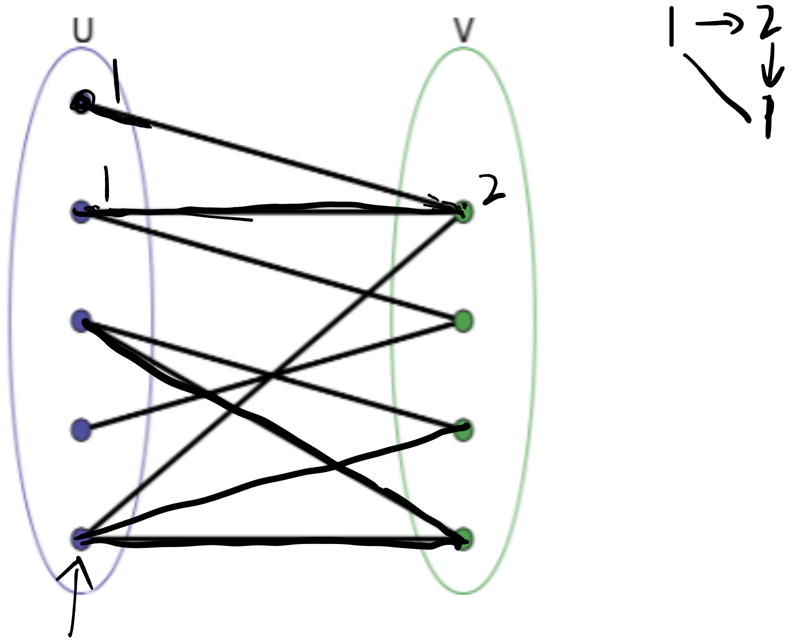
二分图

概念：

如果一张无向图的 N 个节点（$N \geq 2$)可以分成 U, V 两个非空集合，其中$U \cap V = \emptyset$，并且在同一个集合内的点之间都没有边相连，那么称这张无向图为一张二分图。U, V 分别称为二分图的左部和右部。



一张无向图是二分图，当且仅当图中不存在奇环（长度为奇数的环）

```
// 染色法判定二分图
bool dfs(int x, int color){
    v[x] = color;
    for(int i=head[x];i;i=nxt[i]){
        int y = ver[i];
        if(v[y] == 0) {
            if(!dfs(y, 3 - color)) return false;
        } else if(v[y] + v[x] != 3) return false;
    }
    return true;
}
// main中
int flag = 1;
for(int i=1;i<=n;i++) if(v[i] == 0) flag &= dfs(i, 1);
```

1-2

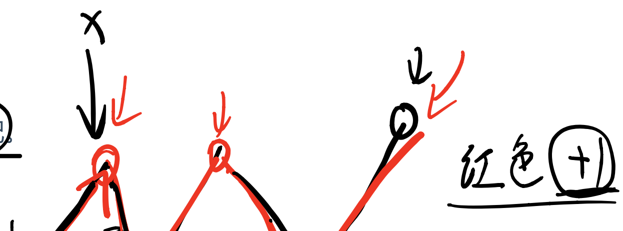
二分图的常见问题

1. 二分图最大匹配
2. 二分图带权匹配 → KM 费用流
3. 二分图最小点覆盖
4. 二分图最大独立集
5. 有向无环图的最小路径点覆盖

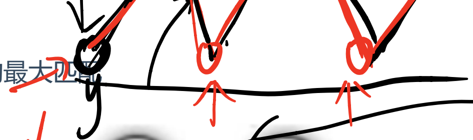
问题3,4,5都可以转换成二分图的最大匹配问题。

二分图最大匹配

“任意两条边都没有公共端点”的边的集合被称为图的一组匹配。



在二分图中，包含边数最多的一组匹配被称为二分图的最大匹配



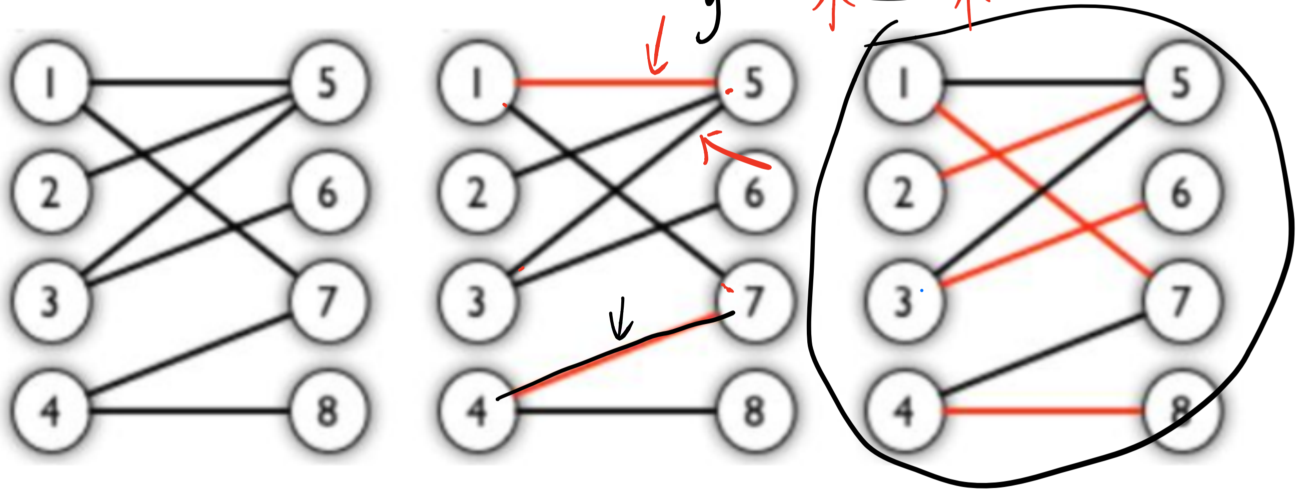


图1 图2 图3

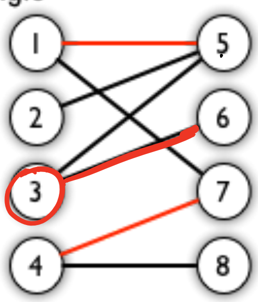
求解最大匹配的关键在于寻找**增广路**

一些相关概念：匹配边，非匹配边，匹配点，非匹配点。

增广路：二分图中存在一条连接两个非匹配点的路径path，使得非匹配边和匹配边在path上交替出现，那么称path为增广路。

另外：增广路长度必须是奇数，非匹配边比匹配边恰好多一条。

Fig.3



每次找到一条增广路，把path上面所有边的状态取反（匹配边变成非匹配边，非匹配边变成匹配边），那么得到的新的边集S' 还是一组匹配，并且匹配边数增加了 1. 进一步可以得到推论：二分图的一组匹配 S 是最大匹配，当且仅当图中不存在 S 的增广路。

匈牙利算法

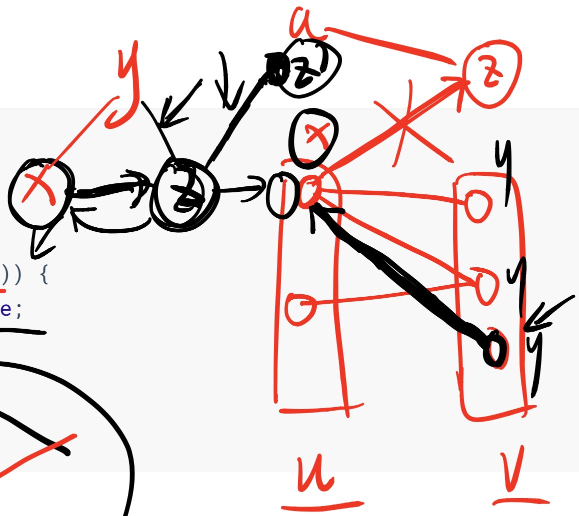
1. 设 $S=\emptyset$，即所有边都是非匹配边
2. 寻找增广路 path， 把路径上所有边的匹配状态取反，得到一个更大的匹配 S'
3. 重复 2，直至图中不存在增广路

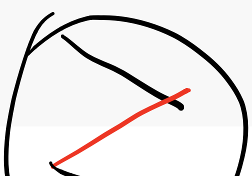
匈牙利算法基于贪心思想，重要特点是：一个节点成为匹配点之后，最多只会因为找到增广路而更换匹配对象，但是绝不会再变回非匹配点。

比较详细的过程演示：https://www.luogu.com.cn/blog/fusu2333/post-2018-wu-yi-qing-bei-pei-xun-er-fen-tu-xiong-ya-li-suan-fa-post

复杂度$O(NM)$，N 为 点数，M 为边数

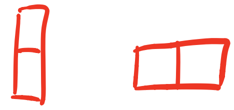
```
bool dfs(int x){
    for(int i=head[x];i;i=nxt[i]) {
        if(!vis[y = ver[i]]) {
            vis[y] = 1;
            if(!match[y] || dfs(match[y])) {
                match[y] = x; return true;
            }
        }
    } return false
}
```





```
// main 中
for(int i=1;i<=n;i++) {
    memset(vis, 0, sizeof vis);
    if(dfs(i)) ans ++;
}
```

u+v=n



[例题1] **棋盘覆盖**

给定一个N行N列的棋盘，已知某些格子禁止放置。

求最多能往棋盘上放多少块的长度为2、宽度为1的骨牌，骨牌的边界与格线重合（骨牌占用两个格子），并且任意两张骨牌都不重叠。

输入格式

第一行包含两个整数N和t，其中t为禁止放置的格子的数量。

接下来t行每行包含两个整数x和y，表示位于第x行第y列的格子禁止放置，行列数从1开始。

输出格式

输出一个整数，表示结果。

数据范围

$1 \le N \le 100$

输出样例:

```
8 0
```

输出样例:

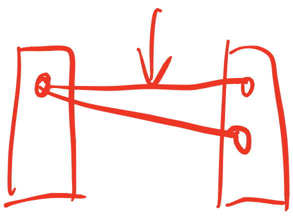
```
32
```

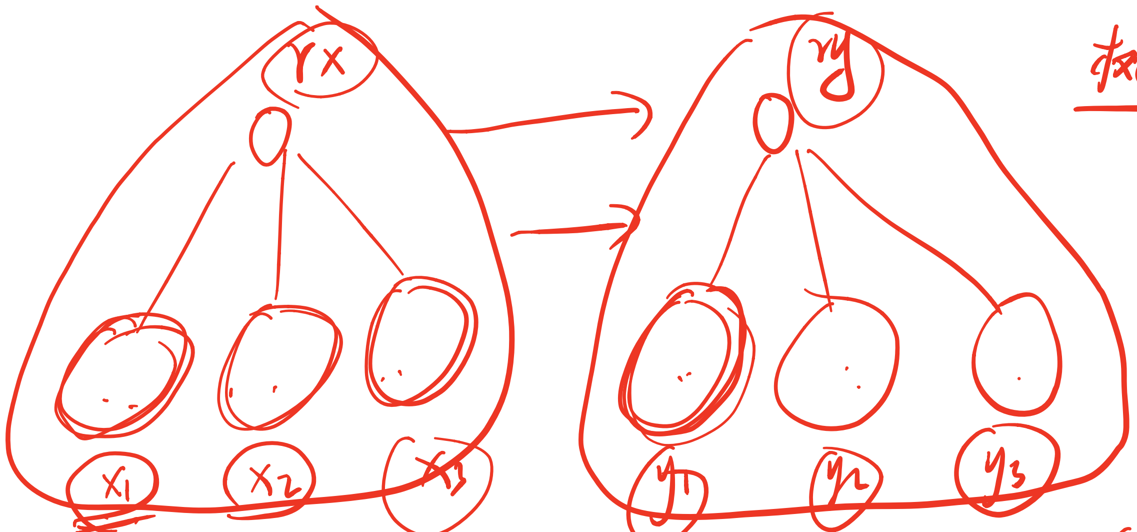
二分图模型两个重要要素：

1. 节点能够分成独立的两个集合，每个集合内部没有边
2. 每个节点只能与 1 条匹配边相连

二分图.

匹配.



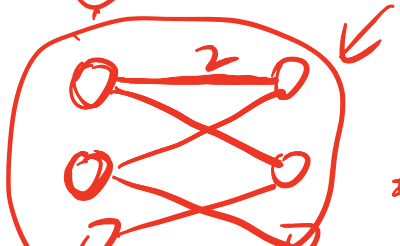


树形DP

[3][3]

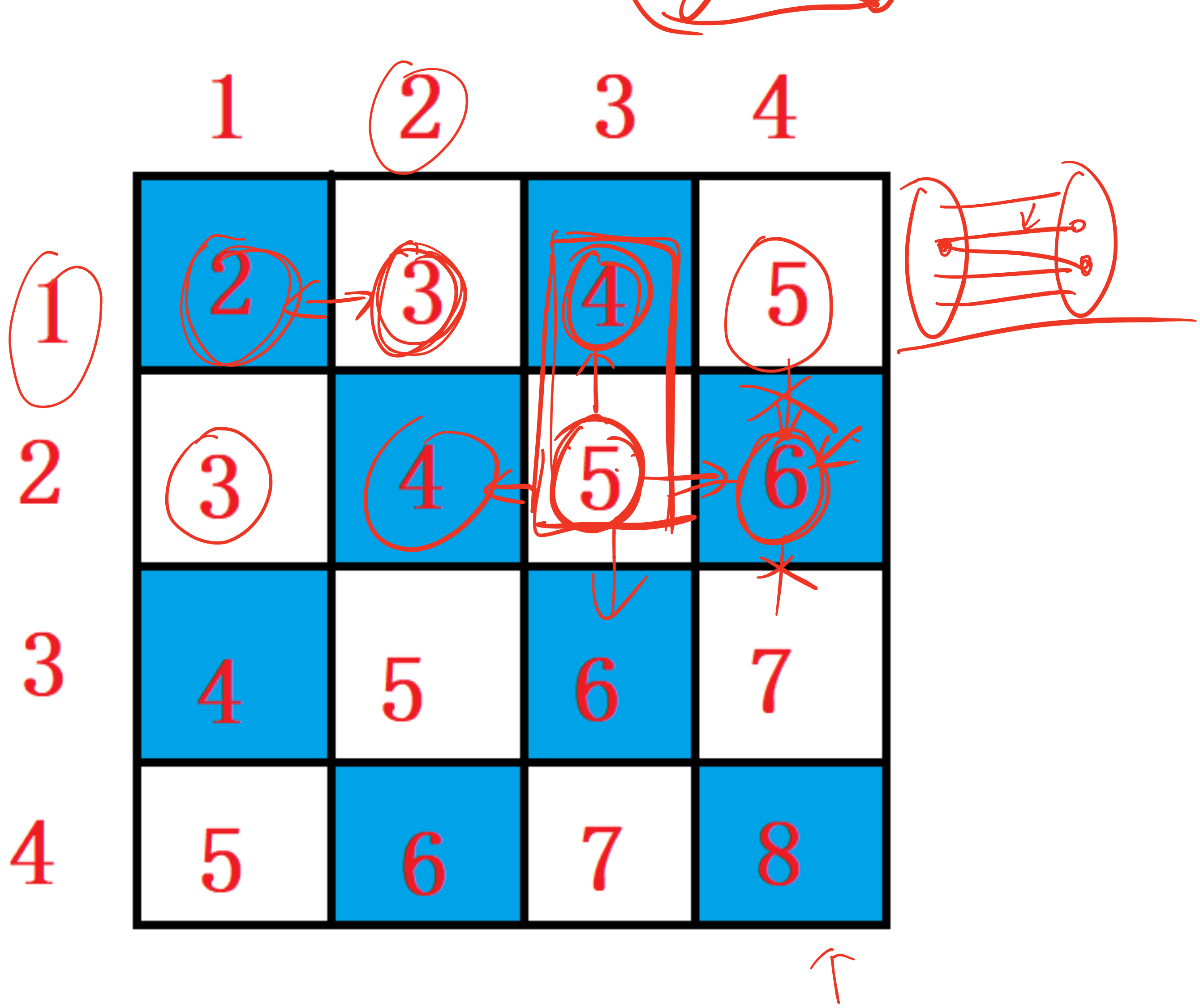
d[x][y]=2

d[x][y]=1



n

最小权值

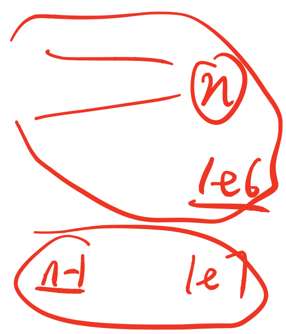


1. 行号和列号之和为偶数染蓝色，否则不染色。同色格子之间没有边满足条件一
2. 每个节点最多只能和相邻的一个节点匹配（当然在这个题目中这两个节点都不能被禁止），满足条件二

```
const int N = 100 + 5;
int a[N][N], vis[N * N], match[N * N];
vector<int> v[N*N];
int n, t, x, y;
int dx[4] = { 1, -1, 0, 0 };
int dy[4] = { 0, 0, 1, -1 };
int id(int x,int y){
    return x * n + y;
}
bool check(int x,int y){
    if(x < 0 || x >= n || y < 0 || y >= n || a[x][y])
        return false;
    return true;
}
bool dfs(int x){
    for (int i = 0,y ; i < v[x].size();i++){
        if(!vis[y = v[x][i]]){
            vis[y] = 1;
            if(!match[y] || dfs(match[y])){
                match[y] = x;
                return true;
            }
        }
    }
    return false;
}
int main() {
    cin >> n >> t;
```

```
    for (int i = 1; i <= t;i++){
        cin >> x >> y;x--;y--;
        a[x][y] = 1;
    }
    for (int i = 0; i < n;i++){
        for (int j = 0; j < n;j++){
            if(a[i][j])
                continue;
            for (int k = 0; k < 4;k++){
                int nx = i + dx[k];
                int ny = j + dy[k];
                if(check(nx,ny)){
                    v[id(i, j)].push_back(id(nx, ny));
                    v[id(nx, ny)].push_back(id(i, j));
                }
            }
        }
    }
    int res = 0;
    for (int i = 0; i < n * n;i++){
        memset(vis, 0, sizeof vis);
        if(dfs(i))
            res++;
    }
    cout << res/2 << endl;
    return 0;
}
```

二分图其他问题



相关问题可以参考《算法竞赛进阶指南》或者洛谷博客：**二分图网络流学习笔记**

1. 二分图带权匹配

   二分图每条边都有一个权值，求该二分图的一组最大匹配，并且匹配边的权值总和最大。

   解法：费用流或者KM（KM必须在满足"带权最大匹配一定是完备匹配"的二分图中求解）

2. 二分图最小点覆盖

   给订一张二分图，求出一个最小的点集S，使得图中任意一条边都有至少一个端点属于S。

   人话：找到一组点，能够覆盖所有的边。

   König定理：二分图最小点覆盖包含的点数等于二分图最大匹配包含的边数

   构造

3. 二分图最大独立集

   独立集：任意两点之间都没有边相连。

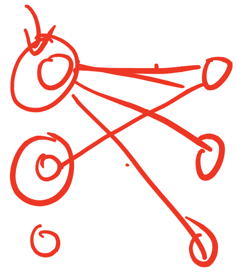
   最大独立集：点数最多的独立集。

   定理：n 个节点的二分图，最大独立集的大小等于 n - 最大匹配数

4. 有向无环图的最小路径点覆盖

   给定一张有向无环图，要求用尽量少的不相交简单路径，覆盖有向无环图的所有顶点。

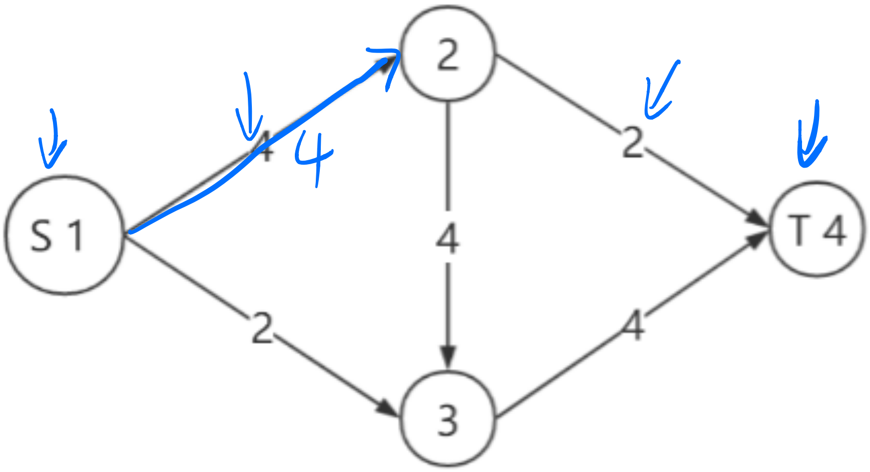
   需要建立拆点二分图，然后求解最大匹配。



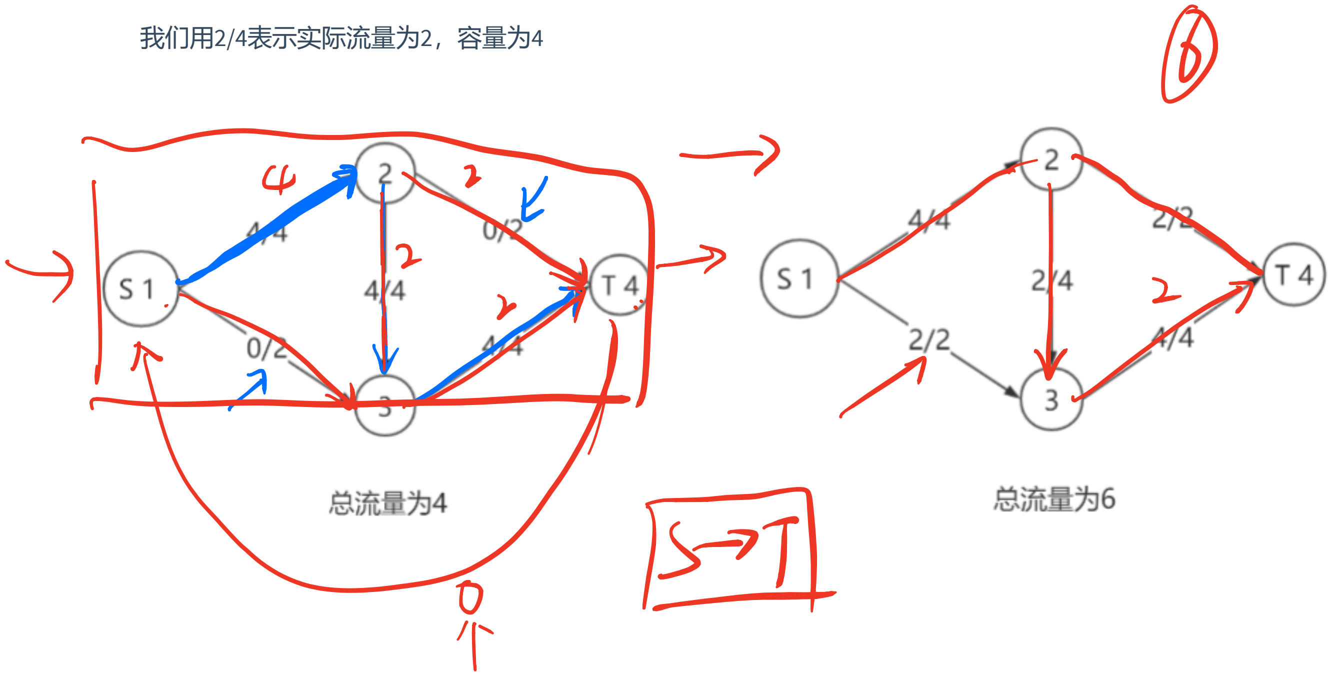
网络流

问题导入

有一个自来水管道输送系统，起点是S，目标是T，途中经过的管道都有最大流量限制（容量）。



我们用2/4表示实际流量为2，容量为4



总流量为4　　　　总流量为6

- 模型

  一个网络 $G=(V,E)$ 是一张有向图，图中每条有向边$(x,y)\in E$ 都有一个给定的权值 $c(x,y)$ ，称为边的容量。

  特别的，若$(x,y)\notin E$ 则 $c(x,y)=0$ 。图中特殊节点$S\in V$ 和 $T\in V(S\neq T)$ ，分别称为源点和汇点。

- 流量概念

  设 $f(x,y)$ 是定义在节点二元组 $(x\in V,y\in V)$ 上的实数函数，且满足:

  1. $f(x,y)\le c(x,y)$ 边流量小于边容量
  2. $f(x,y)=-f(y,x)$ 对称性
  3. $\forall x\neq S,x\neq T,\sum_{(u,x)\in E}f(u,x)=\sum_{(x,v)\in E}f(x,v)$ 流量平衡

  $x\to y = -y\to x$

  f 称为网络的流函数。对于$(x,y)\in E,f(x,y)$ 称为边的流量，$c(x,y)-f(x,y)$ 称为边的剩余容量。

  $\sum_{(S,v)\in E}f(S,v)$ 称为整个网络的流量（其中 S 表示源点）

$\sum_{(v,T)\in E} f(v,T)$

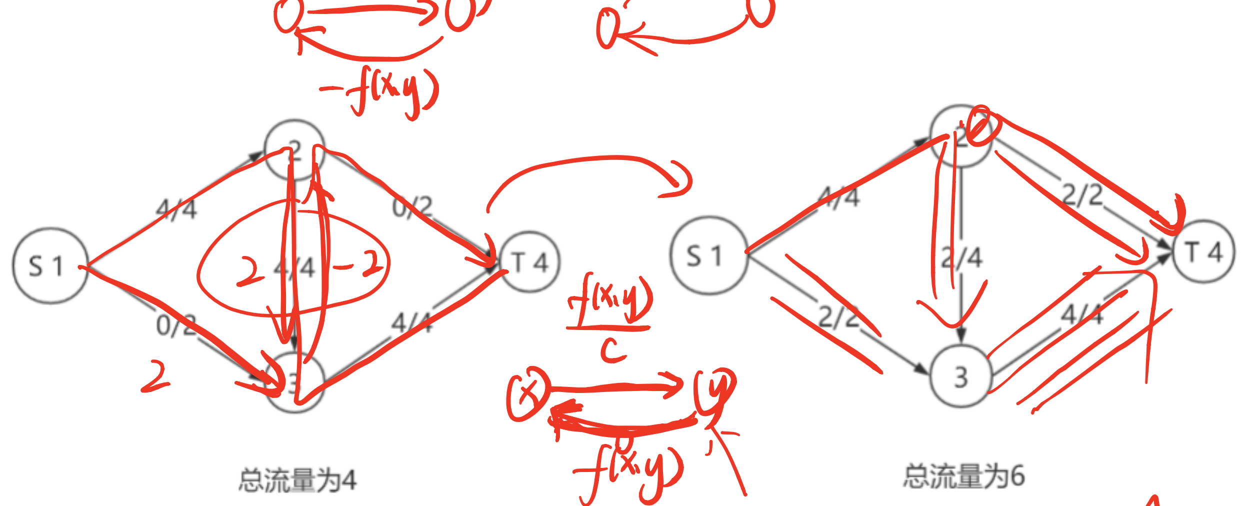
增广路算法(思想)

$S\to T$

1. 初始所有边流量都为0,
2. 找到一条从S,T 的有向路径，并且路径上面所有边的残量大于 0，称为增广路。路径上残量的最小值称为可改进量。
3. 总流量加上可改进量，增广路上所有边的流量加上可改进量（残量减去可改进量）
4. 回到 2，直到找不到增广路为止。

退流与反边

$f(x,y)$



总流量为4　　　　总流量为6

左图中，应当找到一条增广路：$1 \to 3 \to 2 \to 4$，可改进流量为 2

$2 \to 3$ 这条边的流量由 4 变成 2，这种现象称为退流（后悔操作）

2 3　$2 \wedge 1 = 3$　$3 \wedge 1 = 2$

对于$2 \to 3$ 这条有向边，在存边时，需要存两条边，分别是 $2 \to 3$，剩余容量为4的边 和 $3 \to 2$ 剩余容量为 0 的边，即正反两条边（一般称为弧）

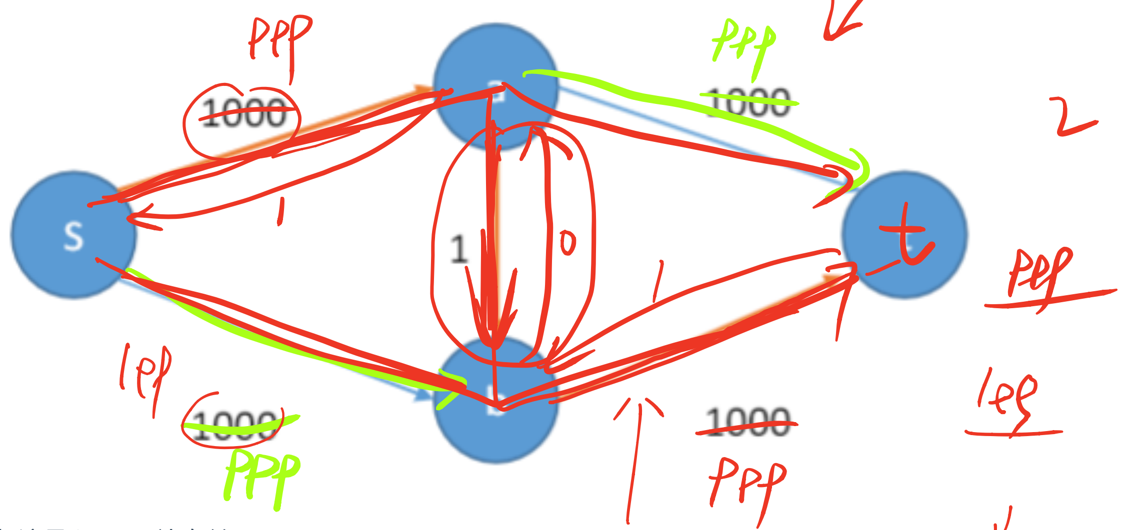
当一条弧的剩余流量减少时，反向边的剩余流量应当增加。

边的存储一般使用链式前向星，例如一条边可以存为编号为 2,3 的正反边，通过异或操作即可$O(1)$ 获得反向边

Ford - Fulkerson算法

1. 初始所有边的剩余流量为边的初始容量
2. 从S出发，沿着剩余流量大于 0 的弧进行DFS，直到遇到汇点 T。S到T存在一条增广路，进行增广
3. 返回2，直到无法从S到达T

看似正确，但是遇到了下面这种情况，就比较棘手了：



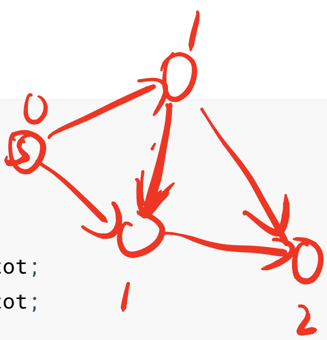
复杂度与流量和DFS寻边有关 $O(E \max f)$

Edmonds-Karp 算法

$O(nm^2)$　$O(nm)$ 增广路　$O(m)$

简称EK算法，将FF算法中DFS寻找增广路变为BFS，由于是BFS，每次找到的增广路，都是基于最短路的（每条边边权为1），所以会相对DFS有些许优化。

复杂度$O(nm^2)$ 但一般远远达不了上界，一般能处理 $10^3 \sim 10^4$ 级别的网络

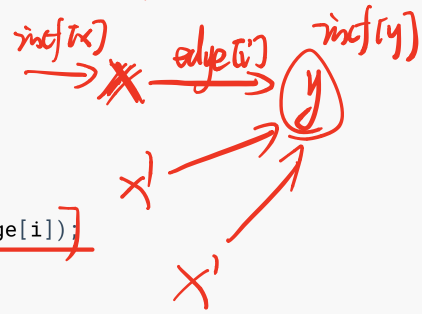


```
const int inf = 1 << 29, N = 2010, M = 20010;
int head[N],ver[M],edge[M],nxt[M],v[N],incf[N],pre[N];
int n,m,s,t,tot,maxflow;
void add(int x, int y, int z){
    ver[++tot] = y, edge[tot] = z, nxt[tot] = head[x], head[x] = tot;
    ver[++tot] = x, edge[tot] = 0, nxt[tot] = head[y], head[y] = tot;
}
```

```
bool bfs(){
    memset(v,0,sizeof v);
    queue<int> q;
    q.push(s); v[s] = 1;
    incf[s] = inf;
    while(q.size()){
        int x = q.front(); q.pop();
        for(int i=head[x];i;i=nxt[i]){
            if(edge[i]){
                int y = ver[i];
                if(v[y])continue;
                incf[y] = min(incf[x], edge[i]);
                pre[y] = i;
                q.push(y), v[y] = 1;
                if(y == t) return 1;
            }
        }
    }
    return 0;
}
void update(){
    int x = t;
    while(x != s){
        int i = pre[x];
        edge[i] -= incf[t];
        edge[i ^ 1] += incf[t];
        x = ver[i ^ 1];
    }
    maxflow += incf[t];
}
int main(){
    while(cin >> n >> m){
        memset(head,0,sizeof head);
        s = 1, t = n, tot = 1, maxflow = 0;
        for(int i=1;i<=m;i++){
            int x,y,z;scanf("%d%d%d",&x,&y,&z);
            add(x,y,z);add(y,x,z);
        }
        while(bfs()) update();
        cout << maxflow << endl;
    }
}
```

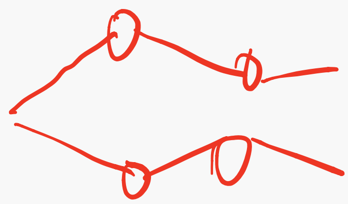
$inf \to S$

edge[i]



O(n)

EK

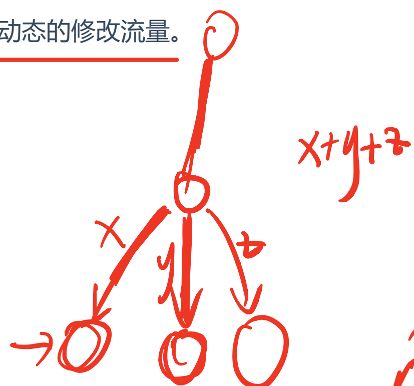


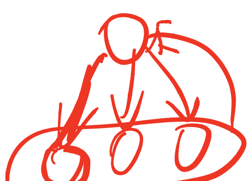
Dinic 算法

EK算法每次BFS只会找到一条增广路，还有进一步的优化空间。网络上所有节点以及剩余容量大于 0 的边构成的子图被称为残量网络。

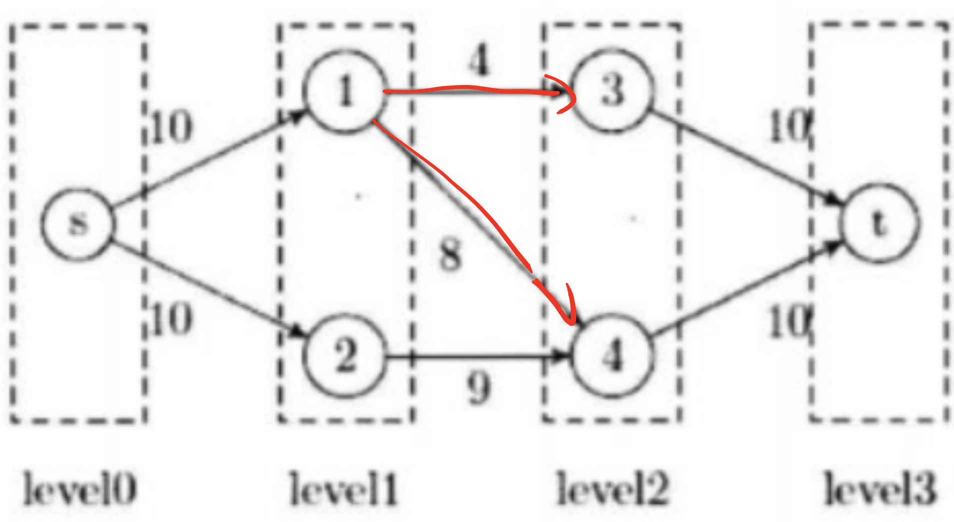
从 S 出发，在残余网络上BFS，按照节点到S的距离分类，可以将残余网络变为一张分层图（是一个DAG有向无环图）

然后从S出发DFS去寻找增广路，动态的修改流量。





DAG



$Dinic$ 算法不断的重复以下步骤，直到在残余网络中 S 不能到达 T：

$O(n^2m)$

1. 在残量网络上 BFS 求出节点的层次，构造分层图
2. 在分层图上DFS寻找增广路，回溯时实时更新剩余容量。

时间复杂度 $O(n^2m)$，一般远远达不到上界，能处理 $O(10^4 \sim 10^5)$ 规模的网络。另外Dinic 求解二分图最大匹配的复杂度为 $O(m\sqrt{n})$ $O(nm)$

EK

```
const int inf = 1<<29, N = 50010,M=30010;
int head[N], ver[M], edge[M], nxt[M], d[N];
int n, m, s, t, tot, maxflow;
queue<int> q;
void add(int x, int y, int z){
    ver[++tot] = y, edge[tot] = z, nxt[tot] = head[x], head[x] = tot;
    ver[++tot] = x, edge[tot] = 0, nxt[tot] = head[y], head[y] = tot;
}
// 建立分层图
bool bfs(){
    memset(d, 0, sizeof d);
    while(q.size())q.pop();
    q.push(s);d[s] = 1;
    while(q.size()){
        int x = q.front();q.pop();
        for(int i=head[x];i;i=nxt[i]){
            if(edge[i] && !d[ver[i]]){
                q.push(ver[i]);
                d[ver[i]] = d[x] + 1;
                if(ver[i] == t) return 1;
            }
        }
    }
    return 0;
}
// 表示从流入 x 的流量为flow
int dinic(int x, int flow){
    if(x == t) return flow;
    int rest = flow, k;
    // 循环条件中的 rest 是很关键的剪枝，如果流量从x都流出去后，没有再遍历的必要
    for(int i=head[x];i && rest; i=nxt[i]){
        if(edge[i] && d[ver[i]] == d[x] + 1){
            k = dinic(ver[i], min(rest, edge[i]));
            if(!k) d[ver[i]] = 0; // 剪枝，如果ver[i] 已经没用了，就将ver[i] 从分层图中删除掉
            edge[i] -= k;
            edge[i ^ 1] += k;
```

rest

edge[i]

DAG

flow

```
                rest -= k;
            }
        }
    return flow - rest;
}
int main(){
    cin >> n >> m;
    cin >> s >> t;
    tot = 1;
    for(int i=1;i<=m;i++){
        int x, y, c;scanf("%d%d%d",&x,&y,&c);
        add(x,y,c);
    }
    int flow = 0;
    while(bfs())
        while(flow = dinic(s,inf)) maxflow += flow;
    cout << maxflow << endl;
}
```

② 3

rest 反向边

①

最小割问题

给定一个网络 $G=(V,E)$，源点为 S, 汇点为T。若一个边集 $E' \subseteq E$ 被删去之后，源点 S 和汇点 T 不再联通，则称该边集为网络的割。边的容量之和最小的割称为网络的最小割

最大流最小割定理

任意一个网络的最大流量等于最小割中边的容量之和，简记为"最大流 = 最小割"

$\Sigma w(x,y)$

费用流

给定一个网络 $G=(V,E)$，每条边(x,y) 除了有容量限制$c(x,y)$,还有一个给定的"单位费用" $w(x,y)$

当边(x,y) 的流量为$f(x,y)$ 时，就要花费 $f(x,y)*w(x,y)$ 。该网络中总花费最小的最大流被称为 最小费用最大流，总花费最大的最大流被称为"最大费用最大流"，二者合称为"费用流"模型。注意：费用流的前提是最大流，然后才考虑费用的最值。

费用流可以用来求解 二分图带权最大匹配问题，每条边权值就是它的单位费用。

使用EK算法求解费用流，只需将BFS寻找任意一条增广路更改为使用SPFA寻找一条单位费用之和最小的增广路。

注意建图时，反向边费用为$-w(x,y)$

$O(nm \cdot spfa)$

```
const int N = 5000 + 5;
const int M = 100010;
int head[N], ver[M], nxt[M], cost[M], edge[M];
int d[N], v[N], pre[N], incf[N];
int n, m, s, t, tot;
int maxflow, ans;
void add(int x, int y, int z, int c){
    ver[++tot] = y, nxt[tot] = head[x], edge[tot] = z, cost[tot] = c, head[x] = tot;
    ver[++tot] = x, nxt[tot] = head[y], edge[tot] = 0, cost[tot] = -c, head[y] = tot;
}
bool spfa(){
    memset(d, 0x3f, sizeof d); // d[i] 表示从s到t的最小费用和，使用SPFA求出
    memset(v, 0, sizeof v);
    d[s] = 0; v[s] = 1;
    incf[s] = inf;
    queue<int> q;
    q.push(s);
    while(q.size()){
        int x = q.front();q.pop();
        v[x] = 0;//注意这里
```

v[x]

```
        for(int i=head[x];i;i=nxt[i])if(edge[i]){
            int y = ver[i];
            if(d[y] > d[x] + cost[i]){
                d[y] = d[x] + cost[i];
                incf[y] = min(incf[x], edge[i]); if
                pre[y] = i;
                if(!v[y]) v[y] = 1, q.push(y);
            }
        }
    }
    return d[t] != inf;    t
}
void update(){
    int x = t;
    while(x != s){
        int i = pre[x];
        edge[i] -= incf[t];
        edge[i^1] += incf[t];
        x = ver[i^1];
    }
    maxflow += incf[t];
    ans += incf[t] * d[t];
}
int main(){
    tot = 1;
    scanf("%d%d%d%d", &n, &m, &s, &t);
    for(int i=1;i<=m;i++){
        int x, y, z, c;scanf("%d%d%d%d", &x, &y, &z, &c);
        add(x, y, z, c);
    }
    while(spfa()) update();
    printf("%d %d\n", maxflow, ans);

    return 0;
}
```