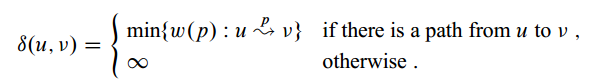
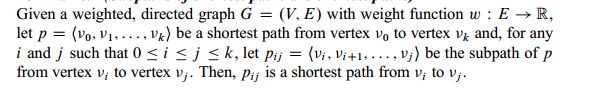
Shortest-path weight

d[v]代表v点到s嘴短距离

从u到v最短weight和

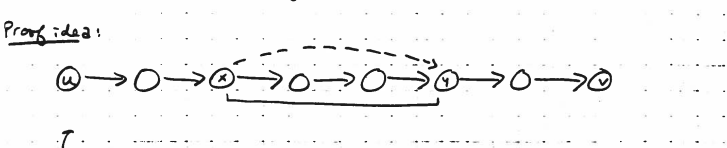


Optimal substructure of shortest path



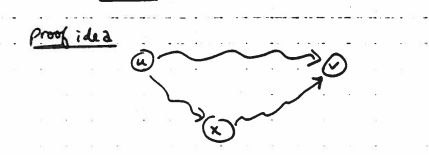
假设有一个最短路径，从v0到vk

那么对于他的子路径vi to vj，必然是vi到vj这两个点之间的最短路径（如果vi到vj之间有多条路径的话）



Triangle Inequality





三角形定理，uv最小值必然小于等于ux+xv

注意这里是有等于的，也就是ux,xv就是最短的情况

如果是大于，那么uv就会被后面那条路径替代

well defined

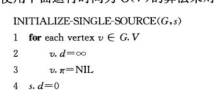
虽然negative weight edge是可以的，但是我们不允许negative length 的cycle

不然可以无限循环无限减小路径长度

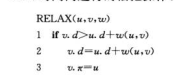
Relaxation 松弛操作

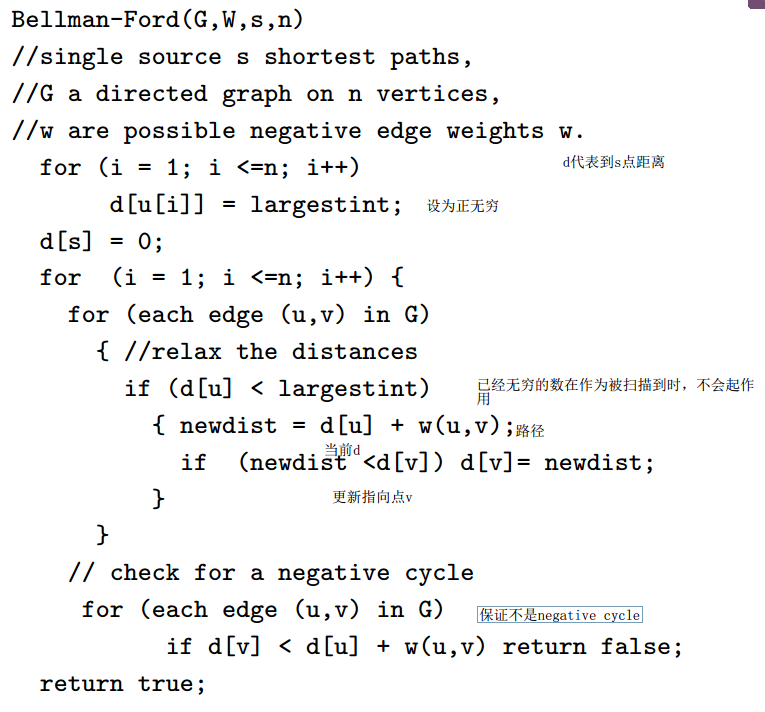
对于每一个vertex,都给他一个attribute v.d,代表着起始点s到v的当前最短距离（后续可能会被更新），我们把v.d叫做shortest-path estimate

v.pi 是v的前驱节点，我们的版本不管



一开始初始点的d设为0，其他点的d设为无穷

然后就松弛，如果这个点加一个edge到达的新path length小于到达的点当前记录的path length,更新



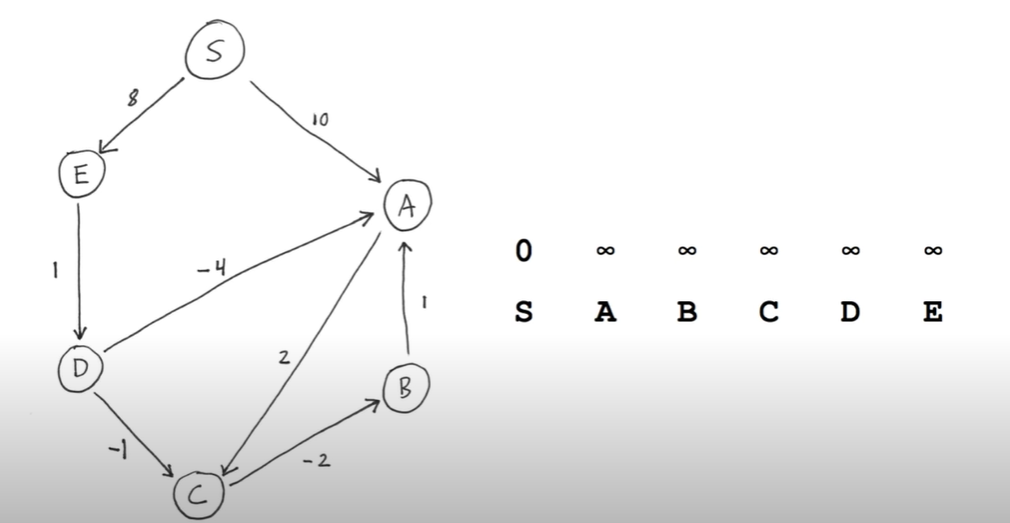
Bellman-Ford algorithm

算从一个点到其余点之间最短距离的算法

好处在于，允许edge是-weight

坏处在于用是比较久

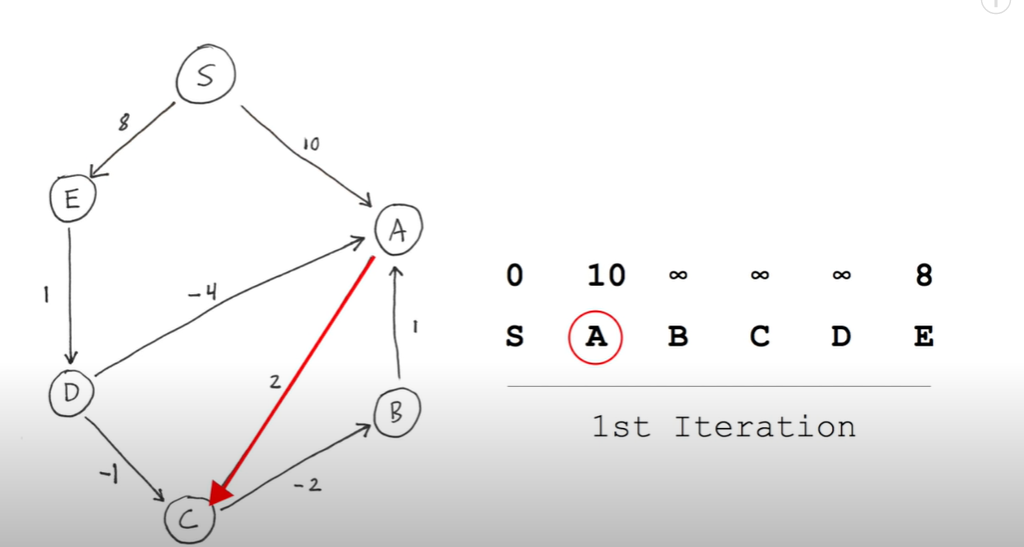
循环次数等于vertices数量减一



一开始除了起始点1，每个点设为无穷

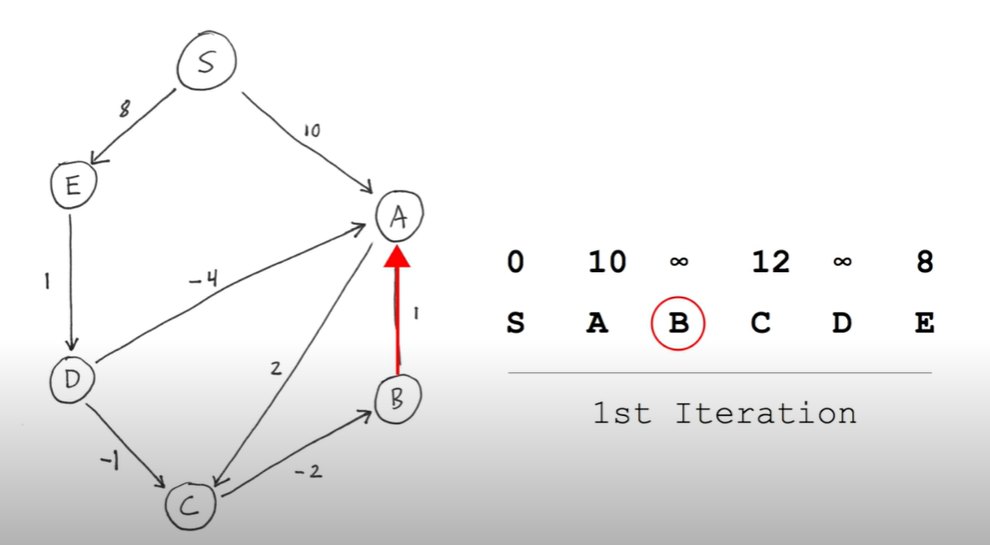
从左到右一个一个循环，更新这个点所指向的点的数据

S:指向EA,



EA被更新，A，指向C

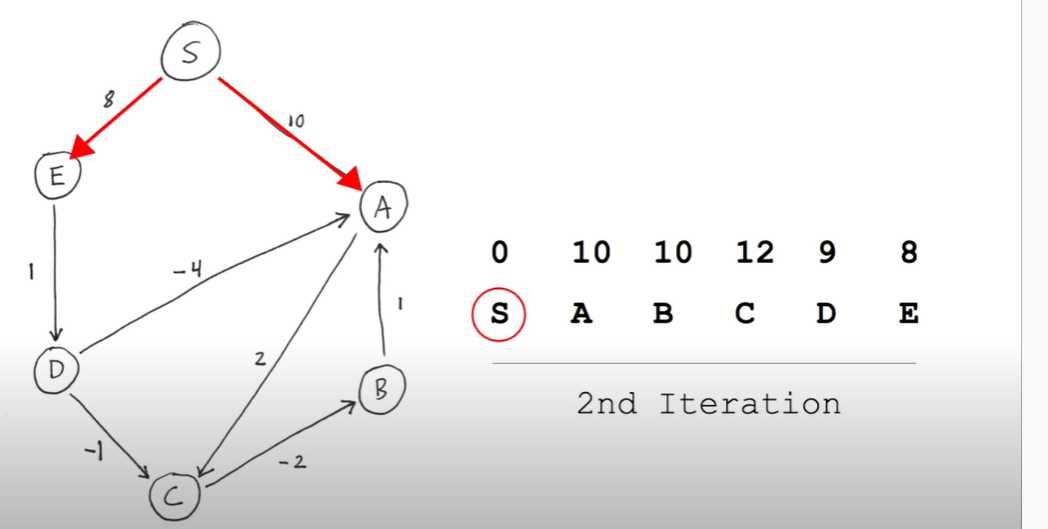
C被更新



B当前是无穷，不用管

C指向B,B更新成10

D不用管，E指向D，D改成9



重新进行第二次循环

S:EA不用改，

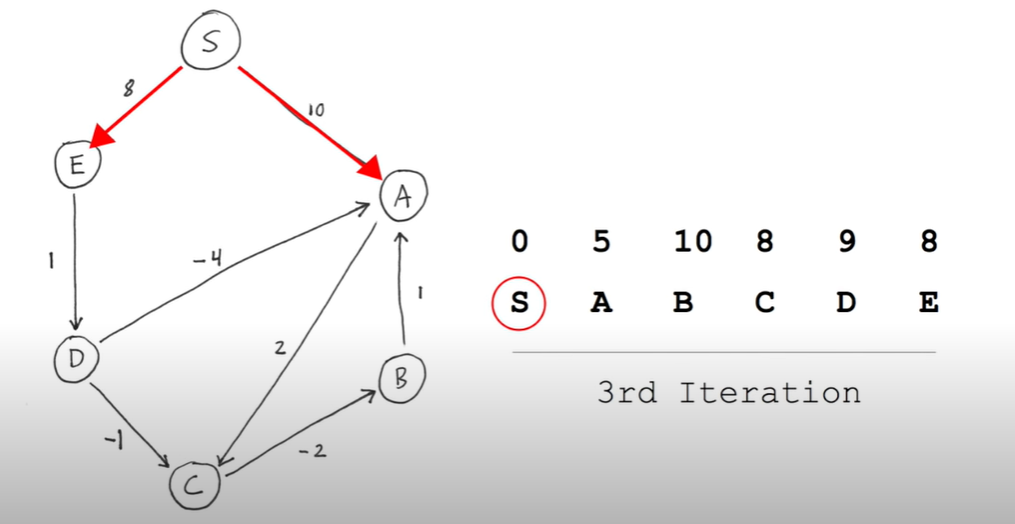
A:C不用改

B:A不用改

C:B不用改

D:C改成8，A改成5,

E:不用改



一直循环，直到第VERTICES-1 Iteration，这里有六个点就是5次循环

或者第n次循环与第n-1次没区别，这里第四次与第三次没区别，停止

为什么BF有效：

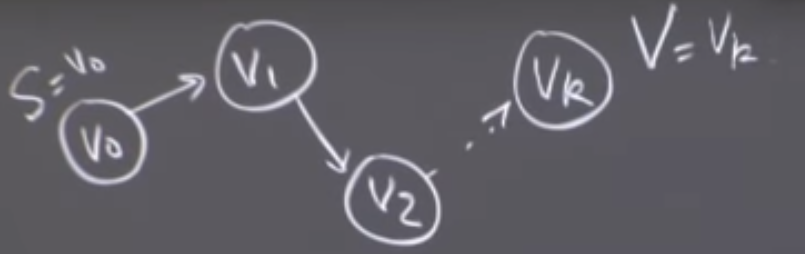
换句话说，证明

在v-1次循环后，任意点到s的距离都是最短路径

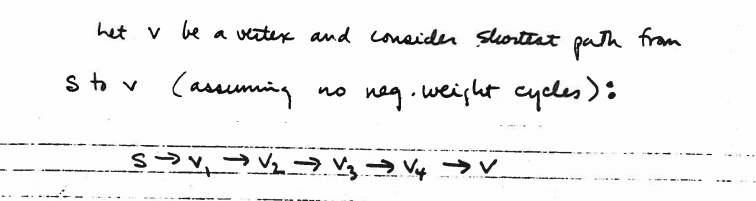
关键点：” Optimal substructure of shortest path

那么对于他的子路径vi to vj，必然是vi到vj这两个点之间的最短路径（如果vi到vj之间有多条路径的话）

首先，s=v0,假设v0到vk是到k的最短路径,一共有V个点,且没有negative weight cycle



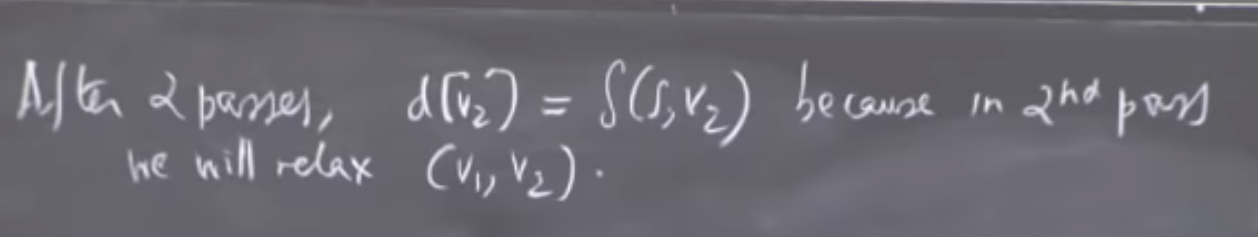
那么k必然小于等于v-1,不然会有Cycle



首先d[s]=0是正确的

在第一次遍历pass后，因为 v0到vk是我们假设的shortest path，那么根据optimal substructure of shortest path,v0到v1也是shortest path,因此d[v1]是正确的，d[v1]=d[s]+w[s,v1]

第二次遍历d[v2]建立在d[v1]上



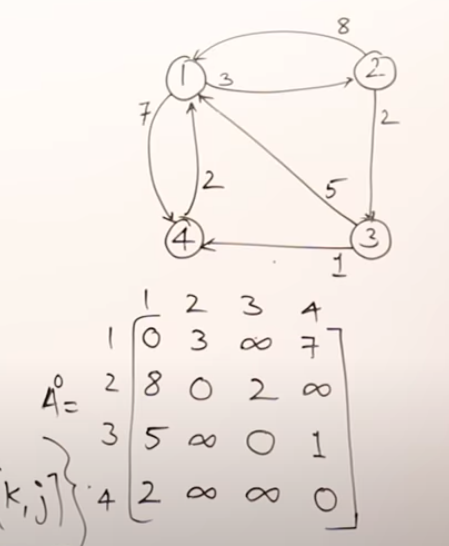
relax起点取决于你第几遍pass，我们要通过relax才能定位出v0到v1,v1到v2之间的最短路径

可不可能存在v0到v2小于v0到v1+v1到v2呢

答案是不可能的，这样v1就不在最短路径里了，不符合假设

All pairs shortest paths:Floyd-Warshall

找到所有点之间最短路径

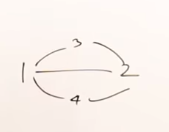


列一个matrix

点到自己距离是0

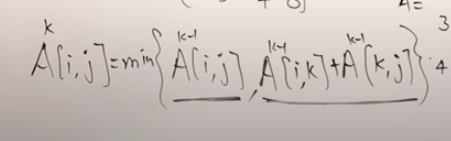
横轴是起始点，竖轴是终点，如果起始点到终点之间一开始没有直观的线，取正无穷

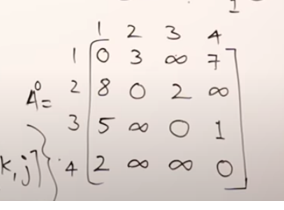
All pairs shortest path目的就是把所有值都压到最小（点对之间最短距离一下求出来）

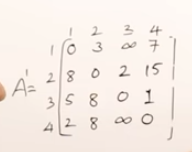
首先我们想知道1到2的最短距离，有可能

情况1. 直接1到2，压缩在1的遍历中

情况2. 3或4为中点，到了1到2，压缩在3或4的遍历中

公式：

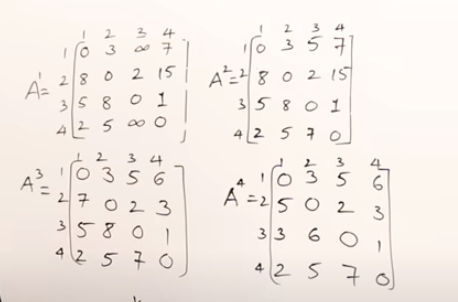
A0：



第一遍关于点1的遍历：斜着的点，仍然保持零

1的竖轴横轴不变，对应情况1，它代表别人直接到他或他到别人的长度

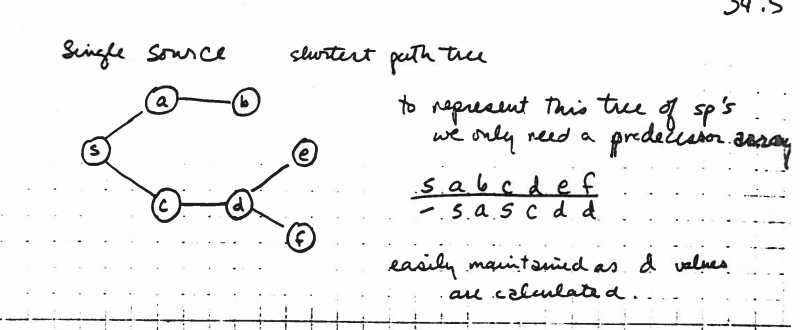
其他的点要带入公式，他们代表情况2，例如变动的24， 它实际上在让1作为中心点比较A0的21+14,以及原24大小，8+7=15



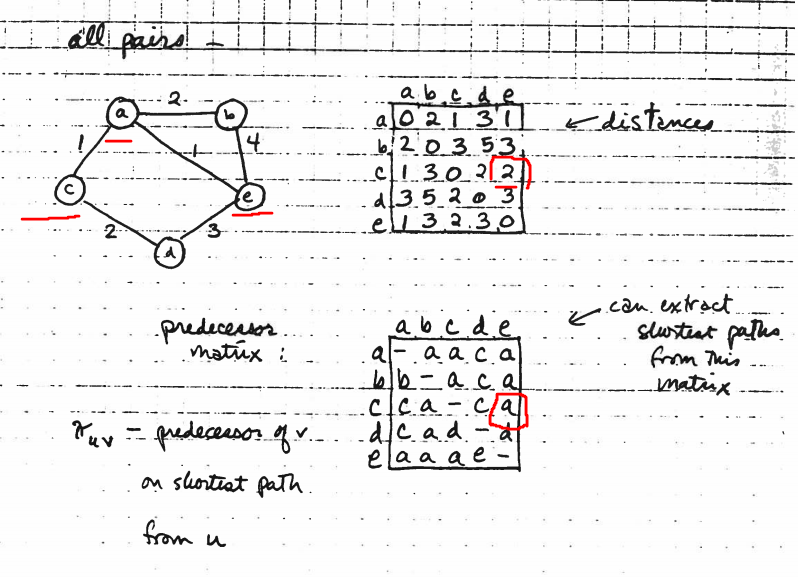
怎么记录最短路径

创造新的表格，随着最短路径值的更新而更新

Single source

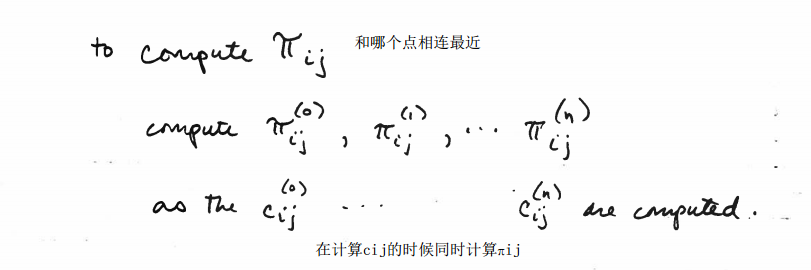


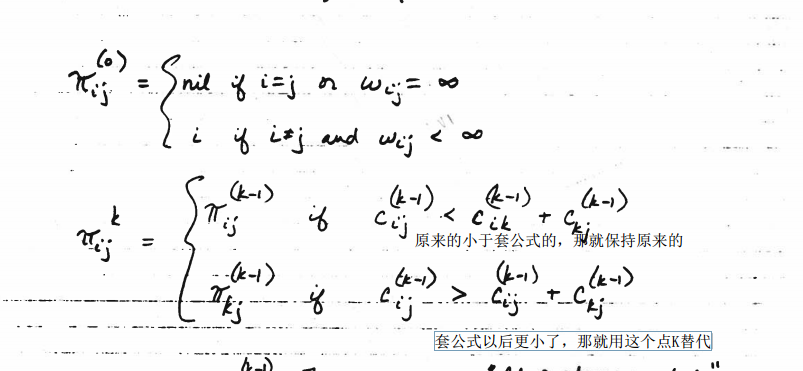
这个简单，如果扫描到哪个点导致与他相邻的点更新了，就说明相连了



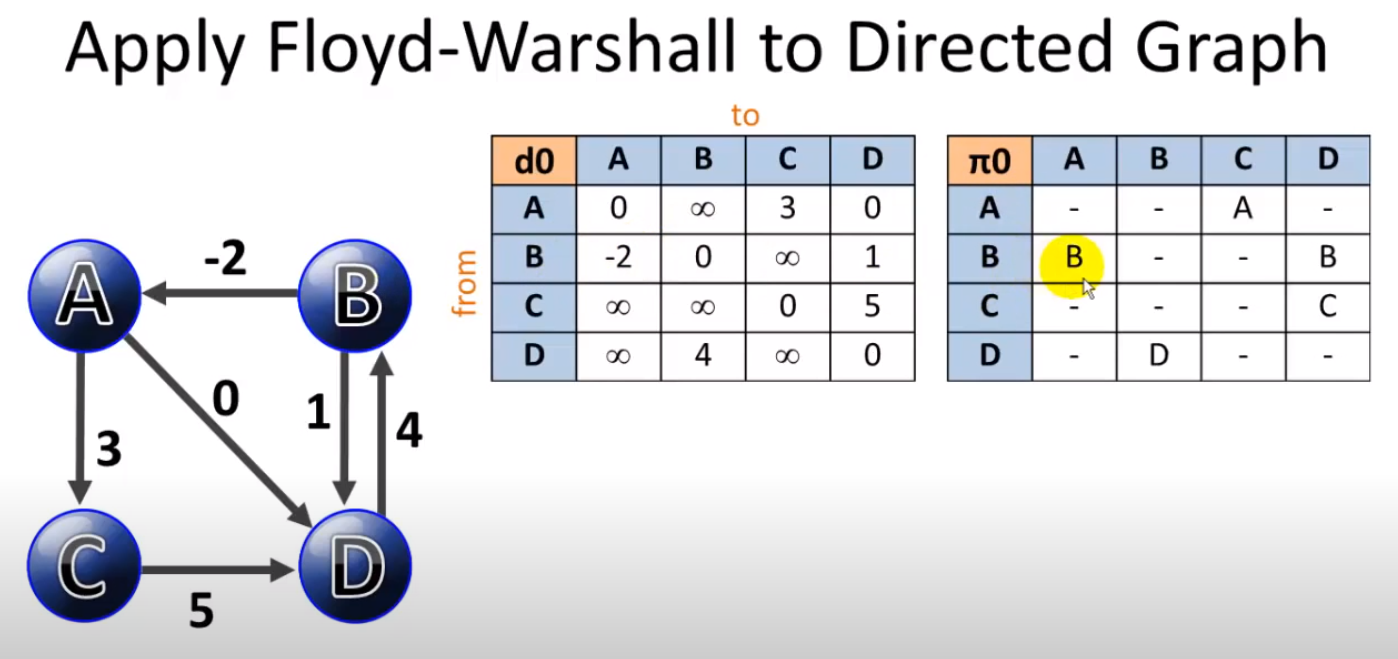
非single source的情况

如果中间通过公式改变了最小值，那就把中间点替换掉直接点





叫做predecessor matrix



左边是from,上边是To，在π表，填的是FROM



BC通过点A变小了，就把A记录上去，这个点叫Predecessor vertex

.

Max flow

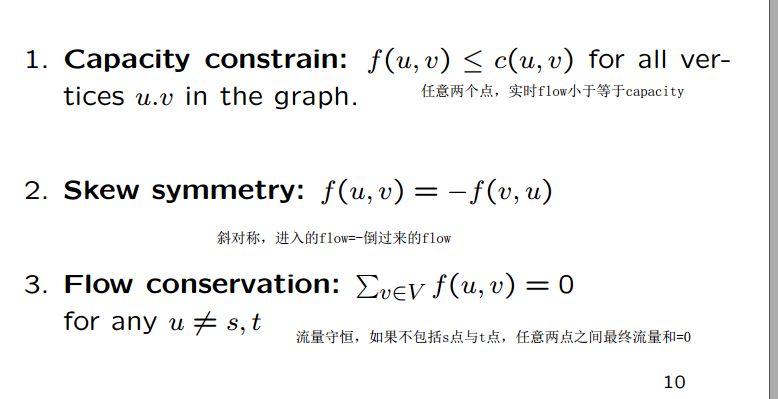
定义一个directed graph,每个edge都有一个capacity(不是weight，而是承载量，例如一根水管，capacity是5，实际上承载4也没问题)

代表着点u到点v的capacity

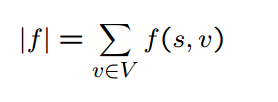
Max flow问题

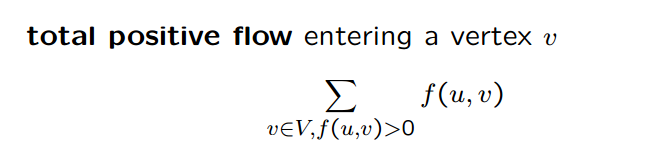
定义一个source点，一个sink点从Source到sink所能承载的最大flow

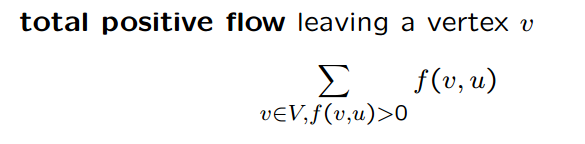
我们假设从source到sink，会经过graph上的每个点（那些不经过的点，我们可以去掉他们而不影响结果）



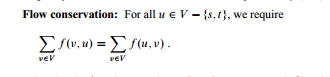
Flow conservation是想告诉你，除去开始与结束两点，中间的capacity是不应该带有flow的，可能部分path当时有flow，但整个网络的和是0

一个flow的值只取绝于这个点到Start点的值

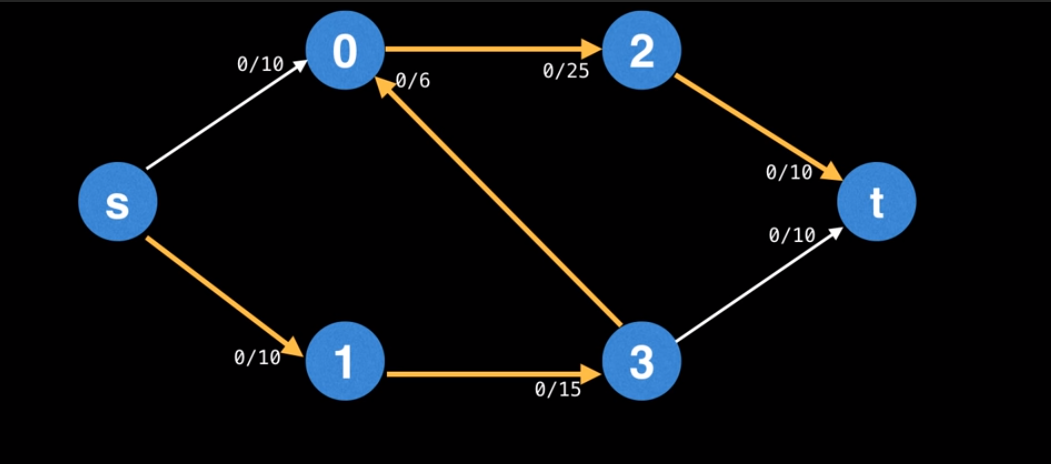




对于任意除了source and sink的点来说，flow in =flow out

也就是书上这一点想表达的.(P709)

Augmenting path:

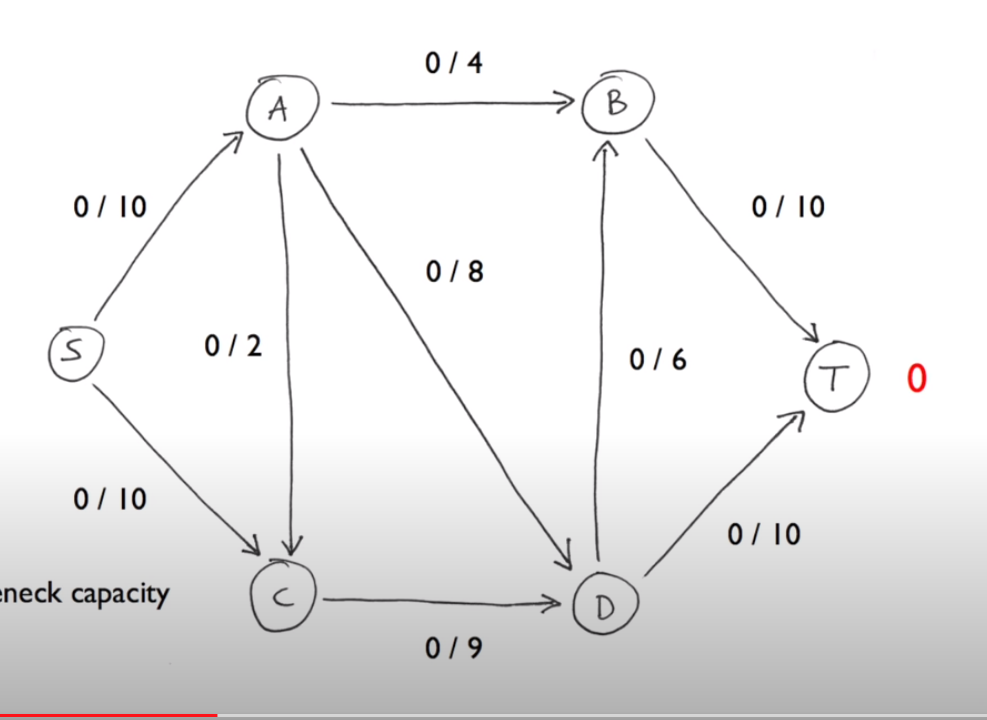


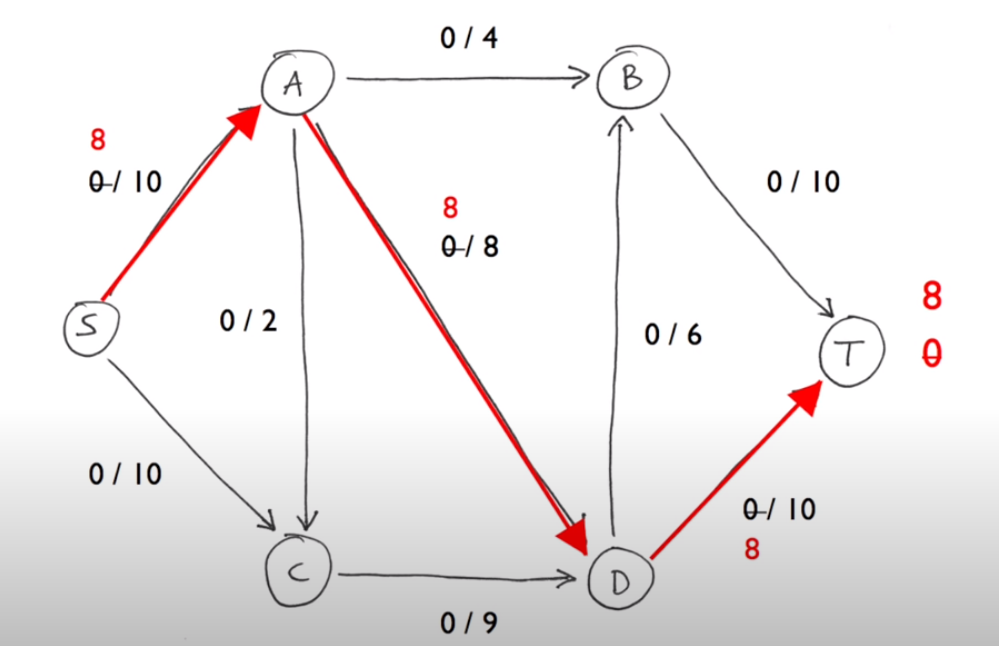
从s到t，如果一条path仍然存在unused capacity，那么就是augumenting path

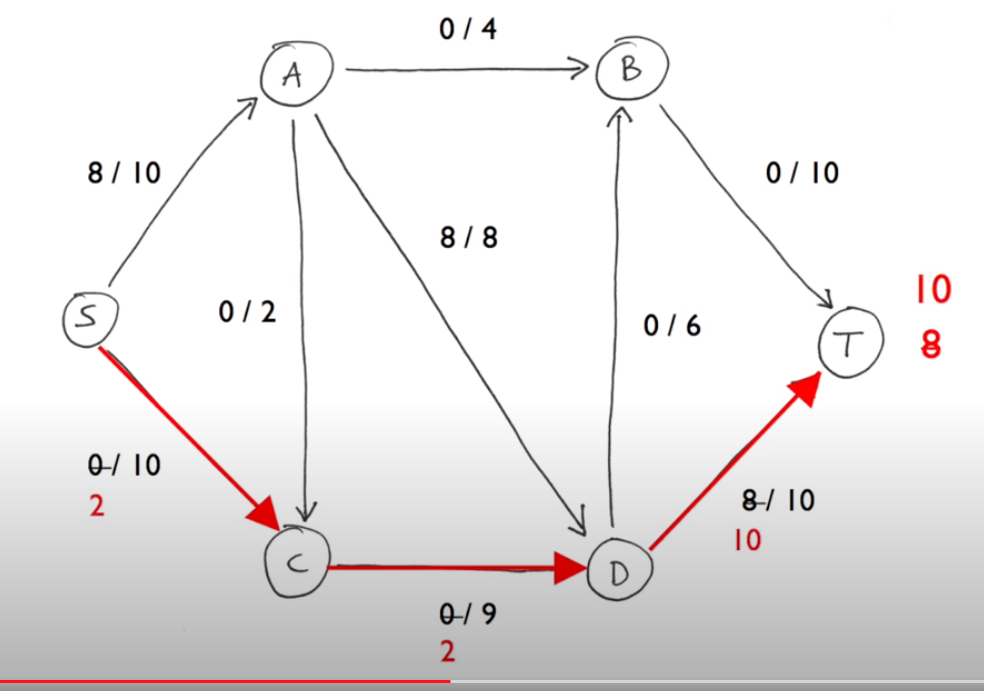
当没有augumenting path时，达到max flow

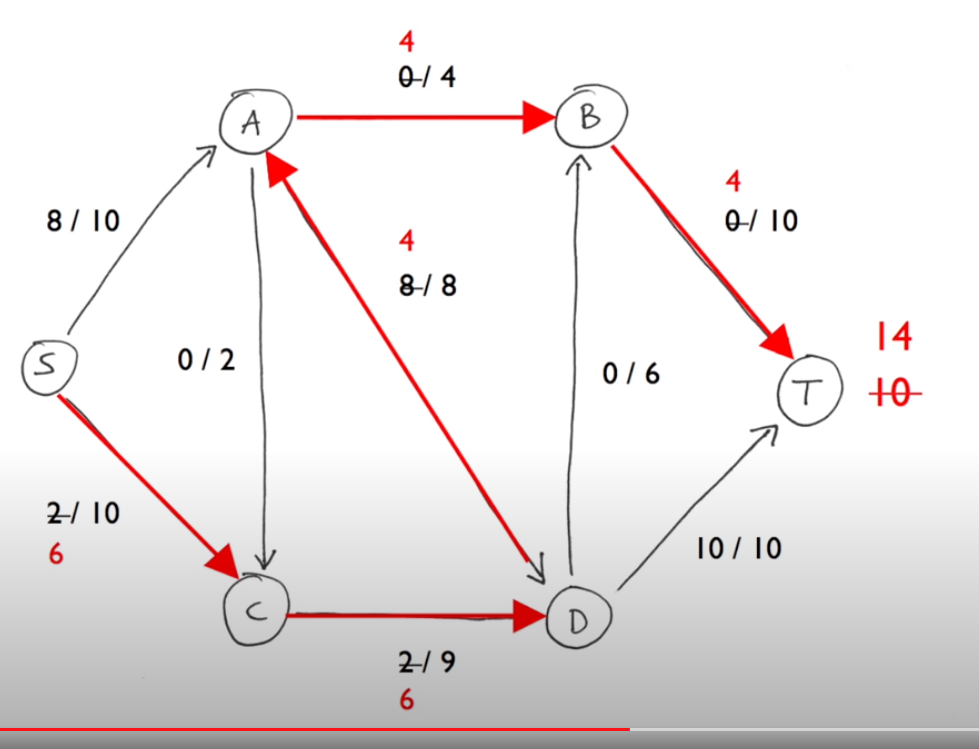
Ford-fulkerson方法

就是不停的找augmenting path，找到以后减去这个path所能所能和承受的capacity,直到再也找不到augmenting path//具体顺序没啥要求

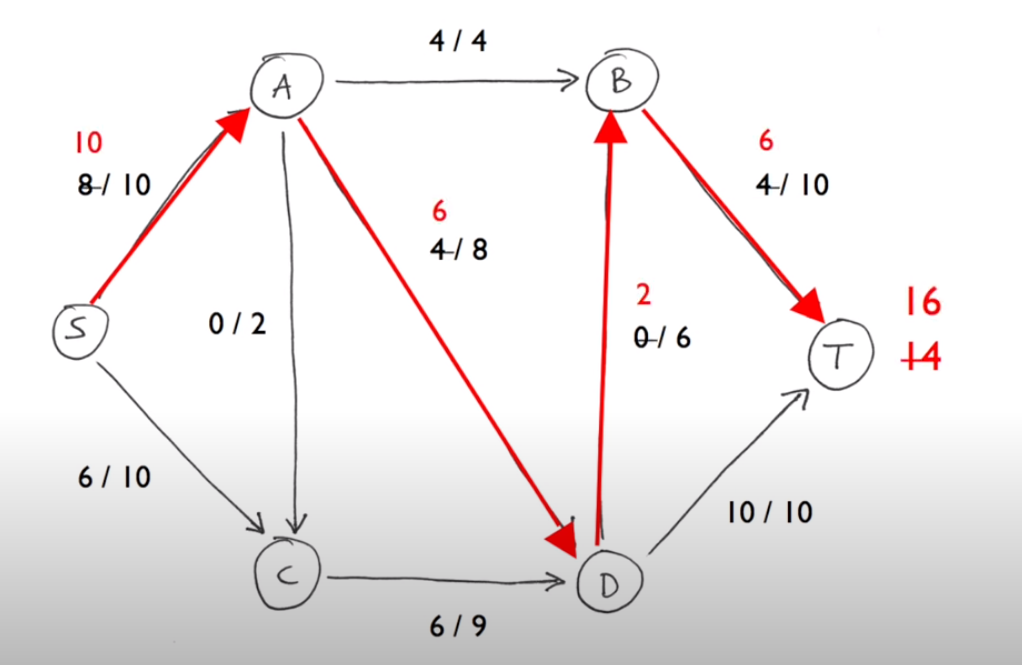


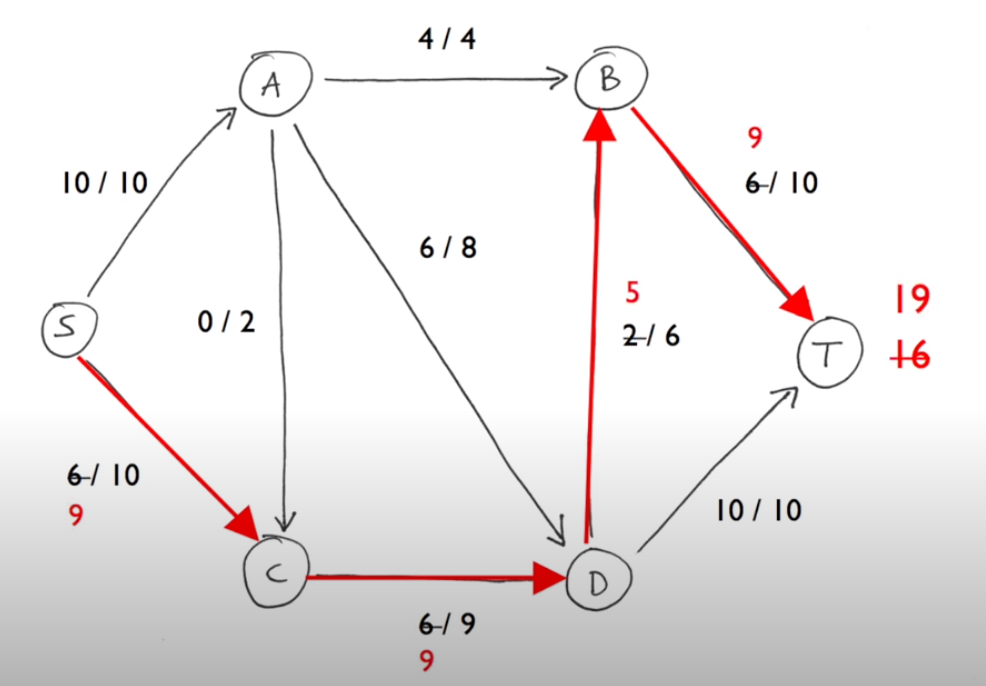






注意这一步，实际上D到A是没有edge的，但是在求max flow时，可以让我们的路逆着走，逆着走的时候减去当前flow就好了，它代表着原来A指向8to D, 0 to B,现在被拆分成了4 to D, 4 to B,也就是一个把指向点拆分实际输出的过程，到D的输出被拆分成了A到B，A到D

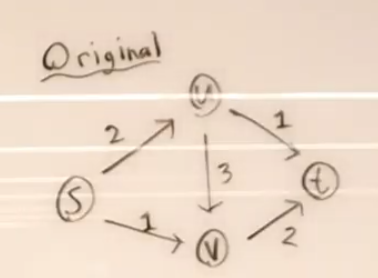




到极限了

但这样并不能保证我们找到最优解

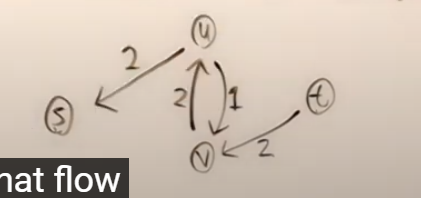
引入新概念，residual graph

Original，一开始只是capacity，整个系统是空的，里面并没有水

而residual graph，可以精确的告诉我们什么时候undo flow(把已经分配好的flow撤销)

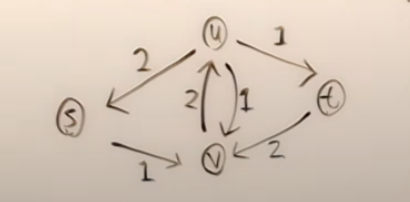
用greedy algorithm

我们第一条path自然是232 //greedy



因为push进了2 unit，所以如果想undo,我们自然能undo 2 unit(相当于word写入以后，写多少撤销多少，)3 那里可以undo2,但还剩1capacity

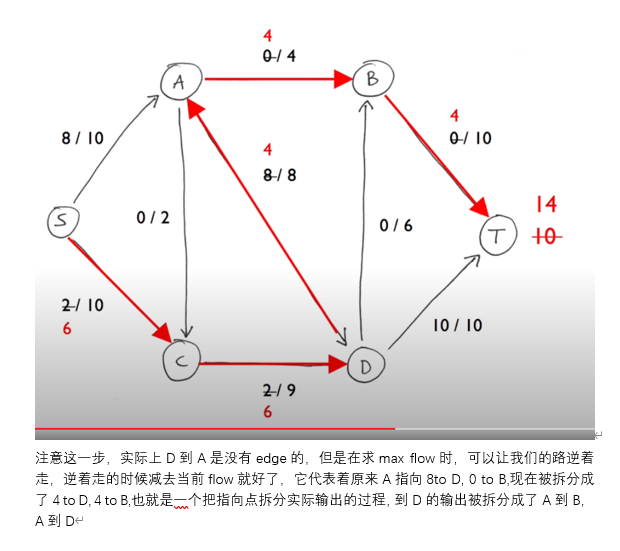
Undo并不代表实际flow,只是代表如果你想撤销的话，我这里能提供多少撤销余额



现在我们还剩两条1 .//实际方向，

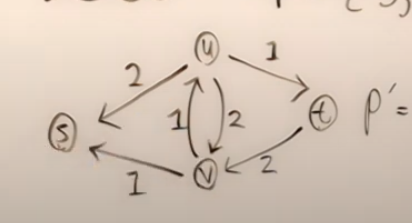
这时我们就可以利用到undo了，虽s->v->u->t

虽然我们知道v到u没有箭头



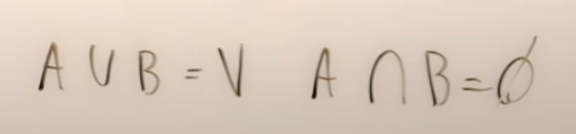
但是和这一步逻辑一样，因为我们undo 余额是2，所以我们可以undo 1 unit u->v flow

实际上是完成了一次u到v的再分配，我们觉得直接指2不好，撤销，两边各指1



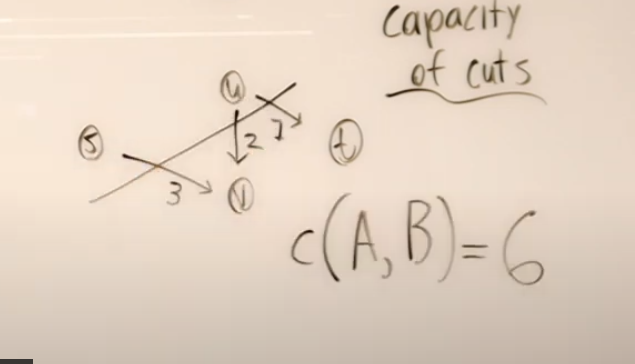
path成立，写undo，

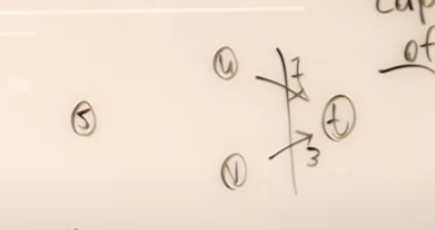
Cut:用一条直线切割,start在一边t在另一边，把vertices切割成两部分，切割线经过的



cut会切割经过的edge,

Min-Cut:切割的所有edge的capacity和最小的cut，

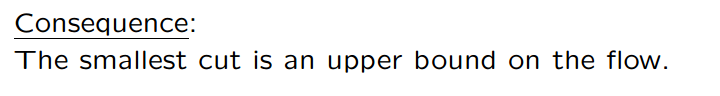


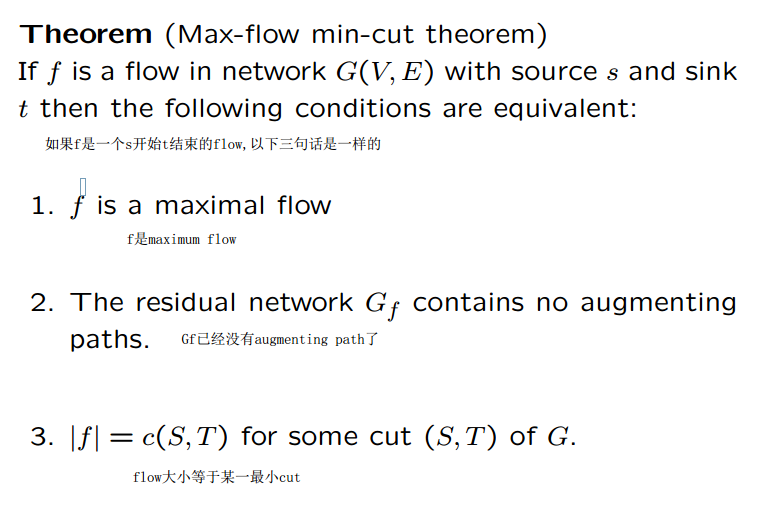


这里的mincut就是4

Mincut的意义：MinCut必然等于max flow

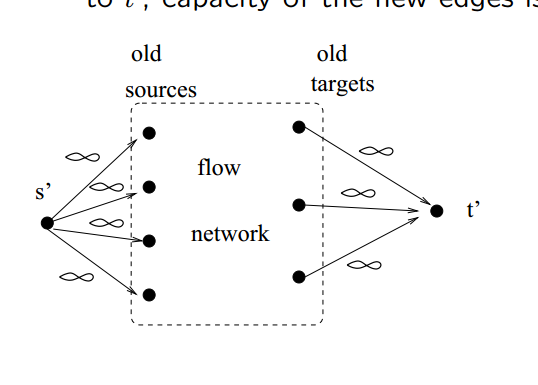
我们可以用mincut哪怕是不精确的mincut，尽量估计max flow





Multiple sources and sinks

这类问题可以被转化成单一source单一sink的问题



创造一个虚拟source 与target，连接原来的source,taerget,capacity是无穷，因为augmenting path主要是最小capacity的瓶颈问题，因此不会影响

**Edmonds-Karp Algorithm**

前面一种瞎找augmenting path的方法是有随机性的，他的表现取决于找augmenting path的顺序

使用Edmonds-karp algorithm，他的表现是polynomial time//稳定的多项式时间

使用BFS找到所用edge最少的path，

然后对这个path进行常规操作

循环

也就是说我们使用的augmenting path的edge是单调递增的

https://www.youtube.com/watch?v=w3Nl2XA0pxA

Maximum Bipartite matching

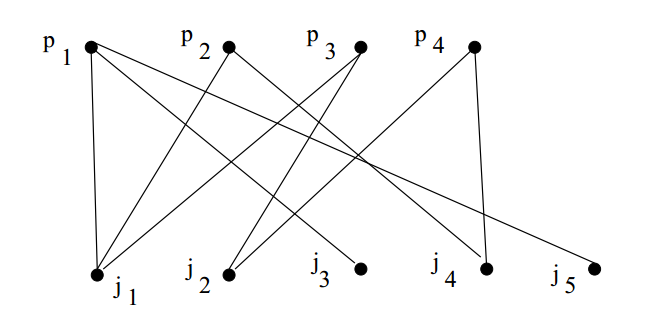
我们有n个人，m个job

每个人可以做指定的工作

现在每人最多做一个工作

最多能完成多少工作

这个问题能被用bipartite graph表示 //二分图



这类问题可以转化成一个maximal flow问题.

引入一个source,一个sink，每个capacity=1

