存在,则最小树形图不存在

2.找E0中的环,缩点。若E0中不存在环,跳到步骤4

3.重新建图,跳到步骤1

4.将缩点展开成环,去掉环中某条边(表达能力不行哦),得到最小树形图

\*/

//求最小树形图的权,不需要将缩点展开

struct Edge//边的权和顶点

{

int u, v;

MyType w;

}edge[N];

int pre[N],id[N],vis[N];

MyType in[N];//存最小入边权,pre[v]为该边的起点

int Directed\_MST(int root,int V,int E)

{

MyType ret = 0;//存最小树形图总权值

while(true)

{

int i;

//1.找每个节点的最小入边,求最短弧集合E0

for( i = 0; i < V; i++)

in[i] = INF;//初始化为无穷大

for( i = 0; i < E; i++)//遍历每条边

{

int u = edge[i].u;

int v = edge[i].v;

if(edge[i].w < in[v] && u != v)//说明顶点v有条权值较小的入边 记录之

{

pre[v] = u;//节点u指向v

in[v] = edge[i].w;//最小入边

}

}

for( i = 0; i < V; i++)//判断是否存在最小树形图

{

if(i == root)

continue;

if(in[i] == INF)

return -1;//除了根以外有点没有入边,则根无法到达它 说明它是独立的点 一定不能构成树形图

}

//2.找环

int cnt = 0;//记录环数

memset(id, -1, sizeof(id));

memset(vis, -1, sizeof(vis));

in[root] = 0;

for( i = 0; i < V; i++) //标记每个环

{

ret += in[i];//记录权值

int v = i;

//向上搜索看是否构成环

while(vis[v] != i && id[v] == -1 && v != root)

{

vis[v] = i;

v = pre[v];

}

//向上搜索找到根节点，肯定不能构成环(根节点入度为0)

if(v != root && id[v] == -1)

{

for(int u = pre[v]; u != v; u = pre[u])

id[u] = cnt;//标记节点u为第几个环

id[v] = cnt++;

}

}

if(cnt == 0)

break; //无环 则break

for( i = 0; i < V; i++)

if(id[i] == -1)

id[i] = cnt++;

//3.建立新图 缩点,重新标记

for( i = 0; i < E; i++)

{

int u = edge[i].u;

int v = edge[i].v;

edge[i].u = id[u];

edge[i].v = id[v];

if(id[u] != id[v])

edge[i].w -= in[v];

}

V = cnt;

root = id[root];

}

return ret;

}

1. uva11354—最小瓶颈路

#include <iostream>

#include <stack>

#include <vector>

using namespace std;

const int INF = 0x3fffffff;

const int N = 50030;

const int M = 100300;

//最小瓶颈路:给定加权无向图的两个节点u,v，求从u到v的一条路径,使得路径上的最长边尽量短

struct Edge{

int u,v,len;

bool operator < (const Edge&a) const{

return a.len > len;

}

}E[M];

int Ecnt,parent[N];

struct NEdge{

int node,len;

NEdge\*next;

}m\_edge[N\*2];

int NEcnt;

NEdge\*head[N];

void mkEdge(int a,int b,int c)

{

m\_edge[NEcnt].node = b;

m\_edge[NEcnt].len = c;

m\_edge[NEcnt].next = head[a];

head[a] = m\_edge+NEcnt++;

}

void init()

{

Ecnt = NEcnt = 0;

fill( head , head+N , (NEdge\*)0 );

for( int i = 0 ; i < N ; ++i ) parent[i] = i;

}

int find(int a)

{

while(a != parent[a]){

parent[a] = parent[parent[a]];

a = parent[a];

}

return a;

}

void merge(int a,int b)

{

int i = find(a);

int j = find(b);

if( i != j ) parent[i] = j;

}

int kruskal( int n , int m )

{

sort(E,E+m);

int len=0,Count=0;

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

int u = E[i].u;

int v = E[i].v;

int L = E[i].len;

if( find(u) != find(v) ){

len += L;

Count++;

mkEdge(u,v,L);

mkEdge(v,u,L);

merge(u,v);

}

if( Count == n-1 ) break;

}

return len;

}

//fa[i]:节点i的父节点编号

//cost[i]:节点i与父节点间的边权

//L[i]:表示节点i的深度

int fa[N],cost[N],L[N];

void dfs( int u , int father )

{

for( NEdge\*p = head[u] ; p ; p = p->next ){

int v = p->node;

if( v != father ){

fa[v] = u;

cost[v] = p->len;

L[v] = L[u]+1;

dfs(v,u);

}

}

}

//无根树转为有根树

void toChangeRoot( int root )

{

fa[root] = root;

cost[root] = L[root] = 0;

dfs(root,-1);

}

//anc[i][j]:节点i的第2^j级祖先编号

//maxcost[i][j]:节点i和它的2^j级祖先间路径上的最大权值

int anc[N][30],maxcost[N][30];

//LCA倍增预处理

void preprocess( int n )

{

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

anc[i][0] = fa[i]; maxcost[i][0] = cost[i];

for( int j = 1 ; (1<<j) < n ; ++j ) anc[i][j] = -1;

}

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

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

int a = anc[i][j-1];

anc[i][j] = anc[a][j-1];

maxcost[i][j] = max(maxcost[i][j-1],maxcost[a][j-1]);

}

}

//查询

int query(int p,int q)

{

int tmp,log;

if(L[p] < L[q]) swap(p,q);

for( log = 1 ; (1<<log) <= L[p] ; log++ ); log--;

int ans = -INF;

for( int i = log ; i >= 0 ; --i )

if(L[p]-(1<<i) >= L[q]){

ans = max(ans,maxcost[p][i]);

p = anc[p][i];

}

if(p == q) return ans;

for( int i = log ; i >= 0 ; --i ){

if(anc[p][i] != -1 && anc[p][i] != anc[q][i]){

ans = max(ans,maxcost[p][i]); p = anc[p][i];

ans = max(ans,maxcost[q][i]); q = anc[q][i];

}

}

ans = max(ans,cost[p]);

ans = max(ans,cost[q]);

return ans;

}

int main()

{

int n,m,cas = 1;

while( ~scanf("%d%d",&n,&m) ){

int a,b,c;

init();

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

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

a--; b--;

E[Ecnt].u = a;

E[Ecnt].v = b;

E[Ecnt++].len = c;

}

kruskal(n,m);

toChangeRoot(0);

preprocess(n);

scanf("%d",&a);

if( cas != 1 ) printf("\n");

while(a--){

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

b--; c--;

printf("%d\n",query(b,c));

}

cas++;

}

return 0;

}

1. csu1965—斯坦纳树

#include <iostream>

#include <cstdio>

using namespace std;

#define move(a)(1<<(a))

typedef long long ll;

typedef pair<int,int>PA;

const int N = 100900;

const int M = 60;

const int INF = 0x3fffffff;

int dp[1<<15][M],dis[M][M];

int st[12]; //存储boss想要哪些员工知道的编号

//斯坦纳树:在平面中给你n个点，问你其中m个点到点p的最小生成树，一般n<=10

//dp[M个必须知道消息的人组成的状态][到老板或某个员工]:最小花费

//将老板看做0点，最后的答案就是dp[(1<<m)-1][0]

//dp[i][j]的松弛:

//dp[i][j] = min{dp[i][j],dp[k][i]+dp[l][i]},其中k和l是对j的一个划分

//dp[i][j] = min{dp[i][j],dp[i][j']+dist[j'][j]},j与j'有边相连

void cal( int n , int m )

{

//初始化

for( int sta = 0 ; sta < move(n) ; ++sta ){

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

dp[sta][i] = INF;

}

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

for( int j = 0 ; j <= n+m ; ++j )

//n个人中的第i个人到第j号人的花费就是st[i]到j间的最小花费

dp[move(i)][j] = dis[st[i]][j];

}

//状态松弛

for( int sta = 1 ; sta < move(n) ; ++sta ){

//是否有子集

if( sta&(sta-1) ){

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

for( int j = sta ; j > 0 ; j = (j-1)&sta ){

if( dp[sta][i] > dp[sta^j][i]+dp[j][i] )

dp[sta][i] = dp[sta^j][i]+dp[j][i];

}

}

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

for( int j = 0 ; j <= n+m ; ++j )

if( dp[sta][i] > dp[sta][j]+dis[j][i] )

dp[sta][i] = dp[sta][j]+dis[j][i];

}

}

}

}

void floyed( int n )

{

for( int k = 0 ; k <= n ; ++k ){

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

for( int j = 0 ; j <= n ; ++j ){

dis[i][j] = min( dis[i][j] , dis[i][k]+dis[k][j] );

}

}

}

}

int main()

{

int T;

scanf("%d",&T);

while( T-- ){

memset( dis , -1 , sizeof(dis) );

int n,m;

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

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

for( int j = 1 ; j <= n ; ++j ){

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

}

}

int a,b;

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

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

dis[0][a] = dis[a][0] = b;

//标记老板想要哪些员工知道消息，老板与这些员工间的连边为花费

st[i] = a;

}

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

if( dis[0][i] == -1 ){

//对于某些不一定要知道消息的员工，老板与他们间的连边为无穷

dis[0][i] = dis[i][0] = INF;

}

}

dis[0][0] = 0;

floyed(n);

cal(m,n-m);

printf("%d\n",dp[move(m)-1][0]);

}

return 0;

}

1. 次小生成树

#include <iostream>

#include <math.h>

#include <string.h>

#include <stdio.h>

using namespace std;

const int MAXN=110;

const int INF=0x3f3f3f3f;

bool vis[MAXN]; //是否将该点放入集合中

int lowc[MAXN]; //剩下的点到vis集合中的最小值

int pre[MAXN]; //未在vis中的点到vis集合中最小权值所对应的点

int Max[MAXN][MAXN];//Max[i][j]表示在最小生成树中从i到j的路径中的最大边权

bool used[MAXN][MAXN]; //保存最小生成树的边

int Prim(int cost[][MAXN],int n)

{

int ans=0;

memset(vis,false,sizeof(vis));

memset(Max,0,sizeof(Max));

memset(used,false,sizeof(used));

vis[0]=true;

pre[0]=-1;

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

{

lowc[i]=cost[0][i];

pre[i]=0;

}

lowc[0]=0;

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

{

int minc=INF;

int p=-1;

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

if(!vis[j]&&minc>lowc[j])

{

minc=lowc[j];

p=j;

}

if(minc==INF)return -1;

ans+=minc;

vis[p]=true;

//保存最小树的边

used[p][pre[p]]=used[pre[p]][p]=true;

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

{

//未在集合vis中的点i到vis集合中的点j之间路径的边的最大权值

if(vis[j]) Max[j][p]=Max[p][j]=max(Max[j][pre[p]],lowc[p]);

if(!vis[j]&&lowc[j]>cost[p][j])

{

lowc[j]=cost[p][j];

pre[j]=p;

}

}

}

return ans;

}

1. 二分匹配
2. 匈牙利算法

#include <iostream>

#include <cstdio>

using namespace std;

typedef long long ll;

const int N = 1000100;

const int M = 1000;

const int INF = 0x3fffffff;

struct Edge{

int node;

Edge\*next;

}m\_edge[N];

int girl[M];

Edge\*head[M];

int Flag[M],Ecnt,cnt;

void init()

{

Ecnt = cnt = 0;

fill( girl , girl+M , 0 );

fill( head , head+M , (Edge\*)0 );

}

//b对g有好感

void mkEdge( int b , int g )

{

m\_edge[Ecnt].node = g;

m\_edge[Ecnt].next = head[b];

head[b] = m\_edge+Ecnt++;

}

bool find( int x )

{

for( Edge\*p = head[x] ; p ; p = p->next ){

int s = p->node; //有好感的女生

if( !Flag[s] ){

Flag[s] = true; //该女生在本轮匹配中被访问

if( girl[s] == 0 || find(girl[s]) ){

//女生没有对象或者另外一个男生能把这个妹纸让给x男

girl[s] = x;

return true;

}

}

}

return false;

}

//构建二分图

void Build()

{

}

void solve( int n )

{

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

fill( Flag , Flag+M , 0 );

if( find(i) ) ++cnt;

}

}

1. KM算法

#include <iostream>

#include <cstring>

#include <cstdio>

using namespace std;

const int MAXN = 305;

const int INF = 0x3f3f3f3f;

int love[MAXN][MAXN]; // 记录每个妹子和每个男生的好感度

int ex\_girl[MAXN]; // 每个妹子的期望值

int ex\_boy[MAXN]; // 每个男生的期望值

bool vis\_girl[MAXN]; // 记录每一轮匹配匹配过的女生

bool vis\_boy[MAXN]; // 记录每一轮匹配匹配过的男生

int match[MAXN]; // 记录每个男生匹配到的妹子 如果没有则为-1

int slack[MAXN]; // 记录每个汉子如果能被妹子倾心最少还需要多少期望值

int N;

bool dfs(int girl)

{

vis\_girl[girl] = true;

for (int boy = 0; boy < N; ++boy) {

if (vis\_boy[boy]) continue; // 每一轮匹配 每个男生只尝试一次

int gap = ex\_girl[girl] + ex\_boy[boy] - love[girl][boy];

if (gap == 0) { // 如果符合要求

vis\_boy[boy] = true;

if (match[boy] == -1 || dfs( match[boy] )) { // 找到一个没有匹配的男生 或者该男生的妹子可以找到其他人

match[boy] = girl;

return true;

}

} else {

slack[boy] = min(slack[boy], gap); // slack 可以理解为该男生要得到女生的倾心 还需多少期望值 取最小值 备胎的样子【捂脸

}

}

return false;

}

int KM()

{

memset(match, -1, sizeof match); // 初始每个男生都没有匹配的女生

memset(ex\_boy, 0, sizeof ex\_boy); // 初始每个男生的期望值为0

// 每个女生的初始期望值是与她相连的男生最大的好感度

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

ex\_girl[i] = love[i][0];

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

ex\_girl[i] = max(ex\_girl[i], love[i][j]);

}

}

// 尝试为每一个女生解决归宿问题

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

fill(slack, slack + N, INF); // 因为要取最小值 初始化为无穷大

while (1) {

// 为每个女生解决归宿问题的方法是 ：如果找不到就降低期望值，直到找到为止

// 记录每轮匹配中男生女生是否被尝试匹配过

memset(vis\_girl, false, sizeof vis\_girl);

memset(vis\_boy, false, sizeof vis\_boy);

if (dfs(i)) break; // 找到归宿 退出

// 如果不能找到 就降低期望值

// 最小可降低的期望值

int d = INF;

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

if (!vis\_boy[j]) d = min(d, slack[j]);

for (int j = 0; j < N; ++j) {

// 所有访问过的女生降低期望值

if (vis\_girl[j]) ex\_girl[j] -= d;

// 所有访问过的男生增加期望值

if (vis\_boy[j]) ex\_boy[j] += d;

// 没有访问过的boy 因为girl们的期望值降低，距离得到女生倾心又进了一步！

else slack[j] -= d;

}

}

}

// 匹配完成 求出所有配对的好感度的和

int res = 0;

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

res += love[ match[i] ][i];

return res;

}

int main()

{

while (~scanf("%d", &N)) {

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

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

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

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

}

return 0;

}

1. 稳定婚姻问题

#include <cstdio>

#include <queue>

using namespace std;

const int maxn = 1000+10;

int pref[maxn][maxn],order[maxn][maxn],Next[maxn];

int future\_husband[maxn],future\_wife[maxn];

queue<int>q; //未订婚的男士队列

void engage(int man,int woman)

{

int m = future\_husband[woman];

if( m ){ //女士有现任未婚夫m

future\_wife[m] = 0; //抛弃m

q.push(m); //m加入未订婚男士队列

}

future\_wife[man] = woman;

future\_husband[woman] = man;

}

int main()

{

int T;

scanf("%d",&T);

while(T--){

int n;

scanf("%d",&n);

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

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

scanf("%d",&pref[i][j]); //编号为i的男士第j喜欢的人

Next[i] = 1; //接下来向排名为1的女士求婚

future\_wife[i] = 0; //没有未婚妻

q.push(i);

}

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

for( int j = 1 ; j <= n ; ++j ){

int x;

scanf("%d",&x);

order[i][x] = j; //在编号为i的女士心目中，编号为x的男士的排名

}

future\_husband[i] = 0; //没有未婚夫

}

while(!q.empty()){

int man = q.front(); q.pop();

int woman = pref[man][Next[man]++];

if( !future\_husband[woman] )

engage(man,woman); //女士没有未婚夫，直接订婚

else if(order[woman][man] < order[woman][future\_husband[woman]])

engage(man,woman); //替代女士的现任未婚夫

else q.push(man); //男人悲剧

}

while(!q.empty()) q.pop();

for( int i = 1 ; i <= n ; ++i ) printf("%d\n",future\_wife[i]);

if(T) printf("\n");

}

return 0;

}

1. 图的连通性

1.2\_sat

#include <iostream>

#include <cstdio>

using namespace std;

const int N = 20000;

const int M = 100000;

struct Edge{

int node;

Edge\*next;

}m\_edge[M];

Edge\*head[N];

int Ecnt;

void init()

{

Ecnt = 0;

fill( head , head+N , (Edge\*)0 );

}

void mkEdge( int a , int b )

{

m\_edge[Ecnt].node = b;

m\_edge[Ecnt].next = head[a];

head[a] = m\_edge+Ecnt++;

}

int mark[N],S[N],top;

bool sat( int x )

{

if(mark[x^1]) return false;

if(mark[x]) return true;

mark[x] = true;

S[top++] = x;

for( Edge\*p = head[x] ; p ; p = p->next ){

if( !sat(p->node) ) return false;

}

return true;

}

bool solve( int n )

{

memset(mark,0,sizeof(mark));

for( int i = 0 ; i < n\*2 ; i += 2 ){

if( !mark[i]&&!mark[i+1] ){

top = 0;

if( !sat(i) ){

while(top) mark[S[--top]] = false;

if( !sat(i+1) ) return false;

}

}

}

return true;

}

int main()

{

int n,m;

while( ~scanf("%d%d",&n,&m) ){

init();

int a,b;

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

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

a--; b--;

mkEdge(a,b^1);

mkEdge(b,a^1);

}

if( !solve(n) ) printf("NIE\n");

else{

for( int i = 0 ; i < 2\*n ; ++i )

if( mark[i] ) printf("%d\n",i+1);

}

}

return 0;

}

2.割点的求解

#include <iostream>

#include <cstdio>

using namespace std;

typedef long long ll;

const int N = 40900;

const int M = 1090;

const int INF = 0x3fffffff;

//low[u]是从u或者u的子孙出发通过回边可以到达的最低深度优先数

//low[u] = min{ dfn[u] ,

// min{ low[w] | w是u的一个子女 } ,

// min{ dfn[v] | v与u邻接，且(u,v)是一条回边} }

/\*

u是割点的充要条件

u是深度优先搜索生成树的根，则u至少有两个子女

u不是生成树的根，但他有个子女w，使得low[w]>=dfn[u]

去掉割点u后，分成几个联通分量

u是根节点，联通分量个数就是子女个数

u不是根节点，若有d个子女w，使得low[w]>=dfn[u],去掉点u，分成d+1个联通分量

\*/

//向下搜索时，如果顶点v是u的相邻顶点，若v还未被访问，则v是u的儿子节点

//若v被访问了，则v是u的祖先节点，且(u,v)是一条回边

struct Edge{

int node;

Edge\*next;

}m\_edge[N];

Edge\*head[M];

int low[M],dfn[M],Flag[M],Ecnt,cnt;

int subnet[M],son,r;

//son记录根节点的子树个树，r为根节点

//subnet记录对于节点u的子节点v，low[v]>=dfn[u]的个数

void init()

{

r = 1;

Ecnt = cnt = son = 0;

fill( subnet , subnet+M , 0 );

fill( Flag , Flag+M , 0 );

fill( head , head+M , (Edge\*)0 );

}

void mkEdge( int a , int b )

{

m\_edge[Ecnt].node = b;

m\_edge[Ecnt].next = head[a];

head[a] = m\_edge+Ecnt++;

}

void tarjan( int u , int father )

{

Flag[u] = 1;

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

for( Edge\*p = head[u] ; p ; p = p->next ){

int v = p->node;

if( !Flag[v] ){

tarjan(v,u);

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

if( low[v] >= dfn[u] ){

if( u != r ) subnet[u]++;

else son++;

}

}

//v是u的祖先，且(u,v)是一条回边

if( Flag[v] && v != father )

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

}

}

int main()

{

int n,m,cas = 0;

while( ~scanf("%d",&n)&&n ){

scanf("%d",&m);

init();

int node = 0;

node = max(node,max(m,n));

mkEdge(n,m);

mkEdge(m,n);

while(1){

scanf("%d",&n);

if( n == 0 ) break;

scanf("%d",&m);

node = max(node,max(m,n));

mkEdge(m,n);

mkEdge(n,m);

}

tarjan(r,-1);

if( cas != 0 ) printf("\n");

printf("Network #%d\n",++cas);

if( son >= 2 ) subnet[1] = son-1;

int flag = 0;

for( int i = 1 ; i <= node ; ++i ){

if( subnet[i] >= 1 ){

flag = 1;

printf(" SPF node %d leaves %d subnets\n",i,subnet[i]+1);

}

}

if( !flag ) printf(" No SPF nodes\n");

}

return 0;

}

1. 点的双联通分量

#include <iostream>

#include <cstdio>

using namespace std;

typedef long long ll;

typedef pair<int,int>PA;

const int N = 1000900;

const int M = 1090;

const int INF = 0x3fffffff;

const double eps = 1e-8;

const double PI = acos(-1.0);

/\*

求点的双联通分量:

维护一个栈，每找到生成树的边或回边，就把这条边加入栈中

如果遇到某个顶点u的子女顶点v满足dfn[u]<=low[v],将边一条条取出，直到遇到边(u,v)

求边的双联通分量:

先找出所有的割边，标记一下，然后搜一下就行了

\*/

struct Edge{

int node;

Edge\*next;

}m\_edge[N\*2];

Edge\*head[M];

int low[M],dfn[M],Flag[M],Ecnt;

int ret[M][M],color[M],tick[M],num,dep;

//num统计联通分量的数目,tick标记图中双联通分量的点

stack<PA>st;

vector<int>graph[M];

void init()

{

Ecnt = num = dep = 0;

//fill( Flag , Flag+M , 0 );

fill( head , head+M , (Edge\*)0 );

for( int i = 0 ; i < M ; ++i ) graph[i].clear();

}

void mkEdge( int a , int b )

{

m\_edge[Ecnt].node = b;

m\_edge[Ecnt].next = head[a];

head[a] = m\_edge+Ecnt++;

}

void tarjan( int u , int father )

{

Flag[u] = 1;

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

//st.push(u);

for( Edge\*p = head[u] ; p ; p = p->next ){

int v = p->node;

if( ret[u][v] != 2 ){

st.push(PA(u,v));

ret[u][v] = ret[v][u] = 2;

}

if( !Flag[v] ){

tarjan(v,u);

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

//求点的双联通分量

if( low[v] >= dfn[u] ){

num++;

while( !st.empty() ){

int t1 = st.top().first;

int t2 = st.top().second;

graph[num].push\_back(t1);

graph[num].push\_back(t2);

st.pop();

if( (t1 == u && t2 == v) || (t1 == v && t2 == u) ) break;

}

}

}

//v是u的祖先，且(u,v)是一条回边

if( v != father && Flag[v] )

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

}

}

//判断点双联通分量中是否含有奇圈

bool getColor( int u )

{

for( Edge\*p = head[u] ; p ; p = p->next ){

int v = p->node;

//找双联通分量中的点

if( tick[v] ){

if( color[u] == color[v] ) return true;

if( color[v] == -1 ){

color[v] = color[u]^1;

if( getColor(v) ) return true;

}

}

}

return false;

}

//图不一定联通，获取所有的双联通分量

void getPoint( int n )

{

fill( Flag , Flag+M , 0 );

while( !st.empty() ) st.pop();

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

if( !Flag[i] ) tarjan( i , -1 );

}

}

int Cal( int n )

{

fill( Flag , Flag+M , 0 );

for( int i = 1 ; i <= num ; ++i ){

fill( tick , tick+M , 0 );

fill( color , color+M , -1 );

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

tick[graph[i][j]] = 1;

}

int t = graph[i][0];

color[t] = 0;

if( getColor(t) )

for( int j = 1 ; j <= n ; ++j ) Flag[j] += tick[j];

}

int ans = 0;

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

if( !Flag[i] ) ans++;

}

return ans;

}

void Build( int n , int m )

{

int a,b;

memset( ret , 0 , sizeof(ret) );

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

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

ret[a][b] = ret[b][a] = 1;

}

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

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

//两个人不互相仇视

if( ret[i][j] == 0 ){

mkEdge(i,j);

mkEdge(j,i);

}

}

}

}

int main()

{

int n,m;

while( ~scanf("%d%d",&n,&m)&&n&&m ){

init();

Build(n,m);

getPoint(n);

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

}

return 0;

}

1. 点的联通度

#include <iostream>

#include <cstdio>

using namespace std;

typedef long long ll;

typedef pair<int,int> PA;

const int N = 20900;

const int M = 109;

const int INF = 0x3fffffff;

//点的联通度求解

/\*

Merger定理：当G是完全图 k(G) = |V(G)|-1

当G不是完全图 k(G) = min(p(A,B)) 其中点A,b不相邻

p(A,B):点A到B独立轨的最大条数

独立轨的求解:

1.构造一个容量网络，原图G中的每个顶点u变成两个点u,u'，u和u'间连条变，容量为1

若u,v连个点相连，则u'到v建边，v'到u建边，容量都为无穷

2.A'为源点，B为汇点，求从A'到B的最大流

求点的联通度:

枚举所有的源点和汇点，求p(s,t)的最下值

\*/

struct Edge{

int node,c;

Edge\*next;

Edge\*redge;

}m\_edge[N\*2];

Edge\*head[M];

int Ecnt;

void init()

{

Ecnt = 0;

fill( head , head+M , (Edge\*)0 );

}

void mkEdge( int a , int b , int c )

{

int t1 = Ecnt++;

int t2 = Ecnt++;

m\_edge[t1].node = b;

m\_edge[t1].c = c;

m\_edge[t1].next = head[a];

m\_edge[t1].redge = m\_edge+t2;

head[a] = m\_edge+t1;

m\_edge[t2].node = a;

m\_edge[t2].c = 0;

m\_edge[t2].next = head[b];

m\_edge[t2].redge = m\_edge+t1;

head[b] = m\_edge+t2;

}

int L[M]; //层次图

bool bfs( int s , int t )

{

fill( L , L+M , -1 );

queue<int>q;

q.push(s);

L[s] = 0;

while( !q.empty() ){

int u = q.front();

q.pop();

//寻找还有残量的边

for( Edge\*p = head[u] ; p ; p = p->next ){

if( p->c <= 0 ) continue;

int v = p->node;

if( -1 != L[v] ) continue;

q.push(v);

L[v] = L[u]+1;

}

}

return -1 != L[t];

}

int dfs( int u , int e , int cf )

{

if( u == e ) return cf;

int tf = 0;

for( Edge\*p = head[u] ; p ; p = p->next ){

int v = p->node;

int c = p->c;

if( L[u]+1 == L[v] && c > 0 && cf > tf ){

int f = dfs( v , e , min(c,cf-tf) );

if( 0 == f ) continue;

p->c -= f;

p->redge->c += f;

tf += f;

}

}

if( 0 == tf ) L[u] = -1;

return tf;

}

int Dinic( int s , int t )

{

int ret = 0;

while( bfs(s,t) ){

int ans;

while( ans = dfs(s,t,INT\_MAX ) ){

ret += ans;

}

}

return ret;

}

vector<PA>mp;

void Build( int n , int m )

{

init();

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

mkEdge(i,i+n,1);

}

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

PA s = mp[i];

mkEdge(s.first+n,s.second,INF);

mkEdge(s.second+n,s.first,INF);

}

}

int main()

{

int n,m;

while( ~scanf("%d%d",&n,&m) ){

while( !mp.empty() ) mp.pop\_back();

int a,b;

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

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

mp.push\_back(PA(a,b));

}

int ans = INF;

//枚举源点和汇点

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

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

Build(n,m);

ans = min(ans,Dinic(i+n,j));

}

}

if( ans == INF ) ans = n;

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

}

return 0;

}

1. 割边的求解

#include <iostream>

#include <cstdio>

using namespace std;

typedef long long ll;

typedef pair<int,int>PA;

const int N = 100900;

const int M = 10900;

const int INF = 0x3fffffff;

//dfn[u]是深度优先搜索树节点u的优先级

//low[u]是从u或者u的子孙出发通过回边可以到达的最低深度优先数

//low[u] = min{ dfn[u] ,

// min{ low[w] | w是u的一个子女 } ,

// min{ dfn[v] | v与u邻接，且(u,v)是一条回边} }

/\*

割边的判断:

无向图中的一条边(u,v)是割边，当且仅当(u,v)是生成树种的边，且满足dfn[u]<low[v]

割边的求解与割点的求解类似

\*/

//向下搜索时，如果顶点v是u的相邻顶点，若v还未被访问，则v是u的儿子节点

//若v被访问了，则v是u的祖先节点，且(u,v)是一条回边

struct Edge{

int node;

Edge\*next;

}m\_edge[N\*2];

Edge\*head[M];

int low[M],dfn[M],Flag[M],Ecnt,cnt;

vector<PA>st;

void init()

{

Ecnt = cnt = 0;

fill( Flag , Flag+M , 0 );

fill( head , head+M , (Edge\*)0 );

}

void mkEdge( int a , int b )

{

m\_edge[Ecnt].node = b;

m\_edge[Ecnt].next = head[a];

head[a] = m\_edge+Ecnt++;

}

void tarjan( int u , int father )

{

Flag[u] = 1;

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

for( Edge\*p = head[u] ; p ; p = p->next ){

int v = p->node;

if( !Flag[v] ){

tarjan(v,u);

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

//判断(u,v)是否是割边

if( low[v] > dfn[u] ){

st.push\_back(PA(u,v));

}

}

//v是u的祖先，且(u,v)是一条回边

if( v != father && Flag[v] )

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

}

}

map<PA,int>mp;

void Build( int m )

{

int a,b,num = 0;

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

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

if( mp[PA(a,b)] != 0 ){

//将重边标记为无穷大，重边不可能是割边

mp[PA(a,b)] = INF;

mp[PA(b,a)] = INF;

num += 2;

}else{

mp[PA(a,b)] = (++num);

mp[PA(b,a)] = (++num);

mkEdge(a,b);

mkEdge(b,a);

}

}

}

int main()

{

int T,cas = 0;

scanf("%d",&T);

while( T-- ){

init();

if( !mp.empty() ) mp.clear();

while( !st.empty() ) st.pop\_back();

int n,m;

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

Build(m);

tarjan(1,-1);

int rec[N],Count = 0;

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

int s = mp[st[i]];

//重边不可能是割边

if( s != INF ) rec[Count++] = (s+1)/2;

}

if( cas != 0 ) printf("\n");

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

if( Count > 0 ){

sort( rec , rec+Count );

printf("%d",rec[0]);

for( int i = 1 ; i < Count ; ++i ){

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

}

printf("\n");

}

++cas;

}

return 0;

}

1. 边的双联通分量

#include <iostream>

#include <cstdio>

using namespace std;

typedef long long ll;

typedef pair<int,int>PA;

const int N = 10090;

const int M = 5090;

const int INF = 0x3fffffff;

//将边双联通分量上的点缩成一个点

//原图中的桥为新边，建立一颗树

//要使树变成边双联通图，至少添加的边数:(树中的叶子节点数+1)/2

struct Edge{

int node;

Edge\*next;

}m\_edge[N\*2];

Edge\*head[M];

int low[M],dfn[M],flag[M],Ecnt,dep;

map<PA,int>mp;

int sta[M]; //统计新图中的叶子节点数

void init()

{

Ecnt = dep = 0;

fill( head , head+M , (Edge\*)0 );

fill( flag , flag+M , 0 );

mp.clear();

}

void mkEdge( int a , int b )

{

m\_edge[Ecnt].node = b;

m\_edge[Ecnt].next = head[a];

head[a] = m\_edge+Ecnt++;

}

void tarjan( int u , int father )

{

flag[u] = 1;

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

for( Edge\*p = head[u] ; p ; p = p->next ){

int v = p->node;

if( !flag[v] ){

tarjan( v , u );

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

if( low[v] > dfn[u] ){

mp[PA(u,v)] = 1;

}

}

if( flag[v] && v != father ){

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

}

}

}

//求边联通分量，同一个联通分量上的点的flag指向的值相同

void dfs( int u , int num )

{

flag[u] = num;

for( Edge\*p = head[u] ; p ; p = p->next ){

int v = p->node;

if( flag[v] ) continue;

if( mp.count(PA(u,v)) == 0 && mp.count(PA(v,u)) == 0 ){

dfs( v , num );

}

}

}

//初始建图

void Build( int m )

{

int a,b;

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

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

mkEdge(a,b);

mkEdge(b,a);

}

}

int solve( int n )

{

int num = 0;

memset( flag , 0 , sizeof(flag) );

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

if( !flag[i] ){

num++;

dfs( i , num );

}

}

memset( sta , 0 , sizeof(sta) );

for( map<PA,int>::iterator it = mp.begin() ;

it != mp.end() ; ++it ){

PA s = it->first;

sta[flag[s.first]]++;

sta[flag[s.second]]++;

}

//记录新树中的叶子节点

int Count = 0;

for( int i = 1 ; i <= num ; ++i ){

if( sta[i] == 1 ) Count++;

}

return Count;

}

int main()

{

int n,m;

while( ~scanf("%d%d",&n,&m) ){

init();

Build( m );

tarjan( 1 , -1 );

int ans = solve( n );

printf("%d\n",(ans+1)/2);

}

return 0;

}

1. 有向图的强联通分量

#include <iostream>

#include <stack>

#include <vector>

#include <cstdio>

using namespace std;

const int maxn = 1e3 + 10;

int low[maxn],dfn[maxn],sccno[maxn];

int scc\_cnt,dep;

stack<int> s;

struct Edge

{

int node;

Edge\*next;

}m\_edge[maxn\*2];

Edge\*head[maxn];

int Ecnt;

void init()

{

Ecnt = 0;

fill( head , head+maxn , (Edge\*)0 );

}

void mkEdge( int a , int b )

{

m\_edge[Ecnt].node = b;

m\_edge[Ecnt].next = head[a];

head[a] = m\_edge+Ecnt++;

}

void dfs( int u )

{

s.push(u);

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

for( Edge\*p = head[u] ; p ; p = p->next ){

int v = p->node;

if( !dfn[v] ){

dfs( v );

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

}else if( !sccno[v] ){

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

}

}

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

scc\_cnt++;

while( !s.empty() ){

int t = s.top();

sccno[t] = scc\_cnt;

s.pop();

if( t == u ) break;

}

}

}

void find\_scc(int n)

{

dfs\_clock = scc\_cnt = 0;

memset(sccno,0,sizeof(sccno));

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

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

if( !dfn[i] ) dfs(i);

}

1. 网络流算法
2. dinic算法

#include <cstdio>

#include <vector>

#include <algorithm>

#include <queue>

using namespace std;

#define SIZE 205

//边的结构

struct edge\_t{

int node;

int c;//c为容量

edge\_t\* next;

edge\_t\* redge;//指向反向边

}Edge[SIZE\*2];

int ECnt;

//图的邻接表

edge\_t\* Ver[SIZE];

void init(){

ECnt = 0;

fill(Ver,Ver+SIZE,(edge\_t\*)0);

}

//生成双向边

void mkEdge(int a,int b,int c){

int t1 = ECnt++;

int t2 = ECnt++;

Edge[t1].node = b;

Edge[t1].c = c;

Edge[t1].next = Ver[a];

Edge[t1].redge = Edge + t2;

Ver[a] = Edge + t1;

Edge[t2].node = a;

Edge[t2].c = 0;

Edge[t2].next = Ver[b];

Edge[t2].redge = Edge + t1;

Ver[b] = Edge + t2;

}

int L[SIZE];//层次图

//建立残留网络从源s到汇t的层次图

bool bfs(int s,int t){

fill(L,L+SIZE,-1);

queue<int> q;

q.push(s);

L[s] = 0;

while( !q.empty() ){

int u = q.front();

q.pop();

//寻找还有残量的边

for(edge\_t\*p=Ver[u];p;p=p->next){

if ( p->c <= 0 ) continue;

int v = p->node;

if ( -1 != L[v] ) continue;

q.push(v);

L[v] = L[u] + 1;

}

}

return -1 != L[t];

}

//在层次图上搜索增广路径，本质上就是搜索可以增广的流量

//这个流量是各层之间流量的最小值

//u为当前节点，cf为当前层的最小流，t为汇点

int dfs(int u,int e,int cf){

if ( u == e ) return cf;

int tf = 0; //tf记录u往下一层的总可行流量

for(edge\_t\*p=Ver[u];p;p=p->next){

int v = p->node;

int c = p->c;

if ( L[u] + 1 == L[v] && c > 0 && cf > tf ){

int f = dfs(v,e,min(c,cf-tf));

if ( 0 == f ) continue;

p->c -= f;//正向边减去可行流量

p->redge->c += f;//反向边加上

tf += f;

}

}

if ( 0 == tf ) L[u] = -1;//修改层次图

return tf;

}

//Dinic算法，s为源，t为汇

int Dinic(int s,int t){

int ret = 0;

while( bfs(s,t) ){//第一步建立分层图

int ans;

//第二步在分层图上查找一条增广路径的可行流量

while( ans = dfs(s,t,INT\_MAX) )

ret += ans;

}

return ret;

}

int main()

{

return 0;

}

1. FF算法

#include <cstdio>

#include <algorithm>

using namespace std;

#define size1 20

#define size2 1009

//边的结构

struct edge\_t{

int node;

int c;//c为容量

edge\_t\* next;

edge\_t\* redge;//指向反向边

}Edge[size2\*2];

int ECnt;

//图的邻接表

edge\_t\* Ver[size1];

void init(){

ECnt = 0;

fill(Ver,Ver+size1,(edge\_t\*)0);

}

//生成双向边

void mkEdge(int a,int b,int c){

int t1 = ECnt++;

int t2 = ECnt++;

Edge[t1].node = b;

Edge[t1].c = c;

Edge[t1].next = Ver[a];

Edge[t1].redge = Edge + t2;

Ver[a] = Edge + t1;

Edge[t2].node = a;

Edge[t2].c = 0;

Edge[t2].next = Ver[b];

Edge[t2].redge = Edge + t1;

Ver[b] = Edge + t2;

}

bool F[size1];

//u当前节点,f为当前流

//寻找增广路径

int dfs(int u,int e,int f){

//N为汇点

if ( u == e ) return f;

F[u] = true;

for(edge\_t\*p=Ver[u];p;p=p->next){

int v = p->node;

if( F[v] ) continue;

int c = p->c;

if ( c > 0 ){

int t = dfs(v,e,min(c,f));

if ( 0 == t ) continue;

p->c -= t;

p->redge->c += t;

return t;

}

}

return 0;

}

int solve( int s , int e ){

int ret = 0;

while(1){

fill(F,F+size1,false);

int t = dfs(s,e,INT\_MAX);

if ( 0 == t ) return ret;

ret += t;

}

}

int main()

{

return 0;

}

六.欧拉回路

1.hdu3018—一笔画问题

#include <cstdio>

#include <cstring>

using namespace std;

typedef \_\_int64 ll;

const int N = 100005;

const int M = 50800;

const int INF = 0x3fffffff;

const int mod = 1e9+7;

int rec[N],parent[N],tick[N],ret[N];

//ret标记每个子图中有多少个点，

//tick标记每个子图中奇度结点的个数

int find( int a )

{

int b = a;

while( b != parent[b] ){

parent[b] = parent[parent[b]];

b = parent[b];

}

return b;

}

void merge( int a , int b )

{

int i = find(a);

int j = find(b);

if( i != j ) parent[i] = j;

}

void init()

{

fill( rec , rec+N , 0 );

fill( ret , ret+N , 0 );

fill( tick , tick+N , -1 );

for( int i = 1 ; i <= N ; ++i ) parent[i]=i;

}

int main()

{

int n,m;

while( ~scanf("%d%d",&n,&m) ){

init();

int a,b,tot=0,re=0;

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

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

++rec[a];++rec[b];

merge(a,b);

}

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

if( parent[i] == i ){

++re;tick[i] = 0;

}

}

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

int p = find(i);

++ret[p];

if( rec[i]%2 != 0 ) ++tick[p];

}

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

if( tick[i] == -1 ) continue;

if( tick[i] == 0 && ret[i] != 1 ) ++tot;

tot += tick[i]/2;

}

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

}

return 0;

}