# 网络最大流：

什么是网络流，给定一个有向图G=(V,E)，把图中的边看作管道，每条边上有一个权值，表示该管道的流量上限。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，问从s流到t的最大水流量是多少

最大流算法分为两大类：增广路和预流推进重标号。也有算法同时借鉴了两者的长处，如SAP 。

能进行网络流的图一般是有向无环图，一定要在图中每条边都建立反向且流量是0的虚拟边，为以后增广算法用，而且就算图是有向有环图，也不用担心，最大流算法不会通过遍历来寻找边，不会混淆虚边和实边，最适合最大流算法的图数据结构是链式前向星。

以洛谷3376为例，链式前向星可以用数组相邻元素边存储相反边方式优化，目的是能用o1时间找到相反边而不是迭代寻找。

总体来讲链式前向星最少快邻接表2倍

## Ford-Fulkerson算法：

一个很简单思路是对图dfs,找到所有可能的s到t的路径，路径权值之和就是最大流，但是请看示例，下图每条边都是100权值

按照*S12T*的路走有100流量

实际按照S1T和S2T的路径走有之和200的流量，原因是S12T中堵住了S2T的道路

改进dfs搜索的策略，有所谓Ford-Fulkerson算法就是一种基于dfs的改进方法，做法是每次找到一条路径，得到路径最小边的权值m,

然后要：路径上每条边权值减去m，然后路径的每相邻两点加反向的边，权值都等于m

直到找不到路径位置，因为是有向图，且总是把边调换方向，所以最后一定会无路可找。在将边调换方向过程中，这个被改变的图就叫做参与网络每次寻找新流量并构造新残余网络的过程，就叫做寻找流量的“增广路径”，也叫 增广

## Edmonds-Karp算法

之前说的dfs的方法遇到以下情况会很慢，

路径权值1到2是1，其余路径都是100

每次增广如果都经过1到2的路径，会进行200次dfs

改进方法是增广时尽可能选择边数比较少的路，权值大小无所谓，来增广，怎么每次边数最少，当然是bfs的方法。

这是Edmonds-Karp算法，时间复杂度 V\*E\*E，代码难度小很好写

### 编码细节：

* 重边可以不考虑，因为本身添加反向边就可以导致重边，这个算法会处理，但最好还是合并成一条边
* 自环不需要考虑，孤立点孤立的块(非连通图)的情况一般题目不会出现，即便出现也无效，也不用考虑
* 用vector可以解决一般问题，但不够快，可以考虑自实现链表或者用list做邻接表，差了10倍左右效率
* 可以单纯存一个Edge p[]数组，p[i].b代表i的父节点编号，p[i].e代表i是它的父节点p[i].b在邻接表的位置，p[i].v代表i父节点到i本身的权值

## Dinin算法：

性价比最高的算法，时间复杂度上一般比赛足够用，编码也不复杂

思路是先用E 的时间复杂度bfs(不一定遍历整个图到达终点即可停止)，同时对点进行分层，layer[i]代表点i的层数最,之后进行只能走不同层的dfs(这样保证了优先走边数最少的路),dfs走到终点或者和终点同层的点就进行

这个算法要多次进行bfs和dfs尽管有时一次dfs可以求出多个路径

总复杂度仍然快于EK算法。

时间复杂度是E\*V^2比EK算法快,这个算法时间复杂度怎么算，而V个点的层次图最多有V层，增广路最多E条，前进和回溯都最多V次

### 编码细节：

* 重边自环EK算法里说过这个问题
* 尽量用邻接矩阵和来链式前向星做网络流的题，邻接表在这里效率不佳，因为邻接表无法实现直接通过两点访问某条边，而邻接矩阵可以直接下标访问,链式前向星

## SAP算法：

最大流较快的算法，效率和Dinic相同级别，因为它相当于把Dinic的操作合并了一下，但快dinic几个常数，但是编码复杂，容错率低(细节多)，代码量比dinic多

沿用了Dinic的分层思想，但是由于Dinic进行了多次bfs而影响相率，SAP只进行一次全图逆向bfs的分层，逆向是指从终点到起点的bfs,走的时候走逆向边，通过更新layer[i]数组，来实现找到所有的流。 维护一个gap[]

gap[i]表示layer=i的点的个数，如果某个gap[i]是0说明不存在layer=i的点，则说明终点不可达，也就说明了找完了所有的流

之后进行dfs,这是一种特殊的dfs,具有以下性质，可用循环实现

* 进行广搜时勿忘更新gap数组，而且核心代码里一定要走经过layer分层的路径，千万不要走到layer[i]==-1的点
* 找到终点后进行增广后直接回溯到原点
* 如果可能,就按照dinic的走法,走满足layer[now]= layer[next]+1的路径,这里dinic的layer[next]= layer[now]+1相似
* 当找不到上述的路径，且没到达终点时，找当前点能到达的不满足上述条件的路径中layer[next]最小的点的minlayer，更新layer[now]= minlayer+1。退回上个节点。
* 满足两种条件之一说明已经扎到所有最大流：有一个gap数组的某个位置变成0；layer[S]>=V也就是起点的层数被更新得大于等于总点数。两种情况其本质是分成出现断层，那么也就不可能来找到新的最大流路径了
* 当前弧优化也适用于sap
* Dinic和sap都要会，这二者性能是一个数量级别，总体sap性能好，但二者都会被特被卡掉，不存在完爆，在稀疏图中sap快，稠密图dinic好点。

## HLPP算法：

最快的网络流算法，时间复杂度V\*V \*sqrt(E),对于边数较多的图，该算法较为明显的提速，在稀疏图里,效率不及加了当前弧优化的dinic和sap。

还是沿用Dinic的思路对图进行分层，步骤如下

1. 通过**bfs**预处理出距离标号h，即到达汇点t的**最短距离**;
2. 将从源点s出发的边设为**满流**，到达的点**v**标记为**活动节点**并加入到优先队列中;
3. 从优先队列中不断取出点**u**进行**推流**操作，要求到达点v的**必须满足h(u)==h(v)+1**;
4. 若推流后u中仍有余流，则进行**重标号**。将**h(u)设为min{h(v)}**，**再次**加入优先队列并重复步骤3;
5. 当优先队列为**空**时，结束算法。

# 最小费用最大流：

是上述最大流模型的扩展问题，在最大流模型上，给每条边增加一个费用，指走过这条边单位流量的费用，要求你求出最大流同时使得总费用最少。

同时要知道这么个事，可以求最大流的图一定内部没有不会包含终点的环(不存在 不包含终点的强联通子图)，因为如果存在，那么沿着这个环走，流量是无限大

这种问题思路就是求最短路，在最短路的同时兼顾求最大流，求最大流实际是求所有原点到终点的路径权值之和，可能有不同组合，因此求最大流下最短路，等同于求多次最短路并且增广，直到找不到最短路为止，多次最短路的权值之和是最大流

## SPFA和最大流结合:

用最短路径的SPFA算法，求最短路径，并且用一个p数组纪录路径，类似EK的广搜那样，执行一次SPFA后，把找到的路径进行增广操作

时间复杂度V\*E\*E\*k k是SPFA的大常数，总之这个算法比较慢

### 编码细节：

* 可以提前就把每条边的反边作好,
* 重边要合并成一条
* 这里的SPFA不需要判断负环的数组，因为最大流网络求最短路只必然没有负环，因为负边是认为加的，本身网络流图性质决定的

## dijstra和最大流结合:

spfa效率边数较多比迪杰斯特拉慢，迪杰斯特拉是稳定ElogV，但是图里面有由增广建立的负数cost的边，所以不能直接用dijstra,应该把权值变成正数，做法是我们如果给每一个节点 加一个属性h[i]：

一开始这个图是没有负边的，负边是加进去的，那么每次求完一次最短路就利用dist[]数组更新h,如何更新h,根据最短路性质，对于正数费用的边的费用满足cost[v,u]<=dist[v]- dist[u],负数的费用是它的相反数，现在试想如果每条边不论正负都加上一个它自身的绝对值，就能保证不是负数了，这个加上绝对值如何体现，就是用那个h,令h[i]=dist[i]相当于把cost[v,u]+ |cost[v,u]|。此外，在获得边上权值时，比如dijstra算法过程中 边距离 不再是每条边原本的cost，而是cost+h[v]-h[u];

还要知道一件事是：对于某条边 e=(u,v)，cost[u,v]= cost0[u,v]+h[u]-h[v];把所有边的花费由cost0变成cost后，最短路变成:

minLen=minLen0+h[s]+h[T], 算总费用要减去(h[s]+h[T])

好了其余就和最大流结合SPFA一样了

迪杰斯特拉算法的堆在stl中效率较低，但是开o2优化才能到达正常水平，经测试总效率和spfa差不多，没有网上某博客吹得那么快

算法总时间是V\*E\*E\*logV。也比较慢

# 下界网络流：

顾名思义，每条边流量规定了下界的网络流问题，通常思路是把边的上下界分离，新建额外点X和Y，

# 网络流的性质：

## 最大流最小割定理：

是指在一个网络流中，能够从源点到达汇点的最大流量等于如果从网络中移除就能够导致网络流中断的边的集合的最小容量和。即在任何网络中，最大流的值等于最小割的容量。怎么求最小割边集，和最大流增广路没有什么直接关系。这里不要以为如果跑完最大流后某条边是满流，说明它在最小割的边集里，因为会由没流完但是从原点汇点都访问不到的边也应该算进去。

实际应该利用的残余网络，从原点和汇点进行dfs，分别标记访问到的点染色0 1 2，之后遍历所有边，如果两个点不都在1或者不都在2就说明是最小割的边