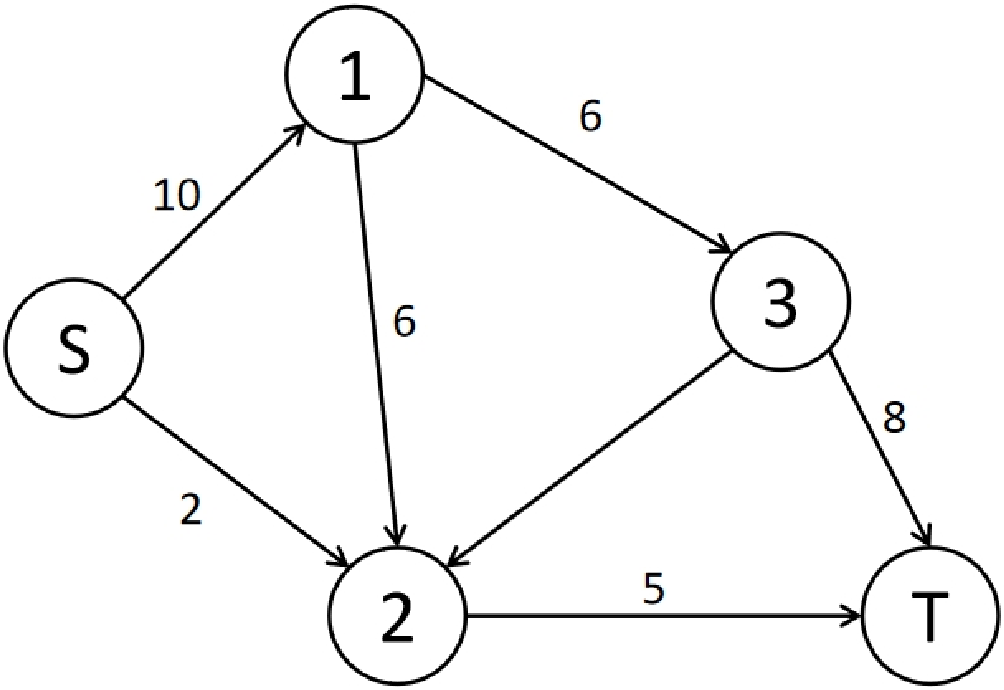
网络流初探

一、模型引入

我们可以将有向图看作网络流图来解决某一类型的问题。网络流适合用来模拟水流从起点通过复杂的网络流向终点的过程。

参看下图，给定一个有向图 $G=(V,E)$，把图中的边看作管道，每条边上有一个流量上限。对于图中每条边 $e \in E$，我们记它的流量上限为 $c(e)$，流过的流量为 $f(e)$。

给定源点 s 和汇点 t，现在假设在 s 处有一个水源，t 处有一个蓄水池，其他点都是中转点（不消耗也不增加流量）。问从 s 到 t 的最大水流量是多少，类似于这类的问题都可归结为网络流问题。

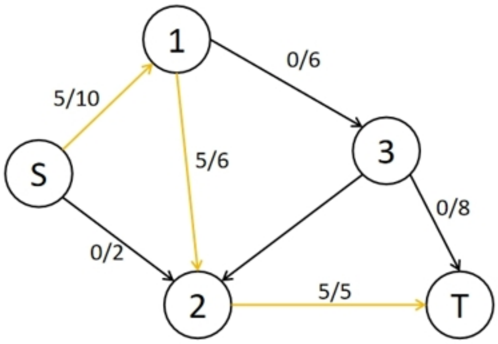


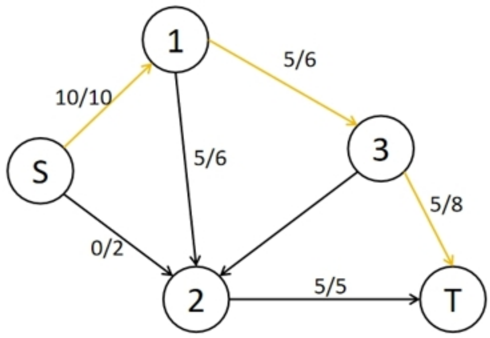
这个水流必定满足下列两个条件：

1. **流量约束条件：** $\forall e \in E, 0 \leq f(e) \leq c(e)$
2. **流量平衡条件：** $\forall v \in V(v \neq s,t), \sum_{e=(u,v)\in E} f(e) = \sum_{e=(v,w)\in E} f(e)$
   $s \to t$ 的水流量可以表示为$\sum_{e=(s,v)\in E} f(e)$,我们要最大化的就是这个值。

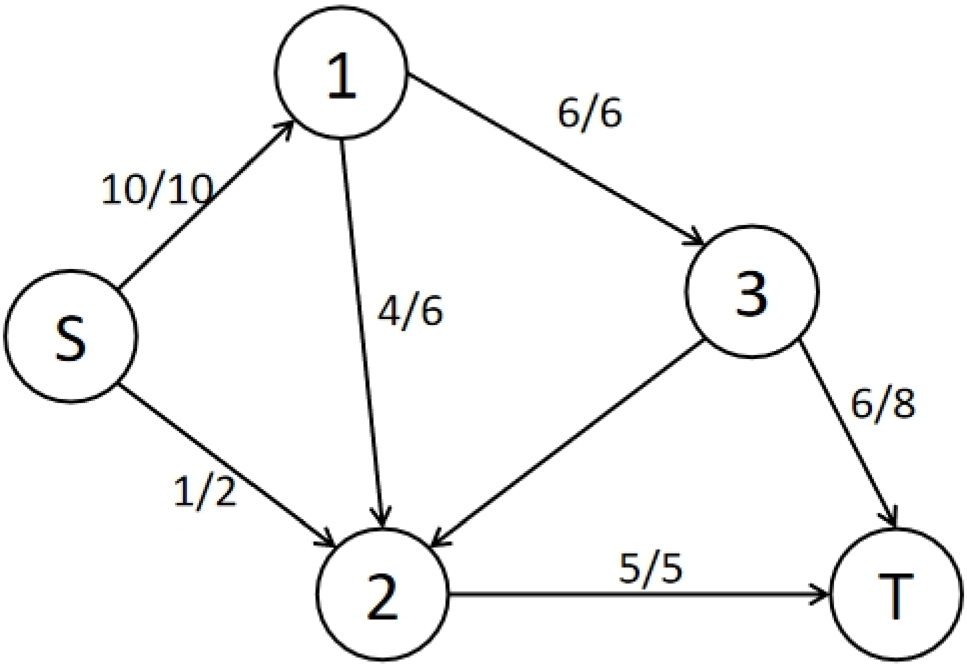
对于这张图如何寻找最大流，我们首先考虑下面的这个贪心算法：

1. 找一条 s 到 t 的只经过 $f(e) < c(e)$ 的边的路径。
2. 如果不存在满足条件的路径，则算法结束。否则，沿着该路径尽可能地增加 $f(e)$, 返回第 1 步。

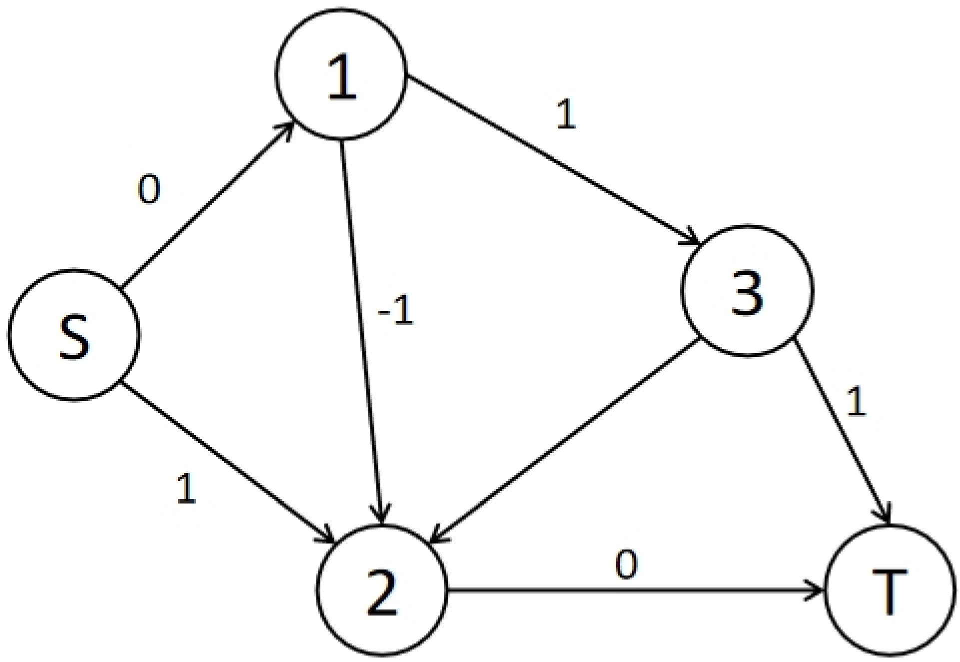




可见贪心能得到最大流量 10，那么这样得到的是最大流吗？事实上采用下图的方案更优，所以这个贪心是不正确的。



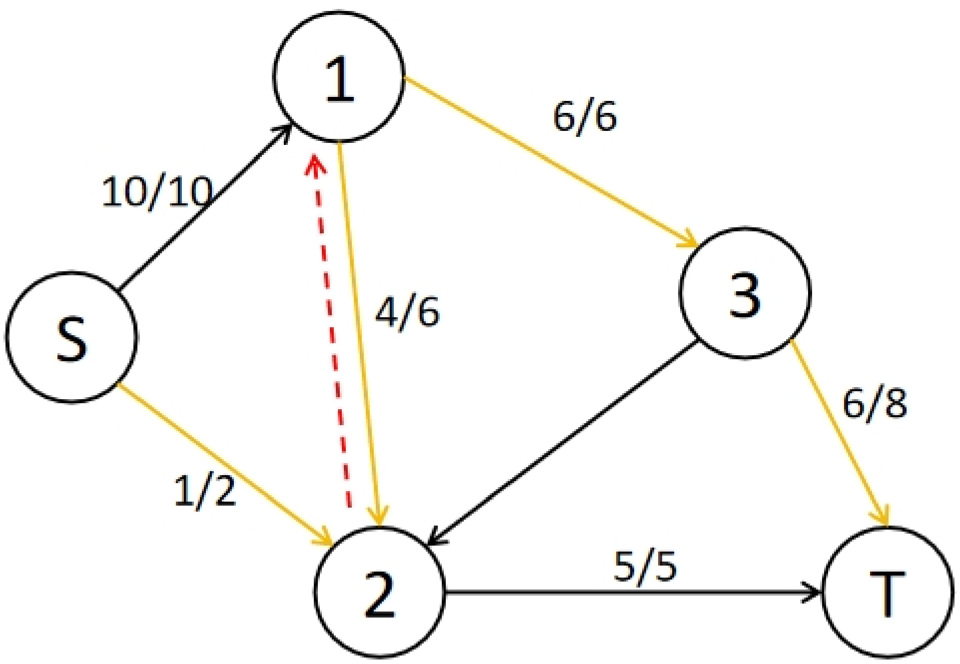
为了找出二者的区别，不妨来看看它们的流量差。



通过对流量差的观察可以发现，我们通过将原先得到的流给退回去（图中的 -1）,而得到新的流（即 **退流** 操作）。因此可以试着在之前的贪心算法中加上这一步骤，算法改进如下：

1. 只利用满足 $f(e) < c(e)$ 的 e 找一条 s 到 t 的路径。
2. 如果不存在满足条件的路径，则算法结束。否则，沿着该路径尽可能地增加流，重复第 1 步。

将改进后的贪心算法用于样例：



这个算法总能求得最大流，我们将会在后面证明。

我们称在步骤 1 中所考虑的 $f(e) < c(e)$ 的 e 和满足 $f(e) > 0$ 的 e 的反向边 $rev(e)$ 所组成的图为**残量网络**，残量网络上的 $s \to t$ 路径也就是**增广路**。

二、基本概念

下文将通过给出有关的数学定义和证明，来总结得到求解最大流的常用算法。

(1) 网络

网络 $G=(V,E)$ 是一张有向图，图中每条边都有给定的权值（下文中称为容量）。图中还有两个指定的特殊点 $s,t(s\neq t)$，称为**源点**和**汇点**。其中源点没有入边，汇点没有出边。

(2)容量和流量

为了方便处理 **反向边**，下面给出与 **模型引入** 有所不同的定义。
对于每个结点二元组 (u,v)，定义它们的容量和流量：
容量函数 c 是定义在结点二元组 $(u,v)(u\in V,v\in V)$ 上的非负实数函数

$$\begin{aligned}\forall(u,v)\in E,\quad c(u,v)>0\\ \forall(u,v)\notin E,\quad c(u,v)=0\end{aligned}$$

流量函数 f 是定义在结点二元组 $(u,v)(u\in V,v\in V)$上的实数函数，且满足：

1. **容量限制：** $\forall(u,v),f(u,v)\leq c(u,v)$。
2. **斜对称性：** $\forall(v,u),f(u,v)=-f(v,u)$（可以理解成流的方向）。3
3. **流守恒性：** 除了源点是流的提供点，汇点是流的收集点，其他点都是 **中转站**，既不增加也不减少流量（下面的斜杠 \ 表示相对补集 $B\backslash A=x\in B|x\notin A$)

$$\forall x\in V\backslash\{s,t\},\sum_{(u,x)\in E}f\left(u,x\right)=\sum_{(x,v)\in E}f\left(x,v\right)$$

也可以写成

$$\forall x\in V\backslash\{s,t\},\sum_{(x,y)}f\left(x,y\right)=0$$

对于(u,v)，$c(u,v)$ 称为**边的容量**，$f(u,v)$ 称为**边的流量**，$c_f(u,v)=c(u,v)-f(u,v)$称为**边的剩余容量。**
整个**网络的流量**为$|f|=\sum vf(s,v)$，即从源点发出的所有流量之和。

(3)残量网络和增广路

对于 G 上的流函数 f，定义**残量网络** $G_f=(V_f=V,E_f=\{(u,v)\mid c_f(u,v)>0\})$。
那么残量网络上任意一条 $s\rightarrow t$ 的路径就可以成为**增广路。**
注意这里满足 $(u,v)\notin E,c_f(u,v)>0$ 的边也可能存在于残量网络，即反向边。

三、Edmond-Karp 算法及其正确性

(1) Edmond-Karp 算法

模型引入 中 **改进后的贪心算法** 就是每次找到一条增广路，然后通过 **正向边能增加的流量** 和 **反向边中能退流的流量** 来计算出当前增广路能增加流的最大值。因为我们将剩余容量定义为 $c_f=c-f$，所以这个 **能退流的流量** 也被包含在了 c_f 当中，我们处理正向边和反向边的时候就没有区别了。

因此上面的 **改进后的贪心算法** 就可以这样描述：

- 每次在当前流 f 的残量网络 G_f 上找到一个增广路 $s\rightarrow t$;
- 将增广路 $(x_1=s,x_2,x_3,\cdots,x_{k-1},x_k=t)$ 描述成一个流函数 f_0，使得对于$1\leq i<k$:

$$f_0(x_i,x_{i+1})=+\min_{1\leq j<k}\{c_f\big(x_j,x_{j+1}\big)\}$$

$$f_0(x_{i+1},x_i)=-\min_{1\leq j<k}\{c_f\big(x_j,x_{j+1}\big)\}$$

- 而其余函数值均为 0。
- 得到一个流量更大的流函数 $f' = f + f_0$。
- 不断重复此过程，直到残量网络上不存在增广路为止，我们就找到了一个最大流。
- 上面这个算法被称为 $Edmond - Karp$ 算法。

代码实现

```
const int N = 205;
const int M = 405;
const int INF =0x3f3f3f3f;
int src, des;
int pre[N];  // pre储存每个点在增广时进入它的边
bool vis[N];
int ecnt =1, adj[N], nxt[M], go[M], cap[M];
// ecnt从2开始计数，这样e的反向边为e^1方便处理
// cap表示这条边剩余的容量，即上文的 c_f(u,v)
//连一条u->v容量为w的边
inline void addEdge(const int &u, const int &v, const int &w) {
    nxt[++ecnt] = adj[u], adj[u] = ecnt, go[ecnt] = v, cap[ecnt] = w;
    nxt[++ecnt] = adj[v], adj[v] = ecnt, go[ecnt] = u, cap[ecnt] = 0;
}
//寻找增广路
bool Bfs() {
    static int qN, que[N];
    int u, v, e;
//存储的时候源点编号最小，汇点编号最大，因此初始化可以这样枚举进行
    for (int i = src; i <= des; ++i) vis[i] = false, pre[i] = 0;
    que[qN = 1] = src, vis[src] = true;
    for (int ql = 1; ql <= qN; ++ql) {
        u = que[ql];
        for (e = adj[u]; e; e = nxt[e]) {
            if (cap[e] > 0 && !vis[v = go[e]]) {
                pre[v] = e, vis[v] = true;
                que[++qN] = v;
                if (v == des) return true; //找到了最短增广路
            }
        }
    }
    return false;
}
//增广
int Augment() {
    int d = INF;
    for (int u = des, e; u != src; u = go[e ^ 1]) {
        e = pre[u];
        d = std:: min(d, cap[e]);
        //计算出最多能增广的流量
    }
    for (int u = des, e; u != src; u = go[e ^ 1]) {
        e = pre[u];
        cap[e] -= d, cap[e ^ 1] += d;
        //处理正向边和反向边的剩余容量
    }
    return d;
}
```

```
int maxFlow() {
    int ans = 0;
    while (Bfs()) ans += Augment();
    //不停增广直至没有增广路
    return ans;
}
```

(2)时间复杂度

如果增广路是随便找的，那么这个算法的最坏复杂度为 $O(F|E|)$，其中 F 是最大流的大小（这里默认容量都是正整数）。

如果用 BFS 来找增广路，找到最短增广路就开始增广，那么时间复杂度是 $O(|V||E|^2)$ 的（即上文参考实现的算法）。下面给出证明：

1. 首先证明 **每次寻找的增广路长度是不降的。**
   - 假设残量网络 G_f 经过一次增广操作后变成 $G_{f'}$，记 $d(u,v)$ 表示 G_f 中 $u \to v$ 最短距离，$d'(u,v)$ 表示 $G_{f'}$ 中 $u \to v$ 最短距离。下面用反证法证明 $\forall v \in V, d'(s,v) \geq d(s,v)$。
   - 假设存在 v 满足其增广后 $s \to v$ 最短路径距离减少，且规定 v 取满足这一条件中 $d'(s,v)$ 最小的点，则 $d'(s,v) < d(s,v)$。
   - 令 u 表示 $G_{f'}$ 中 $s \to v$ 最短路径上 v 的前一个点，则 $d'(s,u)+1 = d'(s,v)$
   - 且 u 的最短距离未减小，因此有

   $$d(s,u)+1 \leq d'(s,u)+1 = d'(s,v) < d(s,v)(*)$$

   - 分两种情况讨论：
     a. $(u,v) \in E_f$，则 $d(s,u)+1 \geq d(s,v)$，与 $(*)$ 矛盾。
     b. $(u,v) \notin Ef$，但是 $(u,v) \in G_f$， 因此在 $Gf \to G_{f'}$， 的增广过程中经过了 (v,u)，因此 $d(s,v)+1 = d(s,u)$， 和 $(*)$ 矛盾，因此该情况也不成立。
   - 综上，$\forall v \in V, d'(s,v) \geq d(s,v)$。
2. 然后证明增广次数 $O(|V||E|)$。

- 定义某次增广时 $c_f(u,v)$ 最小的边 (u,v) 为 **关键边**，则增广后 $(u,v) \notin E_{f'}$ 。之后 (u,v) 要再次出现在残量网络的前提是要沿着 (v,u) 增广。
- 让 (u,v) 在残量网络 G_f 消失的增广中有 $d(s,u)+1 = d(s,v)$。
- 让 (u,v) 在再次出现的增广中有 $d'(s,u) = d'(s,v)+1$。
- 由最短增广路长度不下降得，$d'(s,u) = d'(s,v)+1 \geq d(s,v)+1 = d(s,u)+2$。
- 最短增广路长度不超过 $|V|-1$，故一条边成为关键边次数至多为 $\frac{|V|-1}{2}$ ，每次增广时至少有一条关键边，而 $|Ef| \leq 2|E|$，因此增广次数 $O(|V||E|)$。

(3)最大流最小割定理

为了证明上述算法所求得的确实是最大流，我们首先介绍割这一概念。

在图 $G=(V,E)$ 上，对于某个顶点集合 $P \subseteq V$， 割 $(P, V \backslash P)$ 被定义为 $(P \times (V \backslash P)) \cap E$，即 **边集 E 中起点在 P 终点在 $V \backslash P$ 构成的所有边的集合**。这些边的容量之和被称为割的容量（这里的割并不包括反向边）。

若 $s \in P \wedge t \in V \backslash P$，则这个割又被称为 $s-t$ 割，下面我们简单记 (S,T) 表示一个 $s-t$ 割。

如果将网络中 $s-t$ 割所包含的边都删去，也就不再有从 s 到 t 的路径了。因此可以考虑一下如下问题：对于给定网络，为了保证没有从 s 到 t 的路径，需要删去的边的总容量的最小值是多少？ 该问题

即为**最小割问题。**

形式化地，定义 **割的容量**：

$$C(P,V\backslash P)=\sum_{u\in P}\sum_{v\in V\backslash P}c\left(u,v\right)$$

使得 $|C(S,T)|$ 最小的 $s-t$ 割称为 $s-t$ 的最小割。

先证明：对于同一个网络的任意一个流 f 和任意一个 $s-t$ 割 (S,T)，$|f|\le c(S,T)$。

- 因为$|f|=\sum_{(s,v)\in E}f(s,v)$
- 而对于任意 $x\in S\backslash\{s\}$, 都有 $\sum_{(u,x)\in E}f(u,x)-\sum_{(x,v)\in E}f(x,v)=0$ 。
- 因此可以得到

$$|f|=\sum_{u\in S}\sum_{v\in T}f\left(u,v\right)\le\sum_{u\in S}\sum_{v\in T}c\left(u,v\right)=c\left(S,T\right)$$

接下来我们考虑通过上述贪心算法所求得的流 f。记流 f 对应的残量网络中从 s 可达的 **所有** 顶点组成的集合为 S，其余点构成点集 T。因为 f 对应的残量网络中不存在 $s-t$ 路径（即不存在增广路），因此 (S,T) 就是一个 $s-t$ 割。

根据残量网络的定义，对包含在割中的边 $(u,v)\in(S\times T)\cap E$ 均有 $f(u,v)=c(u,v)$，因此此时构造的流和割就满足：

$$|f|=\sum_{u\in S}\sum_{v\in T}f\left(u,v\right)=\sum_{u\in S}\sum_{v\in T}c\left(u,v\right)=c(S,T)$$

再由任意的流的流量不超过任意 $s-t$ 割的容量可以知道，f 是最大流，(S,T) 是最小割。因此 $Edmond-Karp$ 算法得到的流就是最大流。

同时这也是一个重要的性质：**最大流等于最小割。**

四、Dinic 算法

(1)算法流程

对于上述介绍的方法进行一下总结和优化，就得到了 $Dinic$ 算法。

因为最短增广路的长度在增广过程中始终不会变短，所以无需每次都通过广搜来寻找最短增广路。每一轮增广过程中，我们可以先进行一次广搜，然后考虑由近距离顶点指向远距离顶点的边所组成的分层图，在上面进行深搜寻找最短增广路（这里一次深搜就可以完成多次增广的工作）。则说明最短增广路的长度变长了或不存在增广路了，于是重新构造新的分层图。

相比于 $Edmond-Karp$ 算法，$Dinic$ 算法的优化如下：

①分层图优化

在 $Dinic$ 算法的每一轮增广中，首先对图进行一次 BFS，然后在 BFS 生成的分层图中进行多次DFS 增广（即只访问分层图上的边），这样就切断了原有的图中的许多不必要的连接。同时可以进行多路增广，减少了增广的轮数。

②当前弧优化

在分层图上 DFS 时，若某条边被流满，那么在该轮之后的增广过程中，这条边的剩余容量均为 0 ，等价于从残量网络中被删除。

于是如果我们访问某条边后，发现这条边流满，或者沿着这条边走下去找不到增广路，那么这条边之后就可以不再访问。具体实现时，只需要像下文 **参考程序** 一样，将该点链式前向星的第一条边作为引用传到枚举变量即可。

③ -1 优化

在同一次 DFS 中，如果从一个点引发不出任何的增广路径，就将这个点在层次图中抹去。

代码实现

```
//N为点数，M为边数，INF 代表一个极大值
const int N = 1000;
const int M = 10000;
const int INF = 0x3f3f3f3f;
// src为源点，des为汇点，lev为分层后点的距离标号，cur为当前弧优化所用的临时数组
int src, des, lev[N], cur[N];
// ecnt从2开始计数，这样e的反向边为e^1方便处理
//cap表示这条边剩余的容量
int ecnt = 1, adj[N], nxt[M], go[M], cap[M];
//连一条u->v容量为w的边
void addEdge(const int &u, const int &v, const int &w) {
    nxt[++ecnt] = adj[u], adj[u] = ecnt, go[ecnt] = v, cap[ecnt] = w;
    nxt[++ecnt] = adj[v], adj[v] = ecnt, go[ecnt] = u, cap[ecnt] = 0;
}
//将图分层
bool Bfs() {
    static int qN, que[N];
    int u, v, e;
    //存储的时候源点编号最小，汇点编号最大，因此初始化可以这样枚举进行
    for (int i = src; i <= des; ++i) lev[i] = -1, cur[i] = adj[i];
    que[qN = 1] = src, lev[src] =0;
    for (int ql = 1; ql <= qN; ++ ql) {
        u = que[ql];
        for (e = adj[u]; e; e = nxt[e]) {
             if (cap[e] > 0 && lev[v = go[e]] == -1) {
                lev[v] = lev[u] + 1, que[++qN] = v;
                if (v == des) return true; // 当前还存在增广路
             }
        }
    }
    return false;
}
//从u 点往外尝试流出 flow的流量，返回的是最后成功流出的流量
int Dinic(const int &u, const int &flow) {
    if (u == des) return flow;
    int res = 0, v, delta;
    for (int &e = cur[u]; e; e = nxt[e]) { //&e: 当前弧优化
        //这里e是一个引用，这样随着e = nxt[e], cur[u]也会等于nxt[e],那么原来cur[u]那条边
就不会再被重复检查
        if (cap[e] > 0 && lev[u] < lev[v = go[e]]) { // cap[e]>0: e在残量网络上,
lev[u]< lev[v]: e是层次图上的边
            delta = Dinic(v, std:: min(cap[e], flow - res)); //尝试增广
            if (delta) { //增广成功
                cap[e] -= delta;
                cap[e^1] += delta;//修改前向弧和后向弧的流量
                res += delta;
                if (res == flow) break; //所有流量都成功流出
                //这里没有直接返回，它实际上体现了一种多路增广的思想
            }
```

```
        }
    }
    if (res != flow) lev[u] = -1;
    //当前有流无法增广完，那么当前分层图上这个点永远都不能进行增广了
    //所以将距离标号设成-1以后不再会访问它
    return res;
}
int maxFlow(){
    int ans =0;
    while(Bfs())ans += Dinic(src, INF);
    //不停增广直至没有增广路
    return ans;
}
```

(2)时间复杂度分析

首先，每一轮增广过程中，BFS 建立分层图的过程时间复杂度是 $O(|E|)$ 的。然后单次 DFS 过程中，因为只能沿着层数递增的边在分层图上走，所以单次DFS 推进次数为 $O(|V|)$（当走到汇点时认为是一次 DFS, 即 DFS **树的深度为** $O(|V|)$）。

然后我们分两部分证明时间复杂度:

1. 先证明 **在一张分层图上，执行 DFS 的次数最多为** $O(|E|)$ **(即 DFS 树的叶子数为** $O(|E|)$**)**．

在分层图中，每次成功的 DFS 增广会导致至少一条关键边 (u, v)（即剩余容量最小的边）在分层图中被删除（即当前弧优化）。而增广路只能沿着层数递增的方向走，而分层图不变的情况下（也就是所有顶点的层数不变），反向边 (v, u)是沿着层数递减方向的，在该轮增广过程中必然不会被访问到。因此这条边 (u,v)就不会再出现。

因此每次成功 DFS 增广必然会在分层图上删除至少一条边，并且这条边之后不会再出现，于是一张分层图上执行成功的 DFS 次数最多为 $O(|E|)$。至于一次失败的 DFS 增广，则一定会导致某个点或某条边被删除（即当前弧优化和 -1 优化），因此 DFS 次数仍然是 $O(|E|)$。

2. 再证明 **分层图建立次数不超过** $|V|-1$ **次。**

因为 $d(s,t) \le |V|-1$，那么只要证每轮增广后残量网络的 $d(s,t)$ **严格单调递增即可。**

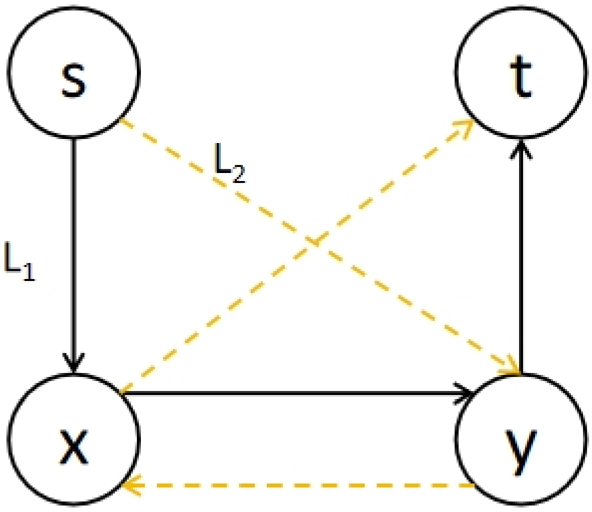
由 $Edmonds-Karp$ 算法复杂度证明已知，增广操作使得源点 s 到任意一点 u 的距离是非递减的，即有 $d'(s,u) \ge d(s,u)$，其中 $d(s,u)$，$d'(s,u)$ 分别表示相邻两次 **增广前** 残量网络 G_f，$G_{f'}$ 上 $s \to u$ 的最短路径长度。

实际上对证明过程稍加改造很容易得到一个对称的结论，即增广操作使得任意一点 u 到汇点 t 的距离也是非递减的，即有 $d'(u,t) \ge d(u,t)$。

已知 $d'(s,t) \ge d(s,t)$，下面用反证法证明不等式无法取等，即 $d'(s,t) > d(s,t)$。

假设 $d'(s,t) = d(s,t)$，那么在残量网络 $G_{f'}$, 的最短增广路 L_2 中，一定存在 G_f 中没有的边（否则这条最短增广路就会在 G_f 中一起增广），也就是 L_2 会经过 G_f 中某个增广路的一条边的反向边。

设这条边是 (x,y) 并且 G_f 的其中一条最短增广路 L_1 经过 $x \to y$，那么 L_2 就经过 $y \to x$， 如下图:



易知 $(d(s,x)+1=d(s,y), d'(x,t)\geq d(x,t), d'(s,y)\geq d(s,y)$。
L_1 的路径长度可以表示为 $d(s,t)=d(s,x)+1+d(y,t)$。
L_2 的路径长度可以表示为 $d'(s,t)=d'(s,y)+1+d'(x,t)$。
因此

$$d'(s,t)\geq d(s,y)+1+d(x,t)=d(s,x)+2+d(x,t)=d(s,t)+2$$

与假设矛盾，因此 $Dinic$ 算法中分层图的最短路径长度严格递增。

综上所述，$Dinic$ 算法的时间复杂度为 $O(|V|^2|E|)$（分层图建立次数 $\times DFS$ 树深度 $\times DFS$ 树叶子数）。

不过在实际应用中 $Dinic$ 算法的消耗时间远达不到这个上界，很多时候即便图的规模很大也没有问题（常常能用来跑 10^4 甚至 10^5 级别的图）。

特别地，在求解二分图最大匹配问题时，$Dinic$ 算法的效率能达到 $O(|E|\sqrt{|V|})$。我们发现这个网络中，所有边的容量均为 1，且除了源点和汇点外的所有点，都满足入边最多只有一条，或出边最多只有一条。我们称这样的网络为 **单位网络。**

对于单位网络，一轮增广的时间复杂度为 $O(|E|)$，因为每条边只会被考虑最多一次。

接下来我们试着求出增广轮数的上界。假设我们已经先完成了前 $\sqrt{|V|}$ 轮增广，因为汇点的层数在每次增广后均严格增加，因此所有长度不超过 $\sqrt{|V|}$的增广 路都已经在之前的增广过程中被增广。

设前 $\sqrt{|V|}$ 轮增广后，网络的流量为 f，而整个网络的最大流为 f'，设两者的差值为 $d=f'-f$。因为网络上所有边的流量均为 1，所以我们还需要找到 d条增广路才能找到网络最大流。又因为单位网络的特点，这些增广路不会在源点和汇点以外的点相交。因此这些增广路至少经过了 $d\sqrt{|V|}$ 个点（每条增广路的长度至少为$\sqrt{|V|}$ ），且不能超过 $|V|$个点。因此残量网络上最多还存在 $\sqrt{|V|}$ 条增广路。也即最多还需增广 $\sqrt{|V|}$ 轮。

综上，对于包含二分图最大匹配在内的单位网络，$Dinic$ 算法可以在 $O(|E|\sqrt{|V|})$ 的时间内求出其最大流。