网络流

网络流是一个有向图模型，其拥有源点和汇点，图上边权称为容量，在不超过容量的情况下源点流出的值最终能从源点流向汇点的多少称为流量。

网络流

- 最大流
- 最大流最小割定理
- Ford-Fulkerson算法
  - luoguP3376 【模板】网络最大流
- Edmond-Karp算法
- Dinic算法
  - 步骤
  - 代码模板
  - 优化版本
- ISAP算法
  - 原理
  - 算法流程
  - 优化
  - 参考代码
- Push-Relabel算法
  - 算法流程
- 最高标号预流推进（HLPP）
  - 算法步骤
  - 优化
- 其他算法
- 例题
  - 圆桌问题
  - P5038 [SCOI2012]奇怪的游戏
  - P3191 [HNOI2007]紧急疏散EVACUATE
- 参考资料

最大流

在网络流最常见的问题就是最大流，即从源点出发，流出的流量足够多，求能够流入汇点的最大量。下面要介绍用于解决这一问题的Ford-Fulkerson算法。

最大流最小割定理

所谓“割”，就是从网络中去掉某些边后，剩下两个不连通的分别包含源点和汇点的点集。割的大小是这些边的容量之和。在所有可行的割中，最小的割称为**最小割**。这个定理就短短几个字：**最大流等于最小割**。

Ford-Fulkerson算法

FF算法核心在找**增广路**。找增广路即我们找到一条从源点到汇点的路径，提供的流量等于路径上的最小容量，那么我们将路径上各边都扣除这一流量，处理完后各边剩余容量称为**残余容量**。增广路即指源点到汇点上，所有边残余容量均大于0的路径，FF算法不停找增广路直到找不到为止。但其实这个做法并不是正确的，需要稍加改进。

```
//这是由1流向4的网络，可以得到2的流量，但是按上面的做法只能得到1
1 2 1
1 3 1
2 3 1
3 4 1
2 4 1
```

为了解决这个问题，我们引入**反向边**的概念。在建图的时候，对每条边建边权为0的反向边，每次扣除正向边流量时，反向边要加上同等流量。我们可以把反向边理解为一个撤销的操作，走反向边时抵消了第一次走这条边的效果，这样我们就可以保证找不到增广路时，能够得到最大流。用dfs实现FF算法复杂度为$O(ef)$，其中e为边数，f为最大流

luoguP3376 【模板】网络最大流

```
//Ford-Fulkerson算法
#include<bits/stdc++.h>
#define inf 0x3f3f3f3f
#define int long long
using namespace std;
const int N=200,M=1e4+7;

int read(){ int x=0,f=1;char ch=getchar();while(ch<'0'||ch>'9'){if(ch=='-')
f=f*-1;ch=getchar();}while(ch>='0'&&ch<='9'){x=x*10+ch-'0';ch=getchar();}return
x*f;}

struct E{
    int v,w,nxt;
}e[M];
int head[N],cnt=1;

inline void addedge(int f,int v,int w){
    e[++cnt]={v,w,head[f]};head[f]=cnt;
}

int n,m,s,t;
bool vis[N];

int dfs(int p=s,int flow=inf){
    if(p==t) return flow;
    vis[p]=1;
    for(int i=head[p];i;i=e[i].nxt){
        int to=e[i].v,w=e[i].w,c;
        if(w>0&&!vis[to]&&(c=dfs(to,min(w,flow)))!=-1){
            e[i].w-=c;
            e[i^1].w+=c;
            return c;
        }
    }
    return -1;
```

```
}

inline int FF(){
    int ans=0,c;
    while((c=dfs())!=-1){
        memset(vis,0,sizeof(vis));
        ans+=c;
    }
    return ans;
}

signed main(){
    n=read();m=read();s=read();t=read();
    for(int i=1,u,v,w;i<=m;i++){
        u=read();v=read();w=read();
        addedge(u,v,w);
        addedge(v,u,0);
    }
    printf("%lld",FF());
    return 0;
}
```

这个写法在随机图上的效率还是比较低的，很容易被卡。下面是一种更高效的算法。

Edmond-Karp算法

我们FF算法写成bfs版本，就变成了为人熟知的EK(Edmond-Karp)算法了。

原理和上面是一样的，直接上模板吧！

```
//Edmond-Karp算法
#include<bits/stdc++.h>
#define int long long
#define inf 0x3f3f3f3f
using namespace std;
const int N=200,M=1e4+7;

int read(){ int x=0,f=1;char ch=getchar();while(ch<'0'||ch>'9'){if(ch=='-')
f=f*-1;ch=getchar();}while(ch>='0'&&ch<='9'){x=x*10+ch-'0';ch=getchar();}return
x*f;}

struct E{
    int v,w,nxt;
}e[M];
int head[N],cnt=1;

inline void addedge(int f,int v,int w){
    e[++cnt]={v,w,head[f]};head[f]=cnt;
}

int n,m,s,t;
int lst[N],flow[N];
bool vis[N];

bool bfs(){
```

```
    memset(lst,-1,sizeof(lst));
    flow[s]=inf;
    queue<int>q;
    q.push(s);
    while(!q.empty()){
        int fro=q.front();
        q.pop();
        for(int i=head[fro];i;i=e[i].nxt){
            int to=e[i].v;
            if(lst[to]==-1&&e[i].w>0){
                flow[to]=min(flow[fro],e[i].w);
                lst[to]=i;
                q.push(to);
                if(to==t) return true;
            }
        }
    }
    return false;
}

inline int EK(){
    int ans=0;
    while(bfs()){
        ans+=flow[t];
        for(int i=t;i!=s;i=e[lst[i]^1].v){
            e[lst[i]].w-=flow[t];
            e[lst[i]^1].w+=flow[t];
        }
    }
    return ans;
}

signed main(){
    n=read();m=read();s=read();t=read();
    for(int i=1,u,v,w;i<=m;i++){
        u=read();v=read();w=read();
        addedge(u,v,w);
        addedge(v,u,0);
    }
    printf("%lld",EK());
    return 0;
}
```

因为BFS可以保证每次的增广路都较短，而DFS很可能会绕远路，所以EK的算法效率比FF更好。EK算法的复杂度上限是$O(ve^2)$，其中v为点数

Dinic算法

划重点！这个算法是目前最常用的网络流算法，采用了**BFS分层**，然后DFS寻找最大流的方法，复杂度上限为$O(v^2e)$。分层指的是预处理出源点到每个点的距离，然后往远的方向走，这样保证不走回头路也不绕圈子。

步骤

（1）在残余网络上通过BFS进行分层

（2）在层次图中DFS，沿着层次增1且cap>flow的方向找增广路，回溯时增流。

（3）重复以上步骤，直到图中不存在增广路为止。

代码模板

```
bool bfs(){
    while(!q.empty()) q.pop();
    for(int i=1;i<=n;i++)   vis[i]=0;
    q.push(s);vis[s]=1;dis[s]=inf;
    while(!q.empty()){
        int fro=q.front();
        q.pop();
        for(int i=head[fro];i;i=edge[i].next){
            int to=edge[i].to,w=edge[i].dis;
            if(w==0||vis[to]) continue;
            vis[to]=1; //用了vis和dis对点进行分层
            q.push(to);
            pre[to]=i;
            dis[to]=min(w,dis[fro]);
            if(to==t) return 1;
        }
    }
    return 0;
}

int dfs(int p){
    while(p!=s){
        int x=pre[p];
        edge[x].dis-=dis[t];
        edge[x^1].dis+=dis[t];
        p=edge[x^1].to;
    }
    return dis[t];
}

void dinic(){
    while(bfs()){
        ans+=dfs(t);
    }
}
```

优化版本

1.当前弧优化

一条边增广之后不会再次增广，所以再次增广就不需要考虑这条边了。用一个$cur[i]$去代替$head[i]$作为起点进行边的枚举即可。

2.多路增广

如果在某点DFS时，还有多余的流量未用，就尝试在该点DFS尝试找到更多的增广路。

```
//Dinic优化算法
#include<bits/stdc++.h>
```

```cpp
#define int long long
#define inf 0x3f3f3f3f
using namespace std;
const int N=200,M=1e4+7;

int read(){ int x=0,f=1;char ch=getchar();while(ch<'0'||ch>'9'){if(ch=='-')
f=f*-1;ch=getchar();}while(ch>='0'&&ch<='9'){x=x*10+ch-'0';ch=getchar();}return
x*f;}

struct E{
    int v,w,nxt;
}e[M];
int head[N],cur[N],cnt=1;

inline void addedge(int f,int v,int w){
    e[++cnt]={v,w,head[f]};head[f]=cnt;
}

int n,m,s,t;
int lv[N];

inline bool bfs(){ //BFS分层
    memset(lv,-1,sizeof(lv));
    memcpy(cur,head,sizeof(head)); //初始化弧优化
    queue<int>q;
    q.push(s);lv[s]=0;
    while(!q.empty()){
        int fro=q.front();
        q.pop();
        for(int i=head[fro];i;i=e[i].nxt){
            int to=e[i].v,dis=e[i].w;
            if(dis>0&&lv[to]==-1){
                lv[to]=lv[fro]+1;
                q.push(to);
                if(to==t) return true;
            }
        }
    }
    return false;
}

inline int dfs(int p=s,int flow=inf){
    if(p==t) return flow;
    int lft=flow; //剩余的流量
    for(int i=cur[p];i&&lft;i=e[i].nxt){ //从当前弧开始出发
        cur[p]=i; //更新当前弧
        int to=e[i].v,dis=e[i].w;
        if(dis>0&&lv[to]==lv[p]+1){ //向层数高的地方增广
            int c=dfs(to,min(dis,lft)); //尽可能多地传递流量
            lft-=c;  //更新剩余流量
            e[i].w-=c; //更新残差流量
            e[i^1].w+=c; //
        }
    }
    return flow-lft; //返回传递出去的流量大小
}

int Dinic(){
```

```
    int ans=0;
    while(bfs()){
        ans+=dfs();
    }
    return ans;
}


signed main(){
    n=read();m=read();s=read();t=read();
    for(int i=1,u,v,w;i<=m;i++){
        u=read();v=read();w=read();
        addedge(u,v,w);
        addedge(v,u,0);
    }
    printf("%lld",Dinic());
    return 0;
}
```

值得一提的是，这个算法如果用在二分图中复杂度是$O(v\sqrt{v})$，优于匈牙利算法。

ISAP算法

在Dinic算法中，我们每次求完增广路后都要跑BFS来分层，而在ISAP算法中，我们选择在反图上，从t点向s点进行BFS，执行完分层过程后，通过DFS来找增广路。与Dinic类似，我们只选择比当前点层数少1的点来增广。与Dinic不同的是，我们并不会重跑BFS来对图上的点重新分层，而是在增广的过程中就完成了分层过程。

原理

Dinic复杂度之所以高，是因为它每次进行搜索增广路都要进行$O(n)$的BFS（每次都需要重新计算dep数组但因为去掉一条边只可能令路径变得更长，这显得很没必要。如果增广之前的残量网络存在另一条最短路，并且在增广后依然存在，那么这条路径无疑是最短的），ISAP对此进行优化，叫做**允许弧优化。**ISAP没有马上更新dep数组，而是继续增广，直到找到死路，才执行回溯操作。回溯的主要任务便是更新dep数组，过程为：

①设从结点u找遍了邻边也没有找到允许弧；

②再设一变量minn,令minn等于残量网络中u的所有邻接点的$dep[v]$的最小值，然后令$d[u]$等于$minn+1$即可。

算法流程

- 定义结点的标号到汇点的最短距离；
- 每次沿允许弧进行增广；
- 找到增广路后，将路径上所有边的流量更新
- 遍历完当前结点的可行边后更新当前结点的标号为$dep[u]=min(dep[v]\&\&add_flow(u,v)>0)+1$，使下次再搜的时候有路可走。
- 图中不存在增广路后即退出程序，此时得到的流量就是最大流。

优化

很多时候ISAP都可以提前结束程序，提速高达100倍以上。**GAP优化**：是当发现u是距离t最短距离为$dep[u]$的最后一个点，删去这个点后，整个残余网络中没有到t最短距离为$dep[u]$的点了，就出现了断层，导致s和t不连通，这时候可以直接结束ISAP函数。

另外，ISAP也可以使用**当前弧优化**，枚举边时利用&传地址，及时更新。

参考代码

```
//ISAP算法
#include<bits/stdc++.h>
#define int long long
#define inf 0x3f3f3f3f
using namespace std;
const int N=20010,M=4e5+7;

int read(){ int x=0,f=1;char ch=getchar();while(ch<'0'||ch>'9'){if(ch=='-')
f=f*-1;ch=getchar();}while(ch>='0'&&ch<='9'){x=x*10+ch-'0';ch=getchar();}return
x*f;}

struct E{
    int v,w,nxt;
}e[M];
int head[N],cur[N],cnt=1; //建反图是cnt从1开始

inline void addedge(int f,int v,int w){ //链式前向星建边
    e[++cnt]={v,w,head[f]};head[f]=cnt;
}

int n,m,s,t,u;
int dep[N],lst[N],num[N];

void bfs(int t){ //这更像是初始化以及对dep预处理
    queue<int>q;
    for(int i=1;i<=n;i++) cur[i]=head[i];
    for(int i=1;i<=n;i++) dep[i]=n;
    dep[t]=0;
    q.push(t);
    while(!q.empty()){
        int fro=q.front();
        q.pop();
        for(int i=head[fro];i;i=e[i].nxt){
            int to=e[i].v;
            if(dep[to]==n&&e[i^1].w>0){
                dep[to]=dep[u]+1;
                q.push(to);
            }
        }
    }
}

int add_flow(int s,int t){ //获取最大流并更新残余网络中的流量
    int ans=inf,u=t;
    while(u!=s){
        ans=min(ans,e[lst[u]].w);
        u=e[lst[u]^1].v;
    }
```

```
    u=t;
    while(u!=s){
        e[lst[u]].w-=ans;
        e[lst[u]^1].w+=ans;
        u=e[lst[u]^1].v;
    }
    return ans;
}

int ISAP(){  //Improved Shortest Augmenting Path
    int u=s,maxflow=0;
    bfs(t); //定义结点到汇点的最短距离
    for(int i=1;i<=n;i++) num[dep[i]]++; //记录每个深度的点数
    while(dep[s]<n){ //如果没出现断层，最多跑n-dep次
        if(u==t) maxflow+=add_flow(s,t),u=s;
        bool flag=0;
        for(int &i=cur[u];i;i=e[i].nxt){ //当前弧优化
            int to=e[i].v,dis=e[i].w;
            if(dep[u]==dep[to]+1&&e[i].w>0){
                flag=1;
                u=to;
                lst[to]=i;
                break;
            }
        }
        if(!flag){  //直到断层/没有可继续前进的点
            int minn=n-1;
            for(int i=head[u];i;i=e[i].nxt){ //遍历当前节点的可行边
                int to=e[i].v,dis=e[i].w;
                if(dis>0) minn=min(minn,dep[to]);
            }
            if((--num[dep[u]])==0) break; //GAP优化！！
            num[dep[u]=minn+1]++; //更新dep使下次搜时有路可走
            cur[u]=head[u];  //当前弧优化
            if(u!=s) u=e[lst[u]^1].v;
        }
    }
    return maxflow; //得到最大流
}

signed main(){
    n=read();m=read();s=read();t=read();
    for(int i=1,u,v,w;i<=m;i++){
        u=read();v=read();w=read();
        addedge(u,v,w);
        addedge(v,u,0);
    }
    printf("%lld",ISAP());
    return 0;
}
```

Push-Relabel算法

预流推进(PR)简单来说就是"能推流就推流，要求的就是最终推到t的流量"。

算法流程

- 设源点s有无线多水(余流)，向周围点推流，并让周围点入队。
- 不断取队首元素推流
- 直到队列为空时结束算法，汇点t的余流即为最大流

这个很简单的思路有一个问题，就是两个点可能有不断来回推流的情况，我们需要给每一个点赋高度，使得水只往低处流，并且**不断对有余流的点更新高度**，直到这些点没有余流为止。

余流：即结点当前拥有的流量。

高度更新：更新余流后结点的高度变为附近最低结点的高度+1。

虽然这东西很玄学，但是我们还是可以感性理解为这个算法是对的。模板就不放了，因为效率相当的低，目前已经过不了洛谷的模板了。为什么呢？我们可以想象，在一条路径上，几个点之间的推流可以重复进行很多次，这极大影响了效率。不过上面介绍的东西都是有用了，为下面的算法稍微做点铺垫。

最高标号预流推进（HLPP）

HLPP(Highest Label Preflow Push)通过BFS预先处理了高度标号，并利用优先队列使得每次推流都是高度最高的顶点，以此减少了推流的次数和重标号的次数。

算法步骤

- 先从t到s反向BFS，使每个点有一个初始高度。
- 从s开始向外推流，将有余流的点放入优先队列。
- 不断从优先队列里取出高度最高的点进行推流操作。
- 如果推完还有余流，更新高度标号，重新放入优先队列。
- 当优先队列为空时结束算法，最大流即为t的余流。

优化

和ISAP算法一样可以使用**gap优化**，即某种高度不存在时，将所有比改高度高的节点标记为不可到达。

```
//HLPP算法
#include<bits/stdc++.h>
#define up(l,r,i) for(register int i=l;i<=r;++i)
#define ergv(u) for(std::vector<E>::iterator p=G[u].begin();p!=G[u].end();++p)
#define ergl(u) for(std::list<int>::iterator
p=lst[u].begin();p!=lst[u].end();++p)
const int inf=2147483647;
const int N=1203,M=120007;

int read(){ int x=0,f=1;char ch=getchar();while(ch<'0'||ch>'9'){if(ch=='-')
f=f*-1;ch=getchar();}while(ch>='0'&&ch<='9'){x=x*10+ch-'0';ch=getchar();}return
x*f;}

int n,m,s,t,mxflow,hst,nwh;
std::vector<int> lft,gap,ht,q,mlst[N]; //lft为余流，gap记录各高度点数，ht记录最高高
度，,mlst用于桶排
```

```
struct E{
    int v,w,nxt;
    E(int _ver,int _flw,int _nxt):v(_ver),w(_flw),nxt(_nxt){}
};
std::vector<E>G[N];
std::list<int>lst[N];
std::vector<std::list<int>::iterator> it;

inline void addedge(int u,int v,int w){     //连边
    G[u].push_back(E(v,w,G[v].size()));
    G[v].push_back(E(u,0,G[u].size()-1));
}
inline void relabel(){      //全局重贴标签
    ht.assign(n+3,n); //assign 可以让vector的多个位置赋同一个值
    gap.assign(n+3,0);
    ht[t]=0;
    q.clear();
    q.resize(n+3);
    int front=0,rear=0,u;
    for(q[rear++]=t;front<rear;){  //手写队列
        u=q[front++];
        ergv(u) if(ht[p->v]==n&&G[p->v][p->nxt].w)
        ++gap[ht[p->v]=ht[u]+1],q[rear++]=p->v;
    }
    for(int i=1;i<=n;i++) mlst[i].clear(),lst[i].clear();
    for(int i=1;i<=n;i++) if(ht[i]<n){
        it[i]=lst[ht[i]].insert(lst[ht[i]].begin(),i);
        if(lft[i]>0) mlst[ht[i]].push_back(i);
    }
    hst=(nwh=ht[q[rear-1]]);
}
inline void psh(int u,E &e){    //推流子函数
    int v=e.v,df=std::min(lft[u],e.w);
    e.w-=df,G[v][e.nxt].w+=df,lft[u]-=df,lft[v]+=df;
    if(lft[v]>0&&lft[v]<=df) mlst[ht[v]].push_back(v);
}

inline void psh(int u){         //真正的推流
    int nh=n,htu=ht[u];
    ergv(u) if(p->w)
            if(ht[u]==ht[p->v]+1){
                psh(u,*p);
                if(lft[u]<=0) return;
            }else nh=std::min(nh,ht[p->v]+1);
    if(gap[htu]==1){          //断层
        up(htu,hst,i){
            ergl(i) --gap[ht[*p]],ht[*p]=n;
            lst[i].clear();
        }
        hst=htu-1;
    }else{
        --gap[htu],it[u]=lst[htu].erase(it[u]),ht[u]=nh;
        if(nh==n) return;
        gap[nh]++,it[u]=lst[nh].insert(lst[nh].begin(),u);
        hst=std::max(hst,nwh=nh),mlst[nh].push_back(u);
    }
}
int HLPP(){ //最高标号预流推进
```

```
    nwh=hst=0;
    ht.assign(n+3,0);  //初始化高度
    ht[s]=n;
    it.resize(n+3);
    for(int i=1;i<=n;i++) if(i!=s)
it[i]=lst[ht[i]].insert(lst[ht[i]].begin(),i);
    gap.assign(n+3,0),gap[0]=n-1,lft.assign(n+3,0),lft[s]=inf,lft[t]=-inf;
    ergv(s) psh(s,*p);
    relabel();
    for(int u;nwh;){
        if(mlst[nwh].empty()) nwh--;
        else u=mlst[nwh].back(),mlst[nwh].pop_back(),psh(u);
    }
    return lft[t]+inf;
}
signed main(){
    n=read(),m=read(),s=read(),t=read();
    for(int i=1,u,v,w;i<=m;i++){
        u=read();v=read();w=read();
        addedge(u,v,w);
    }
    std::printf("%d\n",HLPP());
    return 0;
}
```

上面是洛谷最优解，在各种地方都有优化，速度快得惊人。我还有很多地方没来得及吸收。放上来以供各位参考。

其他算法

MPM(Malhotra, Pramodh-Kumar and Maheshwari) 算法

拥有两种得到最大流的方法：①使用优先队列，复杂度为$O(n^3logn)$；②常用BFS解法，复杂度$O(n^3)$，在寻找增广路的部分需要$O(n^2)$，这部分优于Dinic算法。

例题

圆桌问题

给定代表团数量n和餐桌数量m，再给定各代表团人数和餐桌容量，要求代表团中的每一个人去不同的餐桌吃饭，问是否有解以及任意方案。

```
//Dinic优化算法
#include<bits/stdc++.h>
#define int long long
#define inf 0x3f3f3f3f
using namespace std;
const int N=1000,M=2e5+7;

int read(){ int x=0,f=1;char ch=getchar();while(ch<'0'||ch>'9'){if(ch=='-')
f=f*-1;ch=getchar();}while(ch>='0'&&ch<='9'){x=x*10+ch-'0';ch=getchar();}return
x*f;}

struct E{
```

```
    int v,w,nxt;
}e[M];
int head[N],cur[N],cnt=1;

inline void addedge(int f,int v,int w){
    e[++cnt]={v,w,head[f]};head[f]=cnt;
    e[++cnt]={f,0,head[v]};head[v]=cnt;
}

int n,m,s,t;
int lv[N];

inline bool bfs(){ //BFS分层
    memset(lv,-1,sizeof(lv));
    memcpy(cur,head,sizeof(head)); //初始化弧优化
    queue<int>q;
    q.push(s);lv[s]=0;
    while(!q.empty()){
        int fro=q.front();
        q.pop();
        for(int i=head[fro];i;i=e[i].nxt){
            int to=e[i].v,dis=e[i].w;
            if(dis>0&&lv[to]==-1){
                lv[to]=lv[fro]+1;
                q.push(to);
                if(to==t) return true;
            }
        }
    }
    return false;
}

inline int dfs(int p=s,int flow=inf){
    if(p==t) return flow;
    int lft=flow; //剩余的流量
    for(int i=cur[p];i&&lft;i=e[i].nxt){ //从当前弧开始出发
        cur[p]=i; //更新当前弧
        int to=e[i].v,dis=e[i].w;
        if(dis>0&&lv[to]==lv[p]+1){ //向层数高的地方增广
            int c=dfs(to,min(dis,lft)); //尽可能多地传递流量
            lft-=c;  //更新剩余流量
            e[i].w-=c; //更新残差流量
            e[i^1].w+=c; //
        }
    }
    return flow-lft; //返回传递出去的流量大小
}

int Dinic(){
    int ans=0;
    while(bfs()){
        ans+=dfs();
    }
    return ans;
}


signed main(){
```

```
    n=read();m=read();
    s=0;t=n+m+1;
    int sum=0;
    for(int i=1,x;i<=n;++i) x=read(),sum+=x,addedge(s,i,x);  //每个单位的代表数
    for(int i=1;i<=m;++i) addedge(n+i,t,read()); //每个餐桌的容量
    for(int i=1;i<=n;++i){
        for(int j=1;j<=m;++j){
            addedge(i,j+n,1);
        }
    }
    int mx=Dinic();
    if(mx!=sum) puts("0");
    else{
        puts("1");
        std::vector<int>res;
        for(int i=1;i<=n;++i){
            res.clear();
            for(int j=head[i];j;j=e[j].nxt){
                int to=e[j].v,w=e[j].w;
                if(!w&&to>n) res.emplace_back(to-n);
            }
            sort(res.begin(),res.end());
            for(auto to:res) printf("%d ",to);
            puts("");
        }
    }
    return 0;
}
```

P5038 [SCOI2012]奇怪的游戏

给定$n\times m$的棋盘，每次可以选择相邻的格子**(黑白染色)**使其两个数都加一**（二分图）**，问棋盘上的数都变成同一个数**（最大流）**最少需要操作多少次**（二分）**。

思路：假设最后全部数字变为X，那么就会有
$B\times X-b=W\times X-w$，其中B,W分别为黑白点个数，b,w分别为黑白点的权值总和。判掉无解的情况之后，二分X即可。$check$函数首先将S向黑点连边,容量为$x-v[i]$,向白点连容量inf的边，然后白点再向T连容量为$x-v[j]$的边，最后判是否能跑满最大流V。

```
#include<bits/stdc++.h>
#define int long long
using namespace std;
const int inf=0x7F7F7F7F7F7F7F7F;
const int N=1e4+7,M=N*20;
const int dx[]={0,0,-1,1},dy[]={-1,1,0,0};

struct Graph{
    struct E{
        int u,v,w,nxt;
    }e[M];
    int head[N],cnt;
    int s,t,dis[N];
    inline void init(){
        cnt=1;
```

```
        memset(head,0,sizeof(head));
        s=N-1,t=N-2;
    }
    inline void add(int u,int v,int w){
        e[++cnt]=(E){u,v,w,head[u]};head[u]=cnt;
        e[++cnt]=(E){v,u,0,head[v]};head[v]=cnt;
    }
    queue<int>q;
    bool bfs(){
        memset(dis,-1,sizeof(dis));
        q.emplace(s);
        dis[s]=0;
        while(q.size()){
            int u=q.front();
            q.pop();
            for(int i=head[u];i;i=e[i].nxt){
                int to=e[i].v;
                if(dis[to]==-1&&e[i].w){
                    dis[to]=dis[u]+1;
                    q.emplace(to);
                }
            }
        }
        return dis[t]!=-1;
    }
    int dfs(int u,int flow){
        if(u==t) return flow;
        int lf=flow;
        for(int i=head[u];i&&lf;i=e[i].nxt){
            int to=e[i].v;
            if(dis[to]==dis[u]+1&&e[i].w){
                int k=dfs(to,min(e[i].w,lf));
                e[i].w-=k;
                e[i^1].w+=k;
                lf-=k;
            }
        }
        if(flow==lf) dis[u]=-1;
        return flow-lf;
    }
    int Dinic(){
        int ans=0;
        while(bfs()) ans+=dfs(s,inf);
        return ans;
    }
}G;
#define id(i,j) ((i)*40+(j))
int n,m,a[105][105];

bool check(int val){
    G.init();
    int sum=0;
    for(int i=1;i<=n;++i){
        for(int j=1;j<=m;++j){
            if((i+j)%2==0){
                G.add(G.s,id(i,j),val-a[i][j]);
                sum+=val-a[i][j];
                for(int d=0;d<4;++d){
```

```
                    int tx=i+dx[d],ty=j+dy[d];
                    if(tx>=1&&tx<=n&&ty>=1&&ty<=m){
                        G.add(id(i,j),id(tx,ty),inf);
                    }
                }
            }else{
                G.add(id(i,j),G.t,val-a[i][j]);
            }
        }
    }
    return G.Dinic()==sum;
}

int mx,s0,s1,c0,c1;
void solve(){
    cin>>n>>m;
    mx=s0=s1=c0=c1=0;
    for(int i=1;i<=n;++i){
        for(int j=1;j<=m;++j){
            cin>>a[i][j];
            mx=max(mx,a[i][j]);
            if((i+j)%2==0) s0+=a[i][j],c0++;
            else s1+=a[i][j],c1++;
        }
    }
    if(c0!=c1){
        int x=(s0-s1)/(c0-c1);
        if(x>=mx&&check(x)) cout<<x*c1-s1<<"\n";
        else cout<<"-1\n";
    }else{
        if(s0!=s1) cout<<"-1\n";
        else{
            int L=mx-1,R=(inf>>1),M;
            while(L+1!=R){
                M=((L+R)>>1);
                if(check(M)) R=M;
                else L=M;
            }
            if(R==(inf>>1)) cout<<"-1\n";
            else cout<<R*c1-s1<<"\n";
        }
    }
}

signed main(){
    ios::sync_with_stdio(0);
    cin.tie(0);cout.tie(0);
    int t;
    cin>>t;
    while(t--){
        solve();
    }
    return 0;
}
```

P3191 [HNOI2007]紧急疏散EVACUATE

给定$n\times m$的矩形区域，如果格子是**“.”**，表示一块空地；如果是**“D”**，表示一扇门；如果是**”X“**,表示一道墙，问所有人安全撤离的最短时间。$3 \leq n, m \leq 20$。

解法：二分答案，源点向每个空地连一条容量为1的边；将门拆成mid个点，表示1到mid秒的门，并都向汇点连一条容量为1的边，随后门向下一秒的门连容量为inf的边，然后如果空地在p秒达到的门，就向p秒后的门连边。无解的情况是最大流不等于总人数说明无法全部逃离。

```
#include <bits/stdc++.h>
using namespace std;
const int inf = 0x3f3f3f3f;
const int L=21,M=1e6+5,N=5e4+5;
const int dx[]={-1,1,0,0},dy[]={0,0,-1,1};
int dis[N][L][L], h[N][2], G[L][L],lv[N];
int n,m,s,t,E,R,l=1,r=1000, Ans=-1;
bool vis[L][L]; char c[L][L];

inline int read(){
    int x=0;char ch=getchar();
    while(!isdigit(ch)) ch=getchar();
    while(isdigit(ch)) x=(x<<3)+(x<<1)+ch-'0',ch=getchar();
    return x;
}

inline void write(int x){
    if(x>9) write(x/10);
    putchar(x%10+'0');
}

struct E{
    int v,w,nxt;
}e[M];
int head[N],cur[N],cnt=1;
inline void add(int u,int v,int w){
    e[++cnt]={v,w,head[u]};head[u]=cnt;
    e[++cnt]={u,0,head[v]};head[v]=cnt;
}

queue<int>q;
bool bfs(){
    for(int i=s;i<=t;++i) cur[i]=head[i],lv[i]=-1;
    q.emplace(s);
    lv[s]=0;
    while(q.size()){
        int u=q.front();
        q.pop();
        for(int i=head[u];i;i=e[i].nxt){
            int to=e[i].v;
            if(lv[to]==-1&&e[i].w){
                lv[to]=lv[u]+1;
                q.emplace(to);
            }
        }
    }
    return lv[t]!=-1;
}
```

```
int dfs(int u,int flow){
    if(u==t) return flow;
    int lf=flow;
    for(int &i=cur[u];i&&lf;i=e[i].nxt){
        int to=e[i].v;
        if(lv[to]==lv[u]+1&&e[i].w){
            int k=dfs(to,min(e[i].w,lf));
            e[i].w-=k;
            e[i^1].w+=k;
            lf-=k;
        }
    }
    if(flow==lf) lv[u]=-1;
    return flow-lf;
}
int Dinic(){
    int ans=0;
    while(bfs()) ans+=dfs(s,inf);
    return ans;
}

inline void Ready(int s,int px,int py){
    memset(vis,0,sizeof(vis));
    int x,y,tmp=0,w=1;
    vis[px][py]=1;
    dis[s][px][py]=0;h[w][0]=px;h[w][1]=py;
    while((tmp++)<w){
        for(int d=0;d<4;++d){
            int tx=h[tmp][0]+dx[d],ty=h[tmp][1]+dy[d];
            if(tx<1||tx>n||ty<1||ty>m||vis[tx][ty]||c[tx-1][ty-1]!='.')
continue;
            dis[s][tx][ty]=dis[s][h[tmp][0]][h[tmp][1]]+1;
            h[++w][0]=tx;h[w][1]=ty;vis[tx][ty]=1;
        }
    }
}

inline bool check(int mi){
    s=0;t=R+E*mi+1;cnt=1;
    memset(head,0,sizeof(head));
    for(int i=1;i<=n;++i){
        for(int j=1;j<=m;++j){
            if(c[i-1][j-1]=='.') add(s,G[i][j],1);
        }
    }
    for(int k=1;k<=E;++k){
        for(int i=1;i<=n;++i){
            for(int j=1;j<=m;++j){
                if(c[i-1][j-1]=='.'&&dis[k][i][j]<=mi){
                    add(G[i][j],(k-1)*mi+R+dis[k][i][j],1);
                }
            }
        }
    }
    for(int i=1;i<=E;++i){
        for(int j=1;j<=mi;++j){
            int tmp=(i-1)*mi+R;
            add(tmp+j,t,1);
```

```
            if(j!=mi) add(tmp+j,tmp+j+1,inf);
        }
    }
    return Dinic()==R;
}

signed main(){
    memset(dis,inf,sizeof(dis));
    n=read();m=read();
    for(int i=0;i<n;++i) scanf("%s",c[i]);
    for(int i=1;i<=n;++i){
        for(int j=1;j<=m;++j){
            if(c[i-1][j-1]=='D') Ready(++E,i,j);
            else if(c[i-1][j-1]=='.') G[i][j]=++R;
        }
    }
    while(l<=r){
        int mid=l+r>>1;
        if(check(mid)) Ans=mid,r=mid-1;
        else l=mid+1;
    }
    if(Ans==-1) puts("impossible");
    else write(Ans);
    return 0;
}
```

参考资料

《算法训练营》——陈小玉

《网络流建模》——周尚彦

OI-wiki

https://zhuanlan.zhihu.com/p/122375531

https://www.mina.moe/archives/6704

https://www.luogu.com.cn/blog/ONE-PIECE/jiu-ji-di-zui-tai-liu-suan-fa-isap-yu-hlpp

https://www.luogu.com.cn/blog/McHf/p4722-network-flows-HLPP