1.Minimum Spanning Tree (MST) 最小生成树

Let *G* be an undirected graph.

We are also often interested in finding a spanning tree *T*, i.e. an acyclic spanning subgraph of *G*, which *minimizes* the sum of the weights of the edges of *T*.

Computing a spanning tree *T* with smallest total weight is the problem of constructing a *Minimum Spanning Tree* or MST for short.

即在无向图中有n个点，用n−1条边来产生一棵树，使得这n−1条边的权值的和最小。

2.计算最小生成树的算法

2.1普里姆算法 Prim‘s algorithm

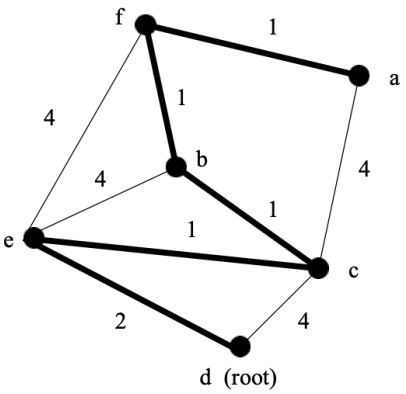
思路是从根结点开始通过贪心算法构建MST，在每一步中通过对已构建对树添加边的方法，确保不会产生权环的构建方法。

在伪代码中有两个顶点的集合，U表示已经访问过的顶点，V−U表示没有被访问过的顶点。通过连接最小权值边的方法来将顶点从V−U移动到U。

伪代码如下：

```
T = ∅;
U = { 1 };
while (U ≠ V)
    let (u, v) be the lowest cost edge such that u ∈ U and v ∈ V - U;
    T = T ∪ {(u, v)};
    U = U ∪ {v}
```

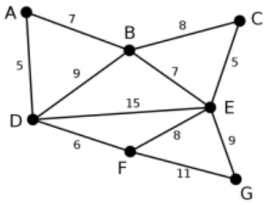
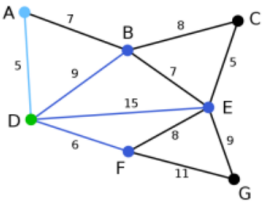
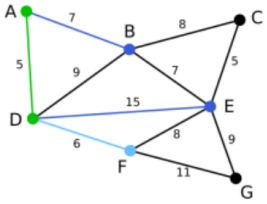
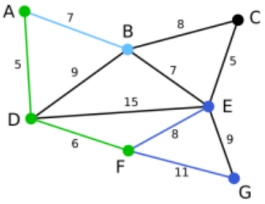
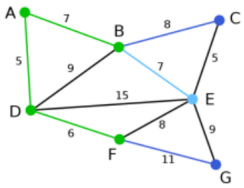
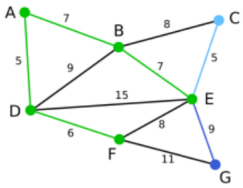
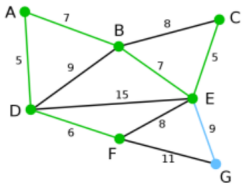
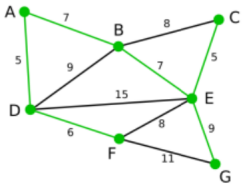
例子：



上图中以d为根结点，此时只有d被访问，此时这颗树只有d一个节点，距离这棵树最近的节点是e，将e加入次节点，得到新的树，然后寻找离新的树最近的节点，继续加入该树形成新的树，持续该操作。

注意在该算法中，每步计算的不是距离新加入的节点最近的节点，而是距离树最近的节点。

再有一个例子，以D为根结点：

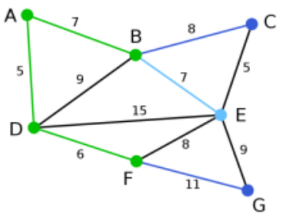
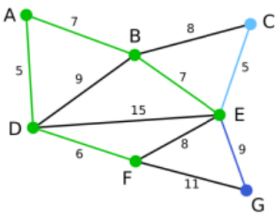
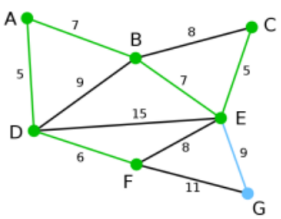
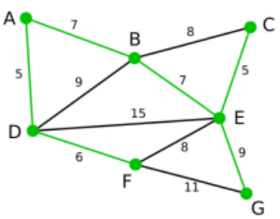
2.2克鲁斯卡尔算法 Kruskal’s Algorithm

该算法也是计算MST的一种贪心算法，和普里姆算法不同，此算法并不是从某个节点开始构建树，而是按从小到大的顺序来添加边，每一步都选取权值最小的边，但要注意在添加边时不要和已经选取的边形成权环。

克鲁斯卡尔算法具体步骤如下：

1. 将所有的边按权值从小到大的顺序排列起来。
2. 按权值从小到大的顺序将边添加都生成树中，注意不要形成权环。
3. 重复上一步骤，直到虽有的节点都加入进去。

例子：

将所有的边按权值顺序排列起来：

AD：5

CE：5

DF：6

AB：7

BE：7

BC：8

EF：8

BD：9

EG：9

FG：11

DE：15

按顺序依次添加AD，CE，DF，AB，BE

BC，EF，BD会形成环故舍弃，继续添加EG，形成权环的FG，DE舍弃，得到的结果如上图。

2.3Borůvka's algorithm

同样是贪心算法，步骤如下：

1. 输入一个连接的带权值的无向图。
2. 把所有的顶点初始化为单一的变量（component）。
3. 将MST初始化为空。
4. 当有不止一个变量时，对每一个变量进行下面的操作：找到连接此变量和其他任何变量最近的权值边，如果没有添加就将这个最近的边加进MST。
5. 返回获得的MST。

可以看作是多源化的普里姆算法，即每次都在一棵树上添加另一颗与这颗树距离最近的树，初始情况下将每个节点看作是一棵树。

例子：

下面的图中，为方便计算将节点从左到右从上到下命名为abcdefgh。

第一步，将八个节点看作八颗树。

第二步，寻找距离每棵树距离最近的子树：

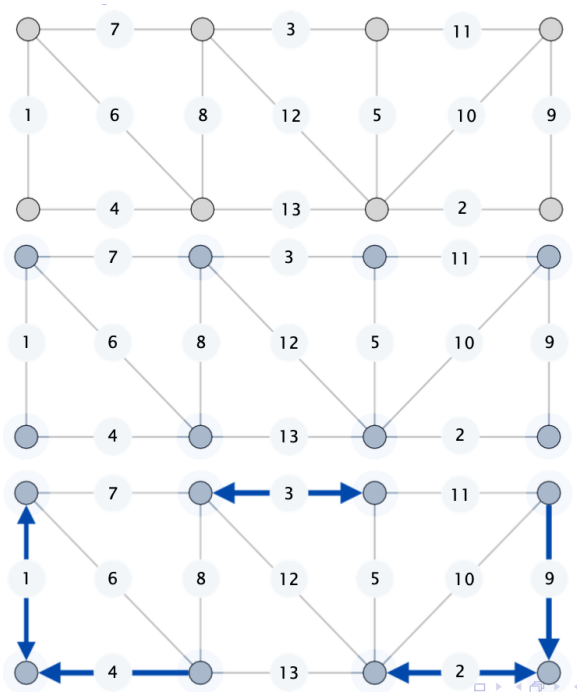
距离a最近的是e，距离b最近的是c，距离c最近的是b，距离的d最近的是h，距离e最近的是a，距离f最近的是a，距离g最近的是h，距离h最近的是g。

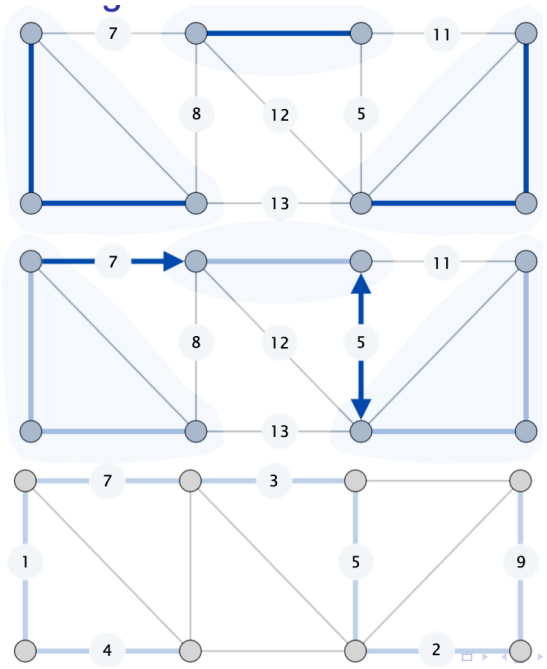
第三步，e和af相连，b和c相连，h和dg相连，这样就重新形成了三棵树。

第四步，将上述三棵树分别命名为1，2，3，距离1树最近的树是2树距离为7，距离2树距离最近的树是3树距离为5，距离3树最近的是2树距离为5。

最后一步，将三棵树按最短距离相连，得到MST。

具体图示如下：





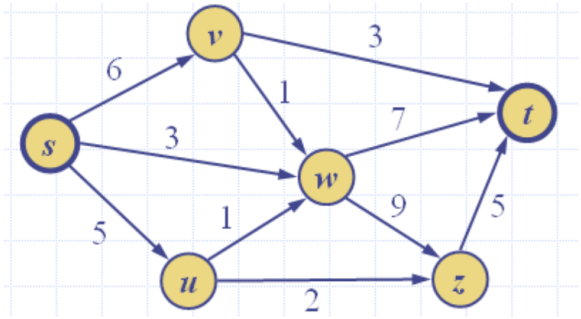
3.Flow network 网络流

3.1网络流和流

定义：网络流（或者网络 network）

一个加权图（非负整数权值边），每条边的权值被称作边的容量（capacity）。这个图中存在两个不同的顶点s和t，分别被称作源顶点（source）和汇聚顶点（sink），使得s没有汇入s的边，t没有流出的边。

例子：



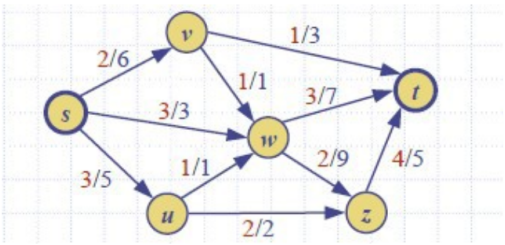
定义：流（flow）

为每条边e赋值一个整数值f(e)且满足下列特征：

1. Capacity Rule: For each edge e, $0 \le f(e) \le c(e)$
2. Conservation Rule: For each vertex $v \neq s, t$

$$\sum_{e \in E^-(v)} f(e) = \sum_{e \in E^+(v)} f(e)$$

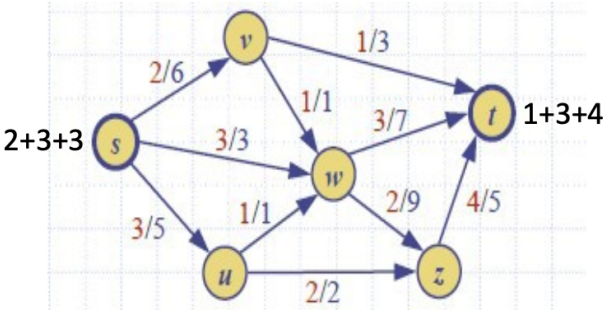
where $E^-(v)$ and $E^+(v)$ are the incoming and outgoing edges of v, respectively.



即对于一个点来说其流入量和流出量相同。

The value of a flow f, denoted $|f|$, is the total flow from the source, which is the same as the total flow into the sink.

源点流出的流等于汇聚点汇入的流。



3.2最大流 Maximum flow

一个流的值是该网络所有流中最大的，即被称作最大流。

最大流问题即寻找给定网络的最大流。

解决网络流问题需要另外一个定义：**割（Cuts）**

Flows are closely related to another concept, known as *cuts*.

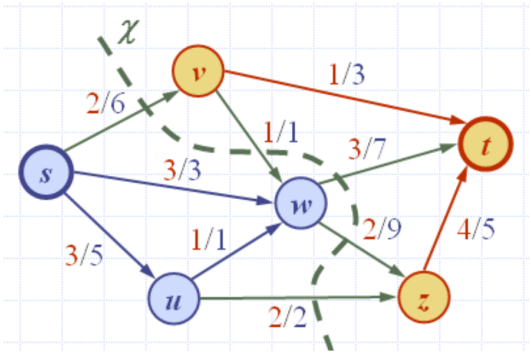
- A *cut* of a network N with source s and sink t is a partition $\chi = (V_s, V_t)$ of the vertices of N such that $s \in V_s$ and $t \in V_t$.
  - Forward edge of cut χ: origin in V_s and destination in V_t
  - Backward edge of cut χ: origin in V_t and destination in V_s
- Flow $f(\chi)$ across a cut χ: total flow of forward edges minus total flow of backward edges
- Capacity $c(\chi)$ of a cut χ: total capacity of forward edges

将网络分成两个区域，其中源点和汇聚点属于两个不同的区域，这个割的前向边（forward edge）即起始点在源点所在的分区终点在汇聚点所在的分区，后向边则反过来。

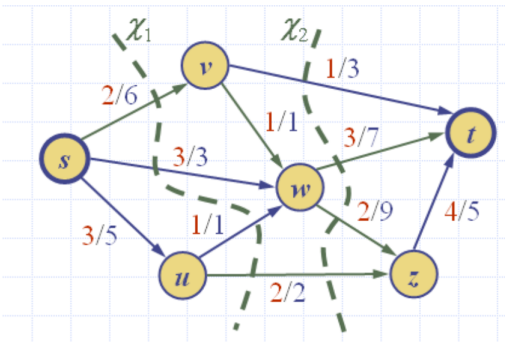
通过割X的流f(X)：所有前向边减去所有后向边。

割X的容量C(X)：所有前向边的容量。

例子：



The *total flow/capacity* of the cut are
- $f(\chi) = 8$
- $c(\chi) = 24$



The *total flow/capacity* of the cut are
- $c(\chi_1) = 12 = 6 + 3 + 1 + 2$
- $c(\chi_2) = 21 = 3 + 7 + 9 + 2$
- For both cuts, the value of the flow is $|f| = 8$

流和割 Flow and cut

引理1:

Lemma 1: Let N be a flow network, and let f be a flow for N. For any cut χ of N, the value of f is equal to the flow across cut χ, that is, $|f| = f(\chi)$.

引理2:

Lemma 2: Let N be a flow network, and let χ be a cut of N. Given any flow f for N, the flow across cut does not exceed the capacity of χ, that is, $f(\chi) \le c(\chi)$.

由引理得出定理：

Theorem 1: The value of any flow is less than or equal to the capacity of any cut, i.e., for any flow f and any cut χ, we have $|f| \le c(\chi)$.

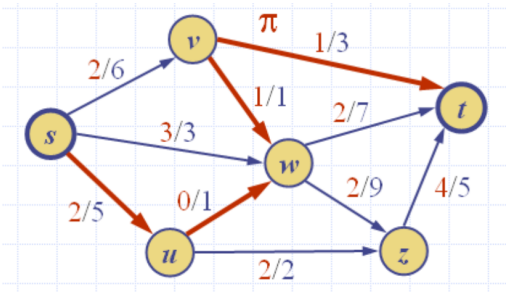
⇒ The value of a maximum flow is no more than the capacity of a minimum cut.

计算最大流还需要一个概念：**增广路径 Augmenting Path**

增广路径定义：

- Consider a flow f for a network N
- Let e be an edge from u to v:
  - *Residual capacity* of e from u to v: $\Delta_f(u,v) = c(e) - f(e)$
  - *Residual capacity* of e from v to u: $\Delta_f(v,u) = f(e)$
- Let π be a path from s to t, the *residual capacity* $\Delta_f(\pi)$ of π is the smallest of the residual capacities of the edges of π in the direction from s to t.
- A path π from s to t is an augmenting path if $\Delta_f(\pi) > 0$

注意此处的路径忽略方向。



We get the following residual capacities and flow

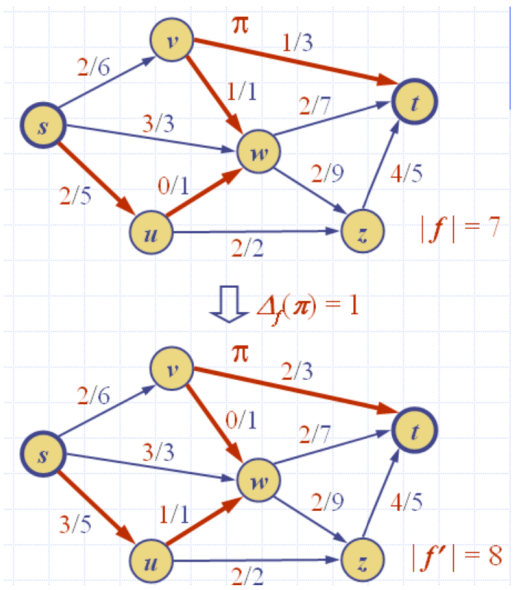
- $\Delta_f(s,u) = 3$, $\Delta_f(u,w) = 1$, $\Delta_f(w,v) = 1$, $\Delta_f(v,t) = 2$.
- $\Delta_f(\pi) = 1$
- $|f| = 7$

引理3:

Flow Augmentation

We can always add the residual capacity of an augmenting path to an existing flow and get another valid flow.

Lemma 3: Let π be an augmenting path for flow f in network N. There exists a flow f' for N of value $|f'| = |f| + \Delta_f(\pi)$



但如果网络流中不存在增广路径：

引理4:

Lemma 4: If a network N does not have an augmenting path with respect to a flow f, then f is a maximum flow. Also, there is a cut χ of N such that $|f| = c(\chi)$

即对于流量f来说如果f且网络流中不存在增广路径，则f是最大流，且存在一个割使得这个割的容量就是该最大流。

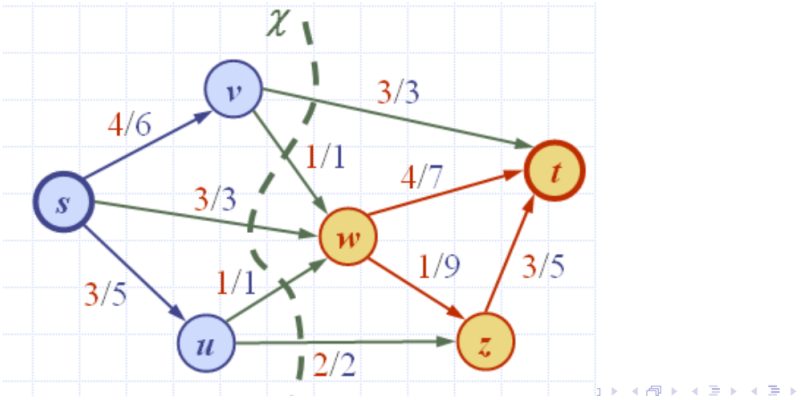
引理4和定理1可以得出定理2:

Theorem 2: The value of a maximum flow is equal to the capacity of a mimimum cut.

定理2即**最大流最小割定理**，也被称作**福特–福克逊理论** Ford–FulkersonTheorem。

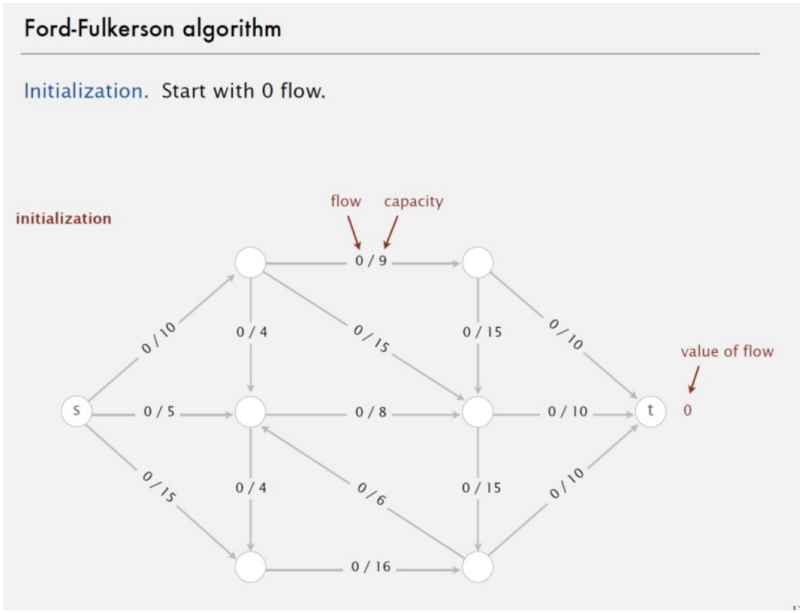
详细如下：

- Define
  - V_s set of vertices reachable from s by augmenting paths
  - V_t set of remaining vertices
- Cut $\chi = (V_s, V_t)$ has capacity $c(\chi) = |f|$
  - Forward edge: $f(e) = c(e)$
  - Backward edge: $f(e) = 0$
- Thus, flow f has maximum value and cut χ has minimum capacity. Below we have $c(\chi) = |f| = 10$.

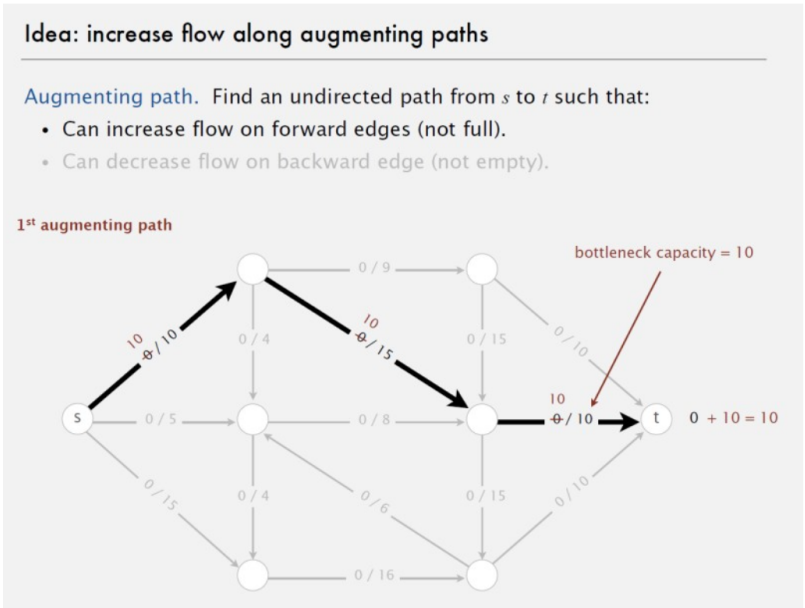


- Initially, $f(e) = 0$ for each edge e
- Repeatedly
  - Search for an augmenting path π
  - Compute bottleneck capacity $\Delta_f(\pi)$
  - Increase flow along the edges of π by bottleneck capacity $\Delta_f(\pi)$

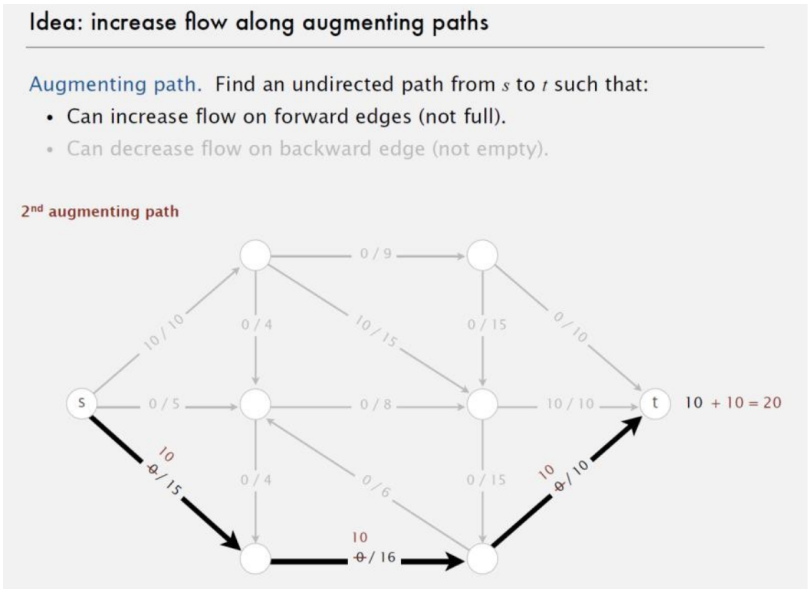
例子：



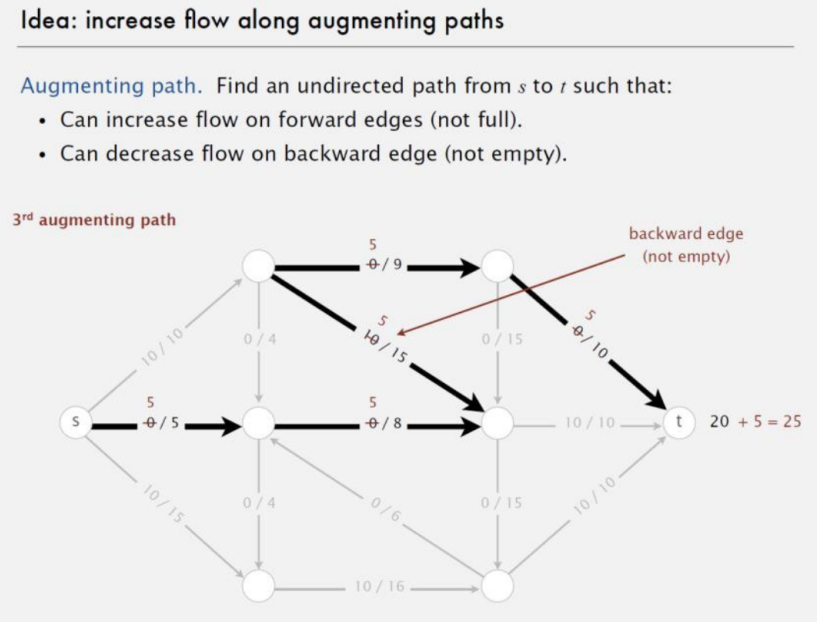
首先将所有的边进行初始化，即每条边的流都初始化为0。



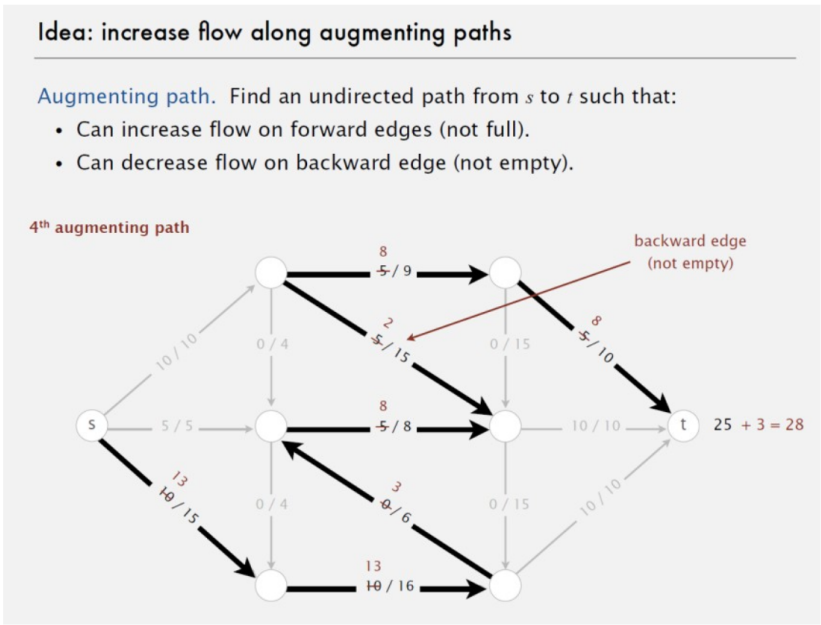
第二步寻找残存网络中存在的增广路径，寻找路径中的最小容量（bottleneck capacity），沿路径增加流（大小为bottleneck capacity）。



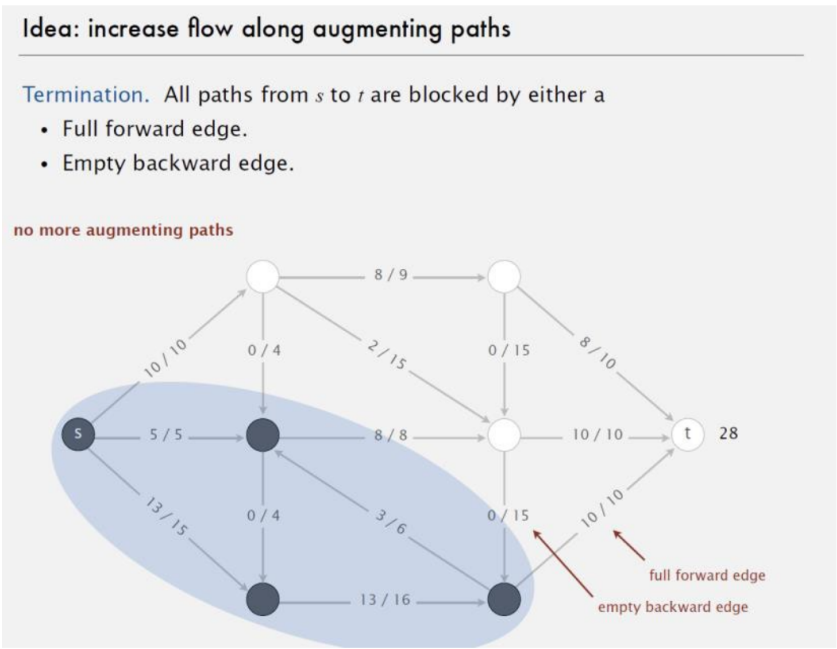
重复上述操作，即对第二条增广路径进行上述操作。



在对第三条增广路径进行操作时，注意上述操作加的操作都针对于前向边，对于后向边要进行减操作。



继续对第四条增广路径进行该操作。



找不到增广路径后，根据增广路径所到达的所有点得到的割来锁住所有的路径：割的前向边要是满的，后向边要是空的。
寻找增广路径时注意：即正向的不是满的，反向的不为0即可以形成增广路径。
寻找增广路径时的顺序不固定，最后得到的最大流的值可以通过割的容量，源顶点和汇聚顶点的流量来判断。