数据结构与算法

第九章-2 图论算法

裴文杰

计算机科学与技术学院 教授

第九章 图

9.5 最短路径问题

9.6 网络流问题

9.7 匹配问题

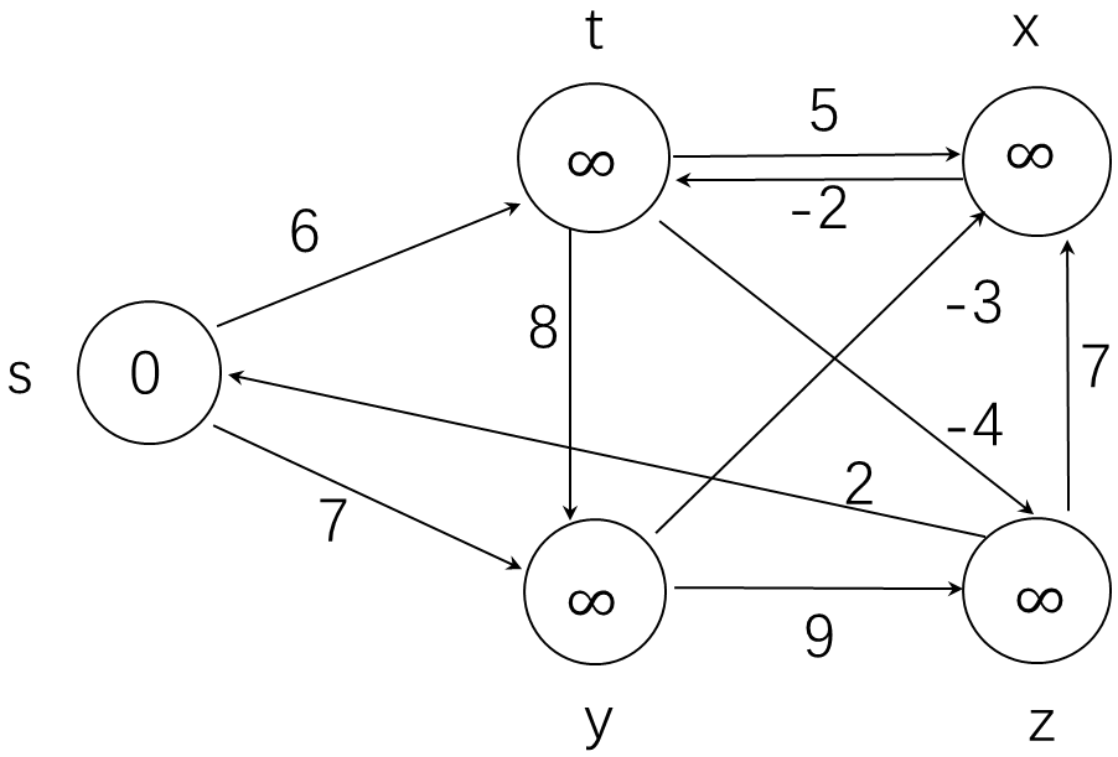
9.5 最短路径问题

单源最短路径

任意两点最短路径

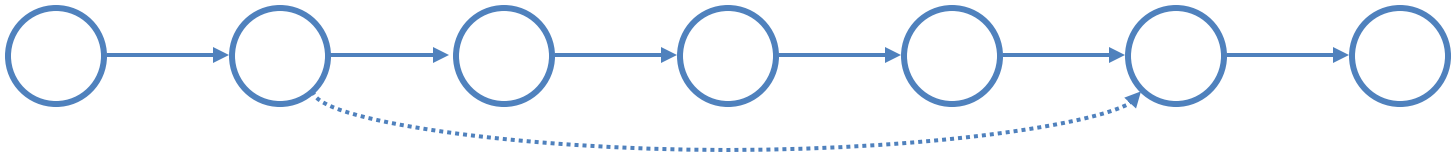
单源最短路径

- 问题:给定一个有权的有向图G，找到从给定源结点s到另一个结点v的最短路径。



最短路径的性质

- 优化子结构：最短路径包含最短子路径



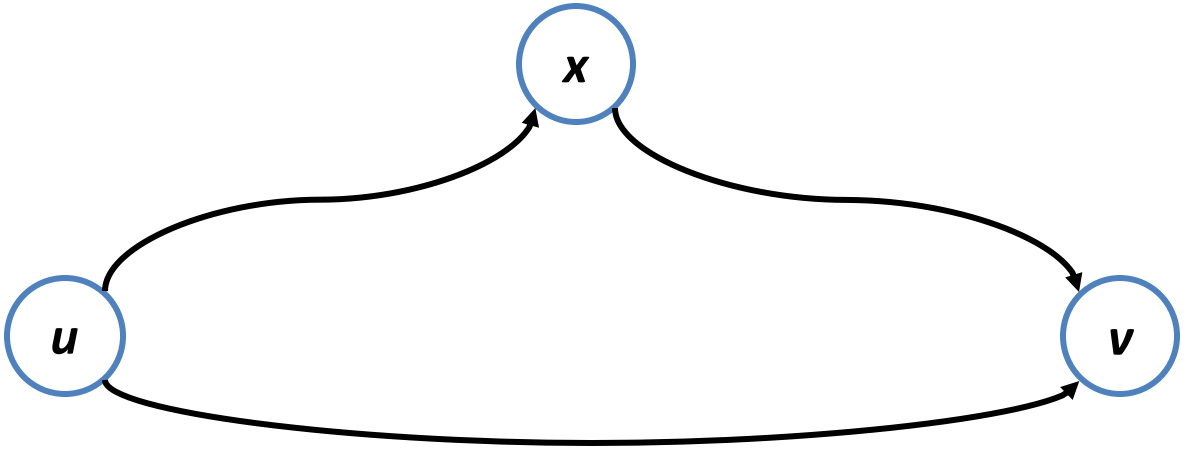
❖ 证明：如果某条子路径不是最短子路径

- 必然存在最短子路径
- 用最短子路径替换当前子路径
- 当前路径不是最短路径，矛盾！

Cut-and-Paste原理

最短路径的性质

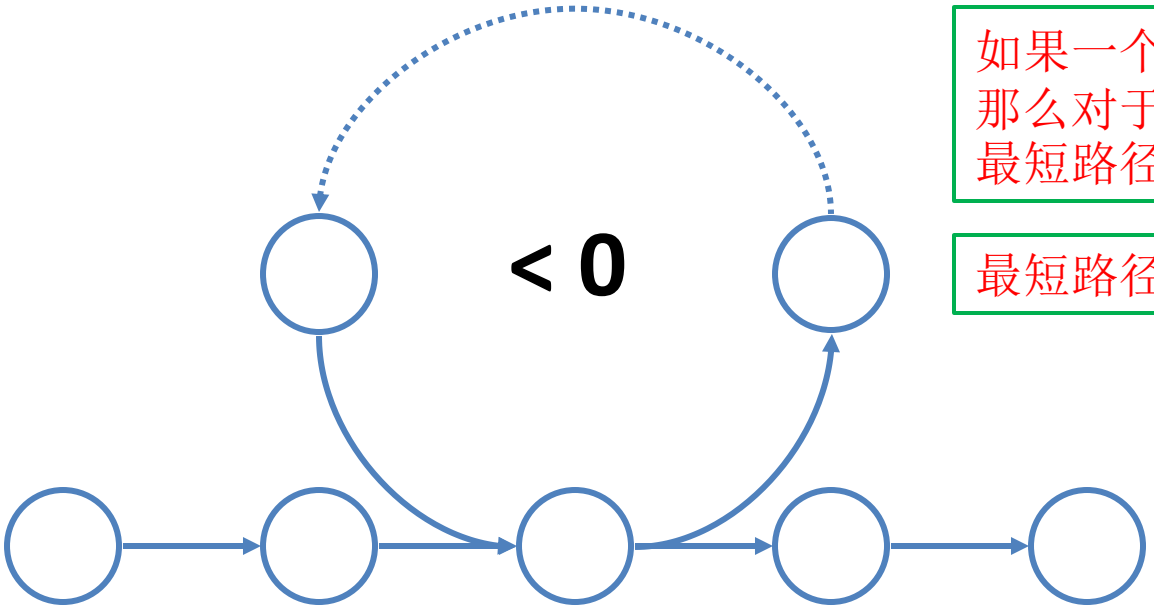
- $\delta(u, v)$ 是从u到v最短路径的权重
- 最短路径满足 *三角不等式*: $\delta(u, v) \leq \delta(u, x) + \delta(x, v)$



这条路径不会比另外两条之和长

最短路径的性质

- 如果图中包含负圈，某些最短路径可能不存在 (*Why?*)：



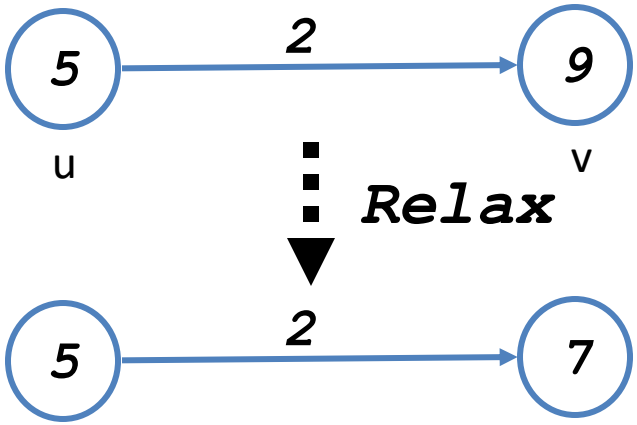
如果一个图不包含负圈，那么对于所有的结点，最短路径都有精确定义。

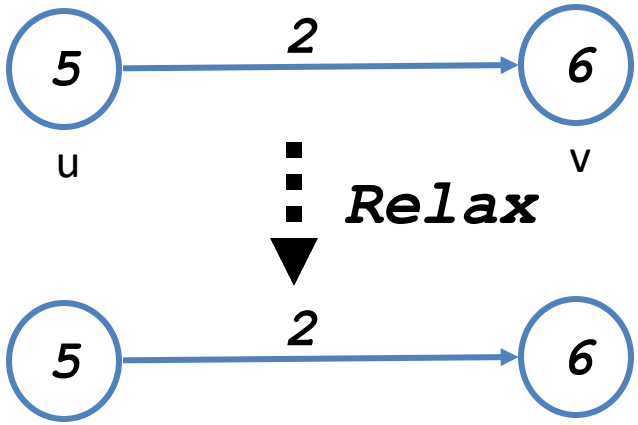
最短路径可能不唯一。

松弛

- 最短路径算法的核心技术是*松弛*

```
Relax(u,v,w) {
    if (d[v] > d[u]+w) then d[v]=d[u]+w;
}
```





Bellman-Ford 算法

- Bellman-Ford算法解决一般情况下的单源最短路径问题。
  - 边的权重可以为负值
  - 给定带权重的有向图，Bellman-Ford算法返回一个bool变量，表明是否存在一个从源节点可以到达的权重为负的环路（负圈）

Bellman-Ford 算法

```
BellmanFord()
    for each v ∈ V
        d[v] = ∞;
    d[s] = 0;
    for i=1 to |V|-1
        for each edge (u,v) ∈ E
            Relax(u,v, w(u,v));
    for each edge (u,v) ∈ E
        if (d[v] > d[u] + w(u,v))
            return "no solution";
```

为什么是|V|-1?

```
Relax(u,v,w): if (d[v] > d[u]+w) then d[v]=d[u]+w
```

Bellman-Ford 算法

```
BellmanFord()
    for each v ∈ V
        d[v] = ∞;
    d[s] = 0;
    for i=1 to |V|-1
        for each edge (u,v) ∈ E
            Relax(u,v, w(u,v));
    for each edge (u,v) ∈ E
        if (d[v] > d[u] + w(u,v))
            return "no solution";
```

初始化 d

松弛：
进行 |V|-1 轮，
松弛每条边

检验结果
何时得到解?

没有从源节点可以到达的负圈时。

```
Relax(u,v,w): if (d[v] > d[u]+w) then d[v]=d[u]+w
```

Bellman-Ford 算法

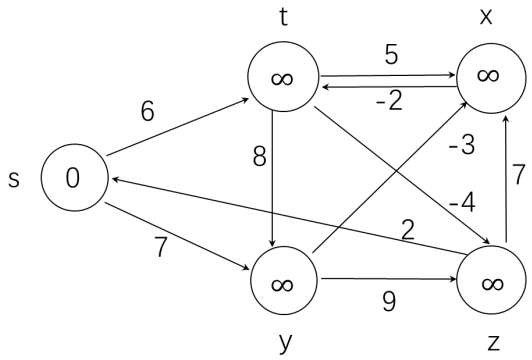
```
BellmanFord()
    for each v ∈ V
        d[v] = ∞;
    d[s] = 0;
    for i=1 to |V|-1
        for each edge (u,v) ∈ E
            Relax(u,v, w(u,v));
    for each edge (u,v) ∈ E
        if (d[v] > d[u] + w(u,v))
            return "no solution";
```

运行时间是多少？

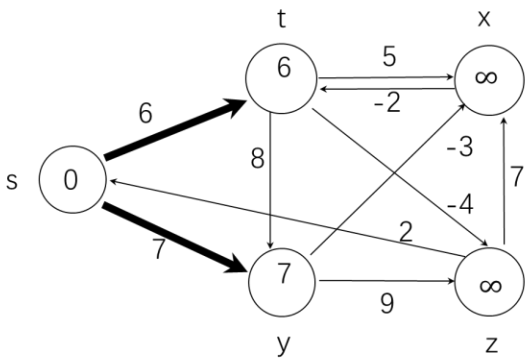
A: O(VE)

```
Relax(u,v,w): if (d[v] > d[u]+w) then d[v]=d[u]+w
```

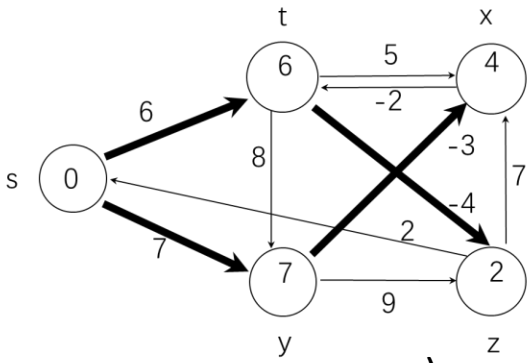
Bellman-Ford 算法



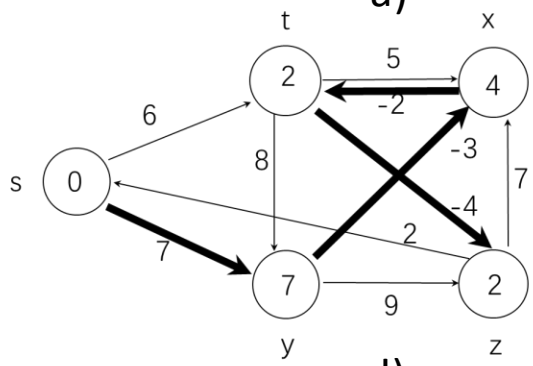
a)



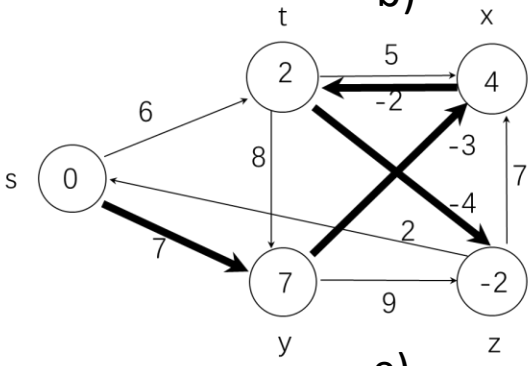
b)



c)



d)



e)

注意：处理边的顺序会影响到收敛速度

前驱节点：得到当前距离的路径的前一个节点

Bellman-ford算法的执行过程。源点是顶点s。d值被标记在顶点内，加粗的边指示了前趋值：如果边（u，v）被加粗，则v的前驱节点π[v]=u。在这个特定的例子中，每一趟按照如下顺序对边进行松弛: (t, x), (t, y), (t, z), (x, t), (y, x), (y, z), (z, x), (z, s), (s, t), (s, y)。a)示出了对边进行第一趟操作前的情况。b)至e）示出了每一趟连续对边操作后的情况，e)中d和前趋值是最终结果。

Bellman-Ford 算法

- 算法正确性证明
  - 证明 在没有权重为负值的环路的情况下，该算法能够正确计算出从源结点到可以到达的所有结点之间的最短路径权重，即：|V|-1 轮之后，对于所有从源节点可以到达的结点v，都有 $d[v] = \delta(s,v)$

Bellman-Ford 算法

- 算法正确性证明
  - 证明： |V|-1轮之后，所有d 的值正确，即都是最短路径

路径松弛性质：

设 $p = <s, v_1, v_2, \ldots, v_k>$为源节点s到$v_k$的一条最短路径，如果对p中所有边沿路径$(s,v_1)$，$(v_1,v_2),\ldots,(v_{k-1},v_k)$依次进行松弛，那么$v_k$的节点权重$d[v_k]$即是s到$v_k$的最短路径权重。该性质的成立与任何其他的松弛操作无关，即使这些松弛操作是与对p上的边所进行的松弛操作穿插进行的。

设 $p = <s, v_1, v_2, \ldots, v_k>$为源节点s到$v_k$的一条最短路径，p最多包含|V|-1条边，而算法中每次对于节点的循环中都要对所有边松弛一遍，每次循环可以松弛p中的一条边，因此Bellman-Ford算法可以确保p中的所有边可以依次被松弛一遍。那么根据 路径松弛性质，v_k的节点权重$d[v_k]$即是s到v_k的最短路径距离。

Bellman-Ford 算法

- 算法正确性证明
  - 证明： |V|-1轮之后，所有d 的值正确，即都是最短路径

考虑从 s 到 v的最短路径:

$s \rightarrow v_1 \rightarrow v_2 \rightarrow v_3 \rightarrow v_4 \rightarrow v$

Bellman-Ford算法进行|V|-1轮边的松弛：

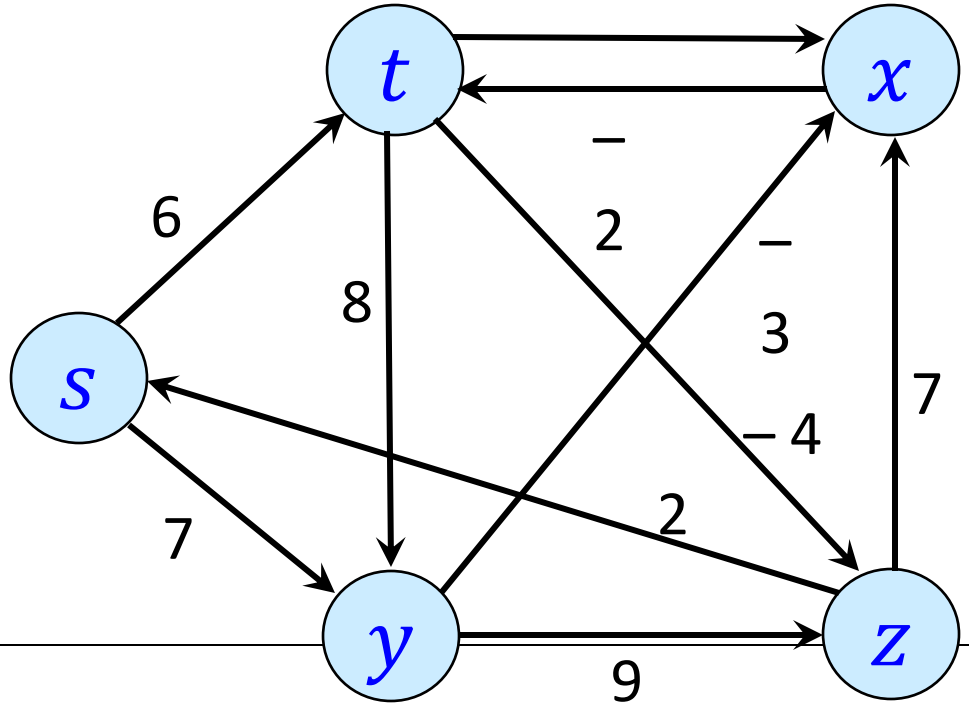
- 开始, d[s] = 0 正确, 之后不发生变化
- 1轮之后, d[v_1] 正确,之后不发生变化
- 2轮之后, d[v_2]正确, 之后不发生变化

…

- |V| - 1 轮之后停下

Bellman-Ford 算法
——动态规划视角

- 优化子结构：最短路径包含最短子路径
- 重叠子问题：从s到x，可经过多条可能的路径（s-t-x, s-y-x, …），而不同的最短路径可能包含同样的子路径（子问题）



对！这个Bellman就是发明动态规划算法的那个Bellman！！！
他和 Lester Ford一起设计了Bellman-Ford算法

Bellman-Ford 算法
——动态规划视角

- 递归定义最优解代价：
- $d[k][v]$表示至多有k条边的最短路径的代价
- $$d[k][v] = min\begin{cases} min_{(u,v)\in E}\{d[k-1][u] + w(u,v)\} \\ d[k-1][v] \end{cases}$$

其实只需关注进入的v的边

- 扫描顺序：$k = 1$ to $|V-1|$

递归公式的核心本质上就是relax操作，
该动态规划算法本质上就是Bellman-Ford算法

Bellman-Ford 算法 ——动态规划视角

```
BellmanFord()
    for each v ∈ V
        d[v] = ∞;
    d[s] = 0;
    for i=1 to |V|-1
        for each edge (u,v) ∈ E
            Relax(u,v, w(u,v));
    for each edge (u,v) ∈ E
        if (d[v] > d[u] + w(u,v))
            return "no solution";
```

$$d[k][v] = min\begin{cases} min_{(u,v)\in E}\{d[k-1][u] + w(u,v)\} \\ d[k-1][v] \end{cases}$$

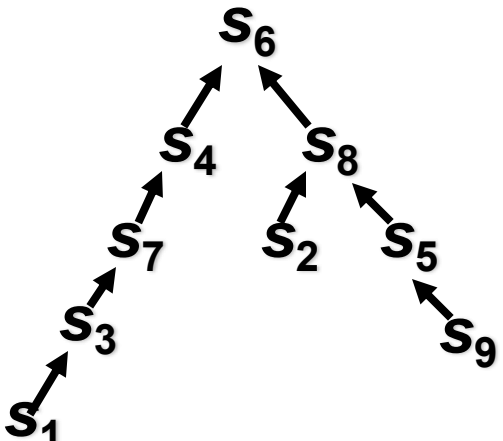
```
Relax(u,v,w): if (d[v] > d[u]+w) then d[v]=d[u]+w
```

DAG 中最短路径

- Problem: 寻找 DAG(有向无环图)中最短路径
  - Bellman-Ford 时间是 O(VE).
  - *能否做的好一点呢?*
  - Idea: 使用拓扑排序
    - 如果沿着最短路径作松弛操作，则可以一遍完成
    - DAG中的每条路径都是拓扑排序得到结点序列的一个子序列，那么，如果按照这个顺序（拓扑序列）处理，我们将沿着最短路径进行处理（路径松弛定理），扫描一次就够了.

为什么不能有环?
有环图可以进行拓扑排序吗?

为什么比Bellman-Ford快?
排除不符合拓扑排序序列的边的遍历



拓扑排序:

$s_1\ s_3\ s_7\ s_4\ s_9\ s_5\ s_2\ s_8\ s_6$

DAG 中最短路径

- Problem: 寻找 DAG(有向无环图)中最短路径
  - Idea: 使用拓扑排序

```
DAG-Shortest-Paths(G, w, s)
    topologically sort the vertices of G // 拓扑排序
    for each v ∈ V
        d[v] = ∞
    d[s] = 0
    for each u, taken in topologically sorted order //按照拓扑排序依次取出结点
        for each v ∈ G.Adj[u] // 对u的每个邻居结点进行松弛对应的边
            Relax(u, v, w(u,v)) //实际效果，对每条边都松弛了一遍
```

运行时间是多少？
拓扑排序：O(V+E)
总共松弛操作：O(E)
初始化：O(V)
总共： O(V+E)

平摊分析

- 正确性证明:

设 $p = \langle s, v_1, v_2, \ldots, v_k \rangle$为源节点s到$v_k$的一条最短路径，因为算法根据拓扑排序的次序对结点进行处理，所以对路径p上的边的松弛次序是按照p的路径进行的，根据路径松弛性质， $d[v_k]$即是s到v_k的最短路径距离，即$d[v_k] = \delta(s, v_k)$

拓扑排序序列不唯一，是否需要尝试所有可能的拓扑序列？
不需要，因为对于任意一个拓扑序列，所有的路径都是其子序列。

Dijkstra 算法

- 如果图中没有负边，Dijkstra算法可以超越Bellman-Ford 算法
- 类似Best-First 搜索
  - 从最小优先队列（最小堆）中取结点
- 类似Prim算法
  - 从源点逐渐扩展到所有结点
  - 使用以d[v]为键的优先队列

Dijkstra 算法

```
Dijkstra(G)
1    for each v ∈ V
2        d[v] = ∞;
3    d[s] = 0; S = ∅; Q = V;
4    while (Q ≠ ∅)
5        u = ExtractMin(Q);
6        S = S U {u};
7        for each v ∈ u->Adj[]
8            if (d[v] > d[u]+w(u,v))
9                d[v] = d[u]+w(u,v);
```

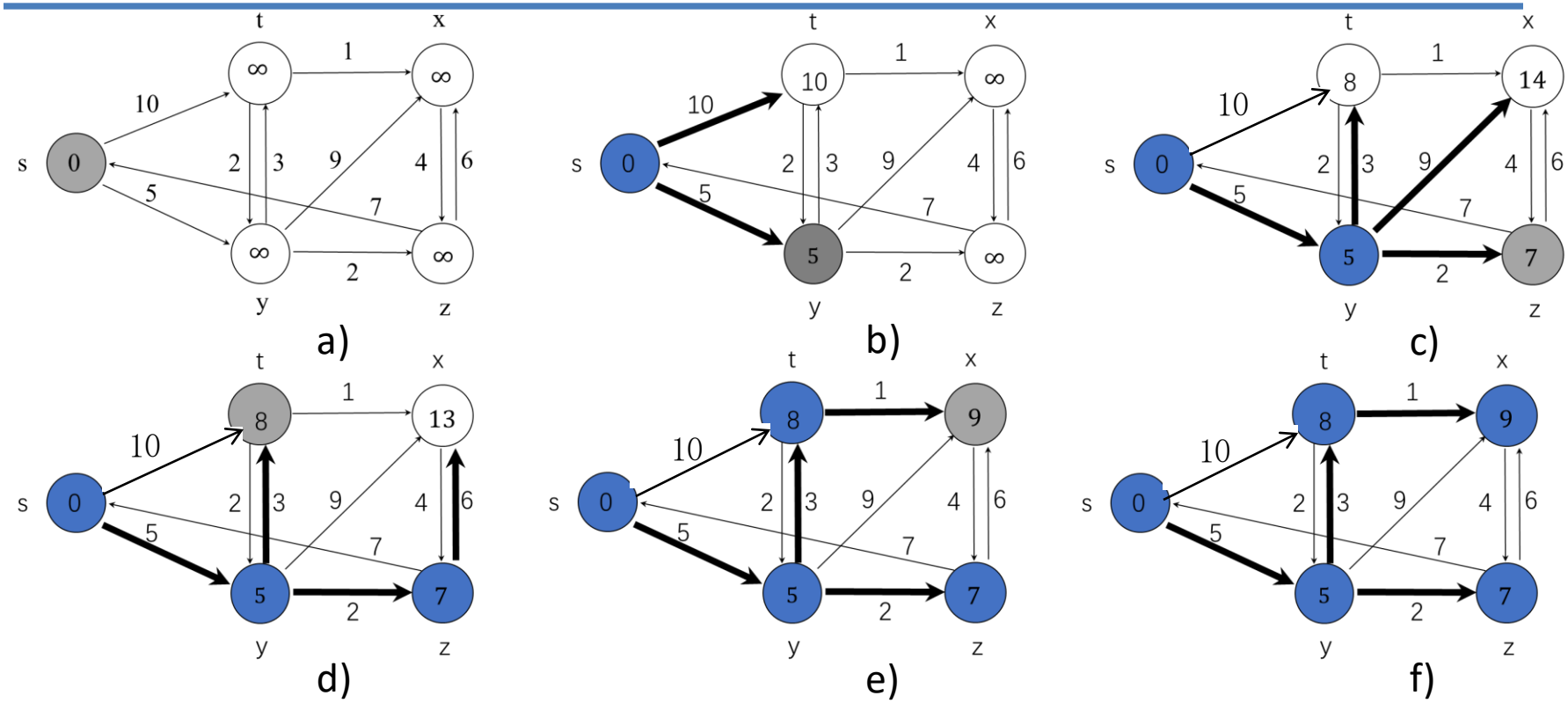
将G中结点分为两部分：S和Q，S为已经找到最短路径的结点集合，Q（V-S）为还未找到最短路径的结点集合。
算法重复从Q中选择最短路径估计值最小的结点u，加入到S中，然后对所有从u出发的边进行松弛。
所以是一种贪心算法。
Q用最小优先队列（最小堆）实现。

这里的邻边指的是u的出边，不包括入边

松弛步骤

Note: 调用了 Q->DecreaseKey()

Dijkstra 算法



Dijkstra算法的执行过程：源点**s**为最左端顶点。最短路径估计值被标记在顶点内，黑边指出了前趋的值。蓝色顶点在集合**S**中，而白色定点在最小优先队列**Q=V-S**中。**a**）第**4~9**行**While**循环第一次迭代前的情形。阴影覆盖的顶点具有最小的**d**的值，而且在算法第**5**行被选为顶点**u**。**b)**至**f) while**循环在每一次连续迭代后的情形。每个图中阴影覆盖的顶点被选作下一次迭代第**5**行的顶点**u**。**f)**图中的**d**和前趋值是最终结果。

Dijkstra 算法

```
Dijkstra(G)
1    for each v ∈ V
2        d[v] = ∞;
3    d[s] = 0; S = ∅; Q = V;
4    while (Q ≠ ∅)
5        u = ExtractMin(Q);
6        S = S U {u};
7        for each v ∈ u->Adj[]
8            if (d[v] > d[u]+w(u,v))
9                d[v] = d[u]+w(u,v);
```

减小d值，调用了 Q->DecreaseKey()

运行时间是多少？

取决于优先队列的实现方式

最小堆：

ExtractMin 和 DecreaseKey 操作时间复杂性都是logV，

总时间为：O((V+E)lgV)

斐波那契堆:

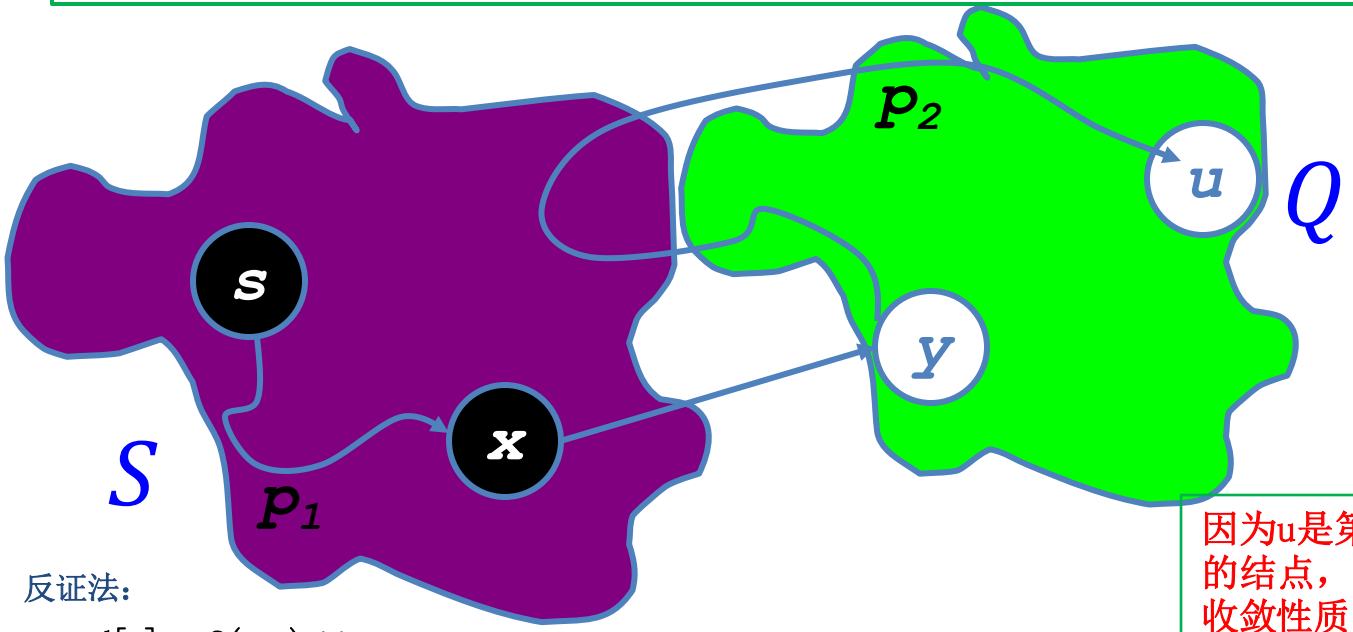
Decrease-Key代价为*O(1)*，

总时间为O(VlgV+E)

正确性：我们必须证明，当u从Q中取出时，它已经收敛了

Dijkstra 算法的正确性

需要证明：对于每个节点u∈V，当结点u被加入S时，有d[u]=δ(s,u)



收敛性质：
如果s～x->y是到y的最短路径，且d[x] = δ(s,x)，那么在对边(x,y)松弛后，有d[y] = δ(s,y)

因为u是第一个不满足d[u] = δ(s,u)的结点，因此d[x] = δ(s,x)，根据收敛性质，d[y] = δ(s,y)

为什么y一定存在？
如果u和x直接相连，那么根据收敛定理，d[u]= δ(s,u)，因此如果d[u]>δ(s,u)，那么一定存在y，介于x和u之间。

反证法：

- d[v] ≥ δ(s,v) ∀v
- 令u是第一个从Q选出的加入到S中时不满足d[u]= δ(s,u)的点 ⇒d[u]>δ(s,u)，即之前加入到S中的点i都满足d[i]= δ(s,i)
- 令y是s→u真实最短路径上的第一个Q中的结点，且前驱结点为x ⇒ d[y]=δ(s,y)（收敛性质）

  d[u] > δ(s,u)
  = δ(s,y) + δ(y,u) （优化子结构）
  = d[y] + δ(y,u)
  ≥ d[y]
- 但是如果 d[u] > d[y]，则从Q中不会选择u. 矛盾。

9.5 最短路径问题

单源最短路径

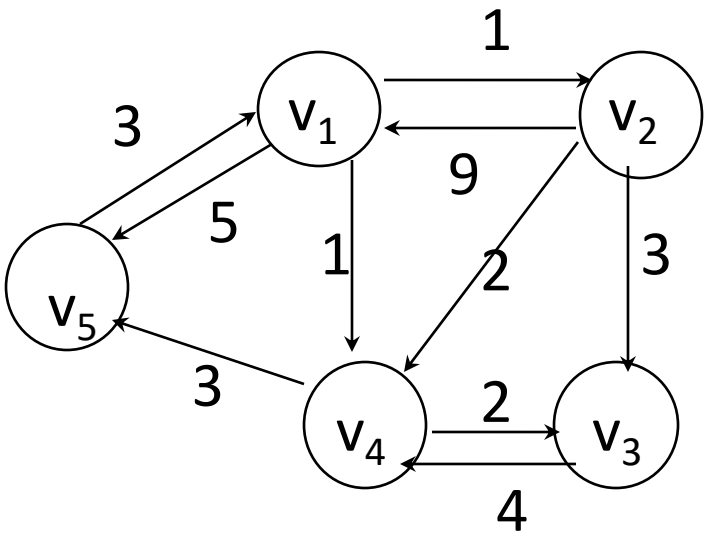
任意两点最短路径

任意两点最短路径

- 问题：给定一个边加权的有向图$G=(V,E)$，找到图中每一对结点(u,v)间最短路径
  - 图G可能包含负边但是不包含负圈
- 依次把有向图G中的每个顶点作为源节点，执行$|V|$次单源最短路径算法，从而得到每对顶点之间的最短路径，有没有更快的算法？
- 动态规划-Floyd算法

图和权矩阵

	1	2	3	4	5
1	0	1	∞	1	5
2	9	0	3	2	∞
3	∞	∞	0	4	∞
4	∞	∞	2	0	3
5	3	∞	∞	∞	0



Floyd算法

- 动态规划算法
- 如何定义子问题？

  - 一种方法是将路径（经过的结点集合）限制在仅包含一个有限集合中的结点

  - 开始这个集合是空的

  - 这个集合可以一直增长到包含所有结点。

Floyd算法

- 如何定义子问题？
  - 令 $D^{(k)}[i, j]$=从v_i 到v_j仅包含 $\{v_1, v_2, \ldots, v_k\}$ 的最短路径长度，即该路径的中间结点只取自这个结点集合$\{v_1, v_2, \ldots, v_k\}$ 。
  - $D^{(0)}=W$ 权矩阵 $D^{(0)}$没有经过任何中间节点，因此就等于W
  - $D^{(n)}=D$ 目标矩阵
- 如何从$D^{(k-1)}$计算 $D^{(k)}$？即递归方程怎么建立。

Floyd算法

- 递归方程
  - 因为

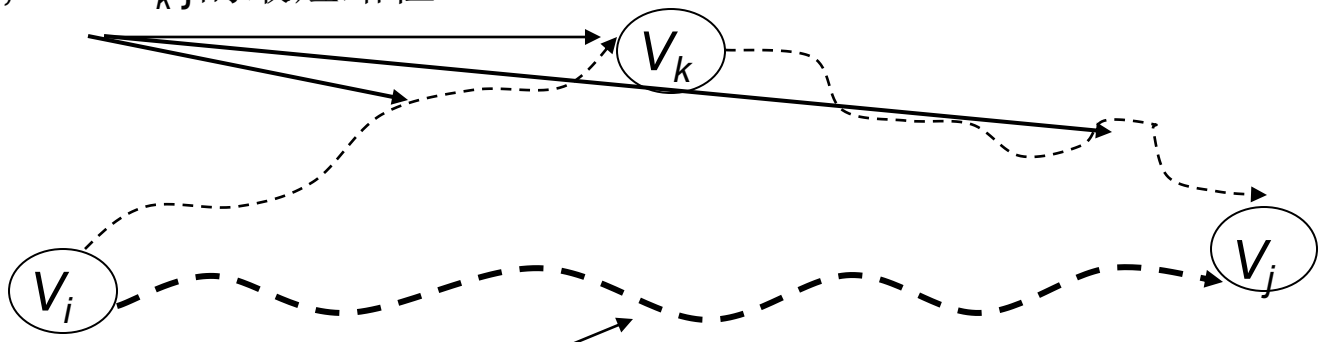
$$D^{(k)}[i,j] = D^{(k-1)}[i,j] \quad \text{or}$$

$$D^{(k)}[i,j] = D^{(k-1)}[i,k] + D^{(k-1)}[k,j].$$

该算法与离散数学中求传递闭包的Warshall算法原理相似！

递归方程决定了自底向上的计算次序：需要先计算出所有的D^{k-1}，然后才能计算D^{k}。

仅包含$\{V_1, \ldots V_k\}$的最短路径



$$D^{(k)}[i,j] = \begin{cases} w_{ij} & if \ k = 0 \\ min\{D^{(k-1)}[i,j], D^{(k-1)}[i,k] + D^{(k-1)}[k,j]\} & if \ k \geq 1 \end{cases}$$

Floyd算法

C[i][j]为邻接矩阵中(i, j)的值

Floyd算法的基本思想(直观理解):

- 假设求顶点v_i到顶点v_j的最短路径。如果从 v_i 到 v_j 存在一条长度为C[i][j]的路径，该路径不一定是最短路径，尚需进行 n 次试探。
- 首先考虑路径 (v_i, v_0, v_j) 是否存在。如果存在，则比较 (v_i, v_j) 和 (v_i, v_0, v_j) 的路径长度取长度较短者为从 v_i 到 v_j 的中间顶点的序号不大于0的最短路径。
- 假设在路径上再增加一个顶点 v_1，也就是说，如果 $(v_i,\ldots, v_1)$和 $(v_1,\ldots, v_j)$ 分别是当前找到的中间顶点的序号不大于0的最短路径,那么 $(v_i,\ldots, v_1,\ldots, v_j)$就是有可能是从$v_i$到$v_j$的中间顶点的序号不大于1的最短路径。将它与已经得到的从v_i到v_j中间顶点序号不大于0的最短路径相比较，从中选出中间顶点的序号不大于1的最短路径，再增加一个顶点v_2, 继续进行试探。
- 一般情况下，若 $(v_i,\ldots, v_k)$ 和 $(v_k, \ldots, v_j)$ 分别是从 v_i到v_k和从v_k到v_j的中间顶点序号不大于 k-1 的最短路径，则将 $(v_i,\ldots, v_k,\ldots, v_j)$ 和已经得到的从 v_i到 v_j且中间顶点序号不大于k-1的最短路径相比较，其长度较短者便是从v_i到 v_j 的中间顶点的序号不大于 k 的最短路径。

$$D^{(k)}[i,j] = \begin{cases} w_{ij} & if\ \ k = 0 \\ min\{D^{(k-1)}[i,j], D^{(k-1)}[i,k] + D^{(k-1)}[k,j]\} & if\ k \geq 1 \end{cases}$$

Floyd算法

1. $D^0 \leftarrow W$ //初始化D
2. $P \leftarrow 0$ // 初始化 P
3. for $k \leftarrow 1$ to n do
4. for $i \leftarrow 1$ to n do
5. for $j \leftarrow 1$ to n do
6. if $(D^{k-1}[\,i,j\,] > D^{k-1}[\,i,k\,] + D^{k-1}[\,k,j\,])$
7. then $D^k[\,i,j\,] \leftarrow D^{k-1}[\,i,k\,] + D^{k-1}[\,k,j\,]$
8. $P[\,i,j\,] \leftarrow k;$
9. else $D^k[\,i,j\,] \leftarrow D^{k-1}[\,i,j\,]$

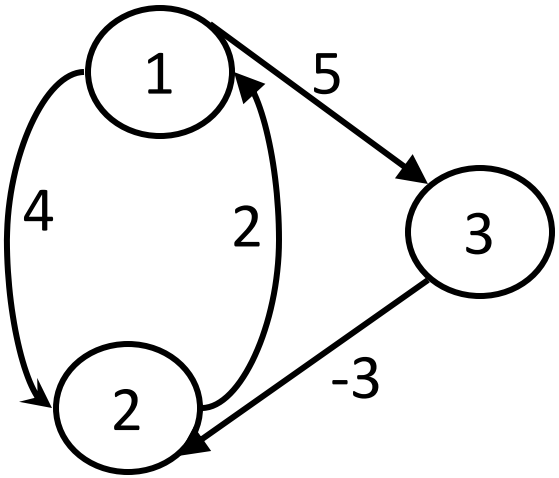
注意：k循环在最外层，因为如果k循环在i，j循环里面，无法自底向上求解，会造成计算结果之间互相依赖的难以解耦：即计算当前项所依赖的其他项还未计算。

P：记录构造优化解的信息

运行时间是多少？

A： $\Theta(V^3)$

示例 1

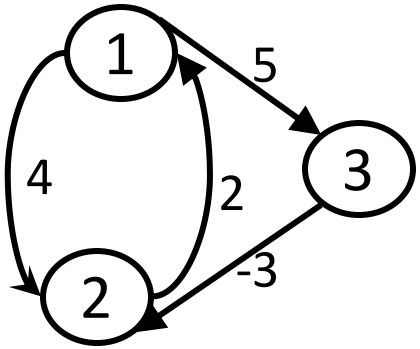


$W = D^0 =$

	1	2	3
1	0	4	5
2	2	0	∞
3	∞	-3	0

$P =$

	1	2	3
1	0	0	0
2	0	0	0
3	0	0	0



D^0 =

	1	2	3
1	0	4	5
2	2	0	∞
3	∞	-3	0

v_k作为起始点或者终点，不需要考虑。

D^1 =

	1	2	3
1	0	4	5
2	2	0	7
3	∞	-3	0

$D^1[2,3] = \min(D^0[2,3], D^0[2,1]+D^0[1,3])$

$= \min (\infty, 7)$

$= 7$

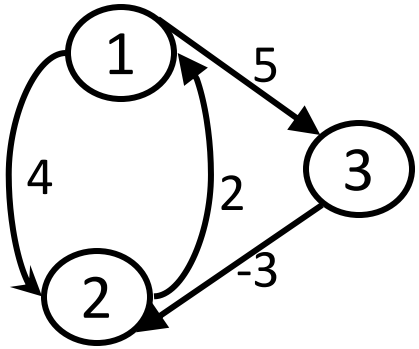
$D^1[3,2] = \min(D^0[3,2], D^0[3,1]+D^0[1,2])$

$= \min (-3,\infty)$

$= -3$

P =

	1	2	3
1	0	0	0
2	0	0	1
3	0	0	0



$D^1 =$

	1	2	3
1	0	4	5
2	2	0	7
3	∞	-3	0

$D^2 =$

	1	2	3
1	0	4	5
2	2	0	7
3	-1	-3	0

$D^2[1,3] = \min(D^1[1,3], D^1[1,2]+D^1[2,3])$

$= \min (5, 4+7)$

$= 5$

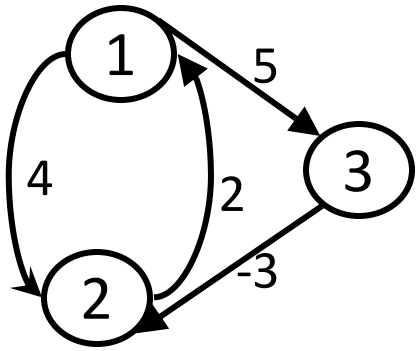
$D^2[3,1] = \min(D^1[3,1], D^1[3,2]+D^1[2,1])$

$= \min (\infty, -3+2)$

$= -1$

$P =$

	1	2	3
1	0	0	0
2	0	0	1
3	2	0	0



D^2 =

	1	2	3
1	0	4	5
2	2	0	7
3	-1	-3	0

D^3 =

	1	2	3
1	0	2	5
2	2	0	7
3	-1	-3	0

$D^3[1,2] = \min(D^2[1,2], D^2[1,3]+D^2[3,2])$

$= \min(4, 5+(-3))$

$= 2$

$D^3[2,1] = \min(D^2[2,1], D^2[2,3]+D^2[3,1])$

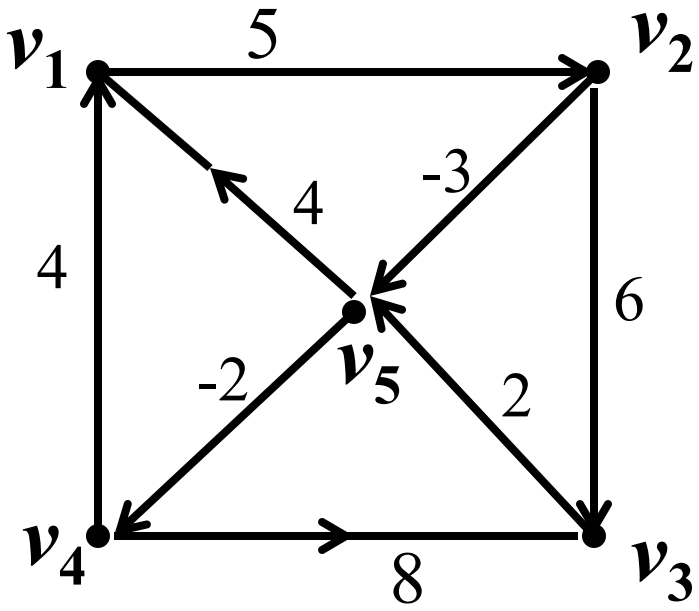
$= \min(2, 7+(-1))$

$= 2$

P =

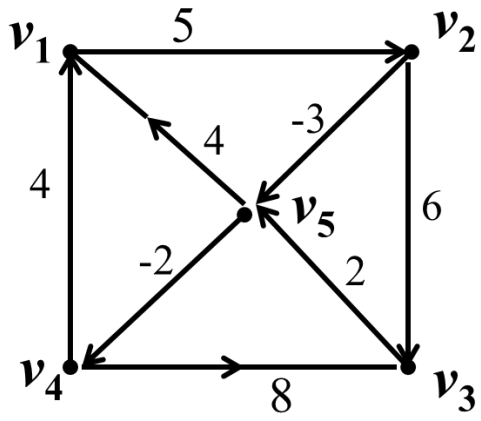
	1	2	3
1	0	3	0
2	0	0	1
3	2	0	0

示例 2



$$W = \boldsymbol{D}^{(0)} = \begin{bmatrix} 0 & 5 & \infty & \infty & \infty \\ \infty & 0 & 6 & \infty & -3 \\ \infty & \infty & 0 & \infty & 2 \\ 4 & \infty & 8 & 0 & \infty \\ 4 & \infty & \infty & -2 & 0 \end{bmatrix}$$

$$P = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$



$$D^{(0)} = \begin{bmatrix} 0 & 5 & \infty & \infty & \infty \\ \infty & 0 & 6 & \infty & -3 \\ \infty & \infty & 0 & \infty & 2 \\ 4 & \infty & 8 & 0 & \infty \\ 4 & \infty & \infty & -2 & 0 \end{bmatrix}$$

$$D^{(1)} = \begin{bmatrix} 0 & 5 & \infty & \infty & \infty \\ \infty & 0 & 6 & \infty & -3 \\ \infty & \infty & 0 & \infty & 2 \\ 4 & 9 & 8 & 0 & \infty \\ 4 & 9 & \infty & -2 & 0 \end{bmatrix}$$

v_k作为起始点或者终点，不需要考虑。

$k = 1$

$$P = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{bmatrix}$$

$$D^{(1)}[2,3] = \min\left(D^{(0)}[2,3], D^{(0)}[2,1] + D^{(0)}[1,3]\right)$$

$$= \min(6, \infty + \infty) = 6$$

$$D^{(1)}[2,4] = \min\left(D^{(0)}[2,4], D^{(0)}[2,1] + D^{(0)}[1,4]\right)$$

$$= \min(\infty, \infty + 4) = \infty$$

$$D^{(1)}[2,5] = \min\left(D^{(0)}[2,5], D^{(0)}[2,1] + D^{(0)}[1,5]\right)$$

$$= \min(-3, \infty + \infty) = -3$$

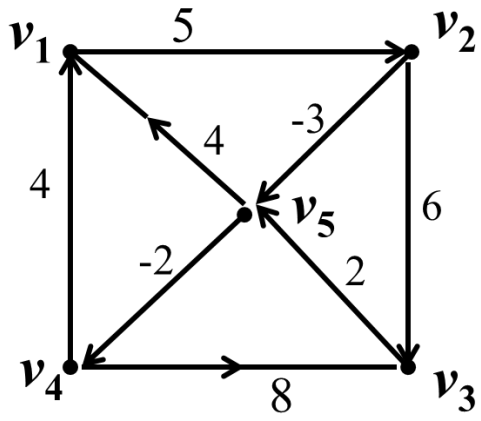
。 。 。

$$D^{(1)}[4,2] = \min\left(D^{(0)}[4,2], D^{(0)}[4,1] + D^{(0)}[1,2]\right)$$

$$= \min(\infty, 4 + 5) = 9$$

$$D^{(1)}[5,2] = \min\left(D^{(0)}[5,2], D^{(0)}[5,1] + D^{(0)}[1,2]\right)$$

$$= \min(\infty, 4 + 5) = 9$$



$$D^{(1)} = \begin{bmatrix} 0 & 5 & \infty & \infty & \infty \\ \infty & 0 & 6 & \infty & -3 \\ \infty & \infty & 0 & \infty & 2 \\ 4 & 9 & 8 & 0 & \infty \\ 4 & 9 & \infty & -2 & 0 \end{bmatrix}$$

$$D^{(2)} = \begin{bmatrix} 0 & 5 & 11 & \infty & 2 \\ \infty & 0 & 6 & \infty & -3 \\ \infty & \infty & 0 & \infty & 2 \\ 4 & 9 & 8 & 0 & 6 \\ 4 & 9 & 15 & -2 & 0 \end{bmatrix}$$

$k = 2$

$$P = \begin{bmatrix} 0 & 0 & 2 & 0 & 2 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 2 \\ 0 & 1 & 2 & 0 & 0 \end{bmatrix}$$

$$D^{(2)}[1,3] = \min\left(D^{(1)}[1,3], D^{(1)}[1,2] + D^{(1)}[2,3]\right)$$
$$= \min(\infty, 5 + 6) = 11$$

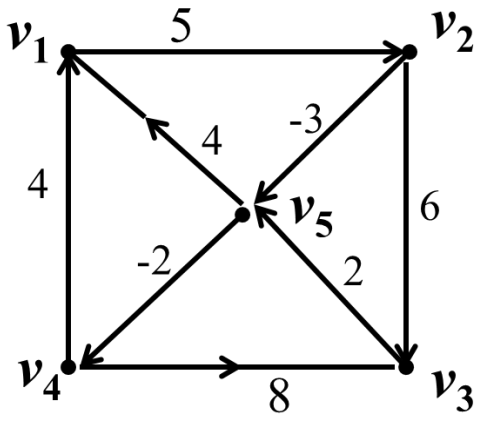
$$D^{(2)}[1,4] = \min\left(D^{(1)}[1,4], D^{(1)}[1,2] + D^{(1)}[2,4]\right)$$
$$= \min(\infty, 5 + \infty) = \infty$$

$$D^{(2)}[1,5] = \min\left(D^{(1)}[1,5], D^{(1)}[1,2] + D^{(1)}[2,5]\right)$$
$$= \min(\infty, 5 - 3) = 2$$

。 。 。

$$D^{(2)}[4,5] = \min\left(D^{(1)}[4,5], D^{(1)}[4,2] + D^{(1)}[2,5]\right)$$
$$= \min(\infty, 9 - 3) = 6$$

$$D^{(2)}[5,3] = \min\left(D^{(1)}[5,3], D^{(1)}[5,2] + D^{(1)}[2,3]\right)$$
$$= \min(\infty, 9 + 6) = 15$$

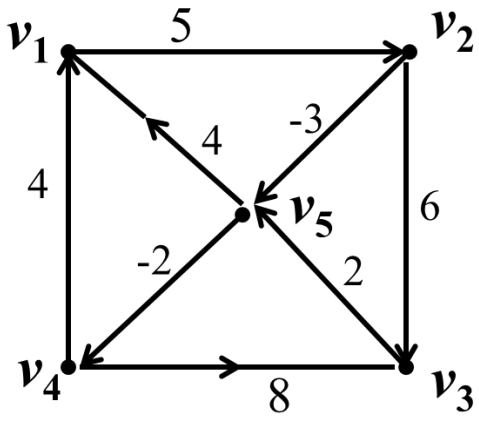


$$D^{(2)} = \begin{bmatrix} 0 & 5 & 11 & \infty & 2 \\ \infty & 0 & 6 & \infty & -3 \\ \infty & \infty & 0 & \infty & 2 \\ 4 & 9 & 8 & 0 & 6 \\ 4 & 9 & 15 & -2 & 0 \end{bmatrix}$$

$$D^{(3)} = \begin{bmatrix} 0 & 5 & 11 & \infty & 2 \\ \infty & 0 & 6 & \infty & -3 \\ \infty & \infty & 0 & \infty & 2 \\ 4 & 9 & 8 & 0 & 6 \\ 4 & 9 & 15 & -2 & 0 \end{bmatrix}$$

$k = 3$

$$P = \begin{bmatrix} 0 & 0 & 2 & 0 & 2 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 2 \\ 0 & 1 & 2 & 0 & 0 \end{bmatrix}$$

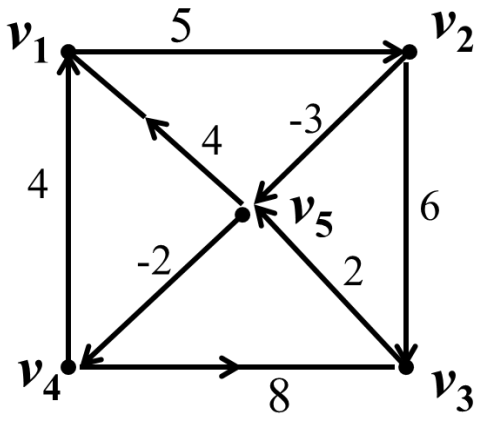


$$D^{(3)} = \begin{bmatrix} 0 & 5 & 11 & \infty & 2 \\ \infty & 0 & 6 & \infty & -3 \\ \infty & \infty & 0 & \infty & 2 \\ 4 & 9 & 8 & 0 & 6 \\ 4 & 9 & 15 & -2 & 0 \end{bmatrix}$$

$$D^{(4)} = \begin{bmatrix} 0 & 5 & 11 & \infty & 2 \\ \infty & 0 & 6 & \infty & -3 \\ \infty & \infty & 0 & \infty & 2 \\ 4 & 9 & 8 & 0 & 6 \\ 2 & 7 & 6 & -2 & 0 \end{bmatrix}$$

$k = 4$

$$P = \begin{bmatrix} 0 & 0 & 2 & 0 & 2 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 2 \\ 4 & 4 & 4 & 0 & 0 \end{bmatrix}$$



$$D^{(4)} = \begin{bmatrix} 0 & 5 & 11 & \infty & 2 \\ \infty & 0 & 6 & \infty & -3 \\ \infty & \infty & 0 & \infty & 2 \\ 4 & 9 & 8 & 0 & 6 \\ 2 & 7 & 6 & -2 & 0 \end{bmatrix} \quad D^{(5)} = \begin{bmatrix} 0 & 5 & 8 & 0 & 2 \\ -1 & 0 & 3 & -5 & -3 \\ 4 & 9 & 0 & 0 & 2 \\ 4 & 9 & 8 & 0 & 6 \\ 2 & 7 & 6 & -2 & 0 \end{bmatrix}$$

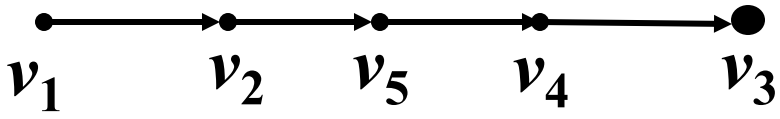
$k = 5$

$$P = \begin{bmatrix} 0 & 0 & 5 & 5 & 2 \\ 5 & 0 & 5 & 5 & 0 \\ 5 & 5 & 0 & 5 & 0 \\ 0 & 1 & 0 & 0 & 2 \\ 4 & 4 & 4 & 0 & 0 \end{bmatrix}$$

$P_{13} = 5,$

$P_{15} = 2,$ $P_{53} = 4,$

$P_{12} = 0,$ $P_{25} = 0,$ $P_{54} = 0,$ $P_{43} = 0$



从v_1到v_3的最短路长度 $D^{(5)}[1,3] = 8$。

Floyd算法

Floyd：使用两个D矩阵实现

1. $D \leftarrow W$
2. $P \leftarrow 0$
3. for $k \leftarrow 1$ to n
   // Computing D' from D
4. do for $i \leftarrow 1$ to n
5. do for $j \leftarrow 1$ to n
6. if $(D[i,j] > D[i,k] + D[k,j])$
7. then $D'[i,j] \leftarrow D[i,k] + D[k,j]$
8. $P[i,j] \leftarrow k$;
9. else $D'[i,j] \leftarrow D[i,j]$
10. *Move D' to D.*

Floyd算法

Floyd： 使用一个D矩阵实现

1. $D \leftarrow W$
2. $P \leftarrow 0$
3. for $k \leftarrow 1$ to n do
4. for $i \leftarrow 1$ to n do
5. for $j \leftarrow 1$ to n do
6. if $(D[i,j] > D[i,k] + D[k,j])$
7. then $D[i,j] \leftarrow D[i,k] + D[k,j]$
8. $P[i,j] \leftarrow k$;

为什么一个D矩阵也可以？不需要区分D^{k-1}和D^k？
因为D^{k-1}[i,k]和D^k[i,k]没有区别，D^{k-1}[k,j]和D^k[k,j]没有区别：v_k作为起始点或者终点，不需要考虑。
即：对于同一个k，任意的i和j，D[k,j]和D[i,k]不会更新。

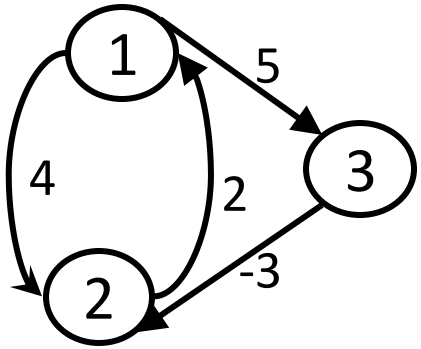
Floyd算法

- 打印出从q到r的路径

```
path(index q, r)
    if (P[ q, r ]!=0)
            path(q, P[q, r])
            println( “v”+ P[q, r])
            path(P[q, r], r)
            return;
    else return  //no intermediate nodes
```

P =

	1	2	3
1	0	3	0
2	0	0	1
3	2	0	0



9.6 网络流问题

网络流问题

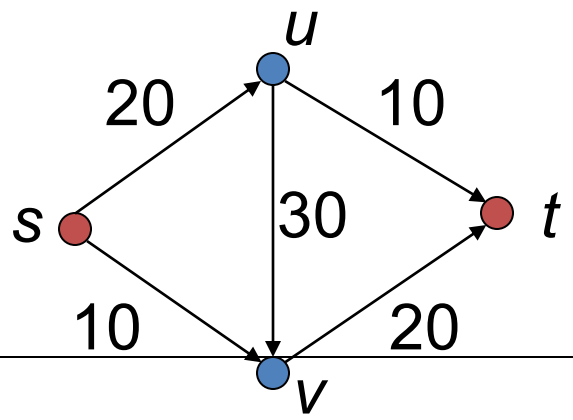
- 我们可以将一个有向图看做是一个“流网络”并使用它回答关于物料流动方面的问题。
  - 比如一种物料从产生它的源结点经过一个系统，流向消耗该物料的汇点这样一个过程。源结点以稳定速率生产物料，汇点以同样速率消耗物料。
  - 类似实际应用包括：液体在管道中的流动、装配线上部件的流动、电网中电流的流动和通信网中信息的流动。
  - 可以把流网络中每条有向边看作是物料的一个流通通道，每条通道有限定的容量，是物料流经该通道时的最大速率。流网络中的结点则是通道的连接点，除了源结点和终结（汇）点外，在连接点并不积累或者消耗，因此物料进入一个结点的速率必须与其离开的速率相等，这个性质叫“流量守恒”。
  - 在最大流问题中，我们希望在不违反任何容量限制和流量守恒的情况下，计算出从源结点运送物料到汇点的最大速率。

流网络

有向图 G = (V,E) 满足

- 每个有向边 e 有一个非负容量c_e
- 每条有向边没有反向边
- 有一个源结点 s 没有入边
- 有一个目标点（汇点） t (*target*) 没有出边
- 假设对于每个结点，都有一条经过该结点的从s到t的路径

u,v – 中间结点

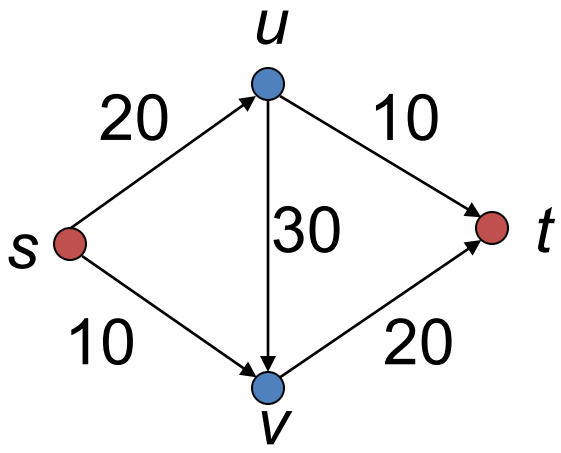


流网络

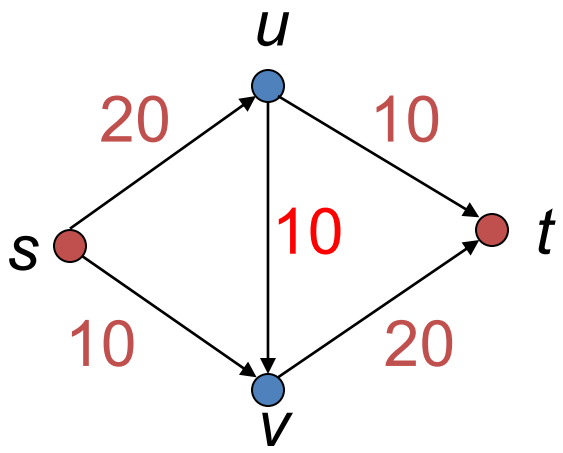
G = (V,E) 中的s-t 流是一个从E 到 $\mathbf{R}^+$的函数 f满足

- **容量限制条件:** 对每个e, $0 \le f(e) \le c_e$
- **流量守恒（保存条件）:** 对每个中间结点v, 有:
  $\sum_{e \text{ in } v} f(e) = \sum_{e \text{ out } v} f(e)$
- **源和汇满足:** $\sum_{e \text{ in } t} f(e) = \sum_{e \text{ out } s} f(e)$

网络:



流:

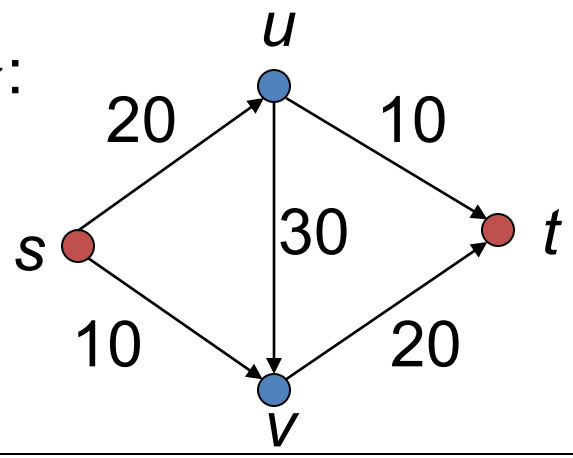


流网络

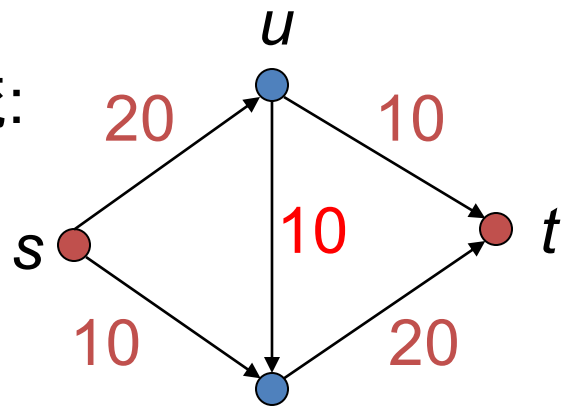
给定图G = (V,E)中 s-t 流 f 和任意结点集合 B

- $f^{in}(B) = \sum_{e \text{ in } B} f(e)$
- $f^{out}(B) = \sum_{e \text{ out } B} f(e)$
- 例子: $f^{in}(u,v) = f^{out}(u,v) = 30$
- 性质: $f^{in}(t) = f^{out}(s)$

网络:



流:



流网络

最大流问题

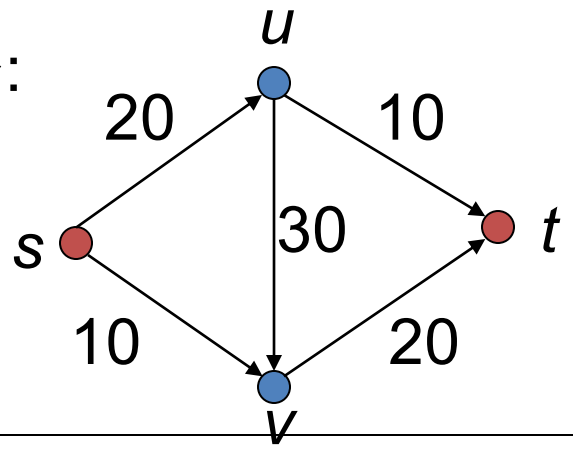
- 对于给定的图G = (V,E)，最大的$f^{in}(t)$是多少？
- 如何高效地计算?

注意：满足“容量限制”和“流量守恒”两个条件的前提下。

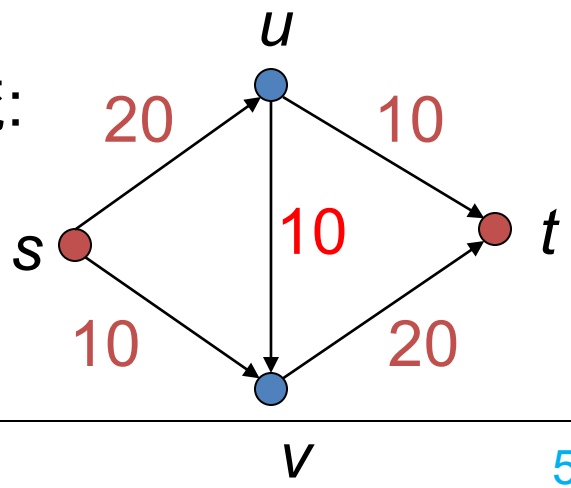
假设: 容量都是正数.

例: $f^{in}(t) = f^{out}(s) = 30$

网络:



流:



Ford-Fulkerson方法

- 基本思想：循环增加流的值
  - 初始，对于所有的边e，初始化流f(e)=0
  - 每一次迭代中，增加图中的流值，方法：在当前的余图（残存网络）中寻找一条增广路径，即符合条件的新流路径，从而增加流值
  - 不断迭代，直到不存在增广路径为止，即流值已经最大化

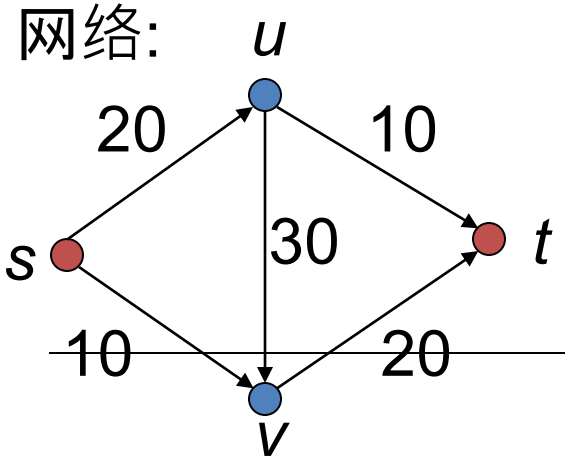
余图（残存网络）

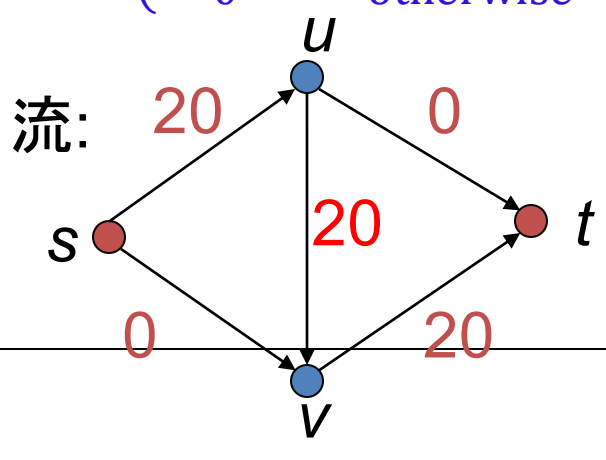
- 余图由那些仍有空间对流量进行调整的边构成

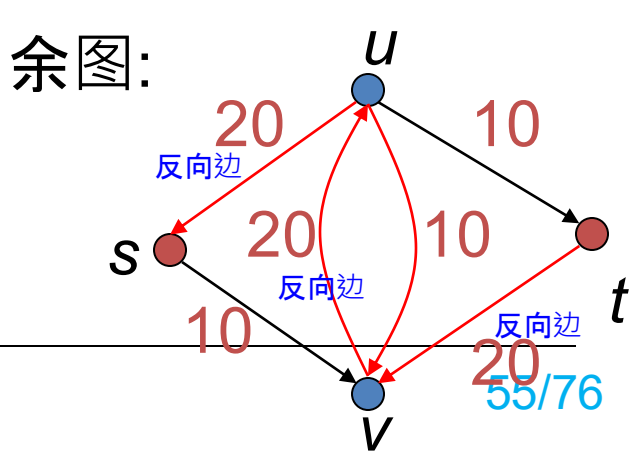
给定图G中的流f，余图 G_f 定义如下：

- 同样的结点：同样的中间结点和s, t
- 对于每条边 e 满足 $c_e > f(e)$ 赋给权重 $c_e - f(e)$ (仍有剩余容量)
- 对于每条边 $e = (u,v)$ 给其逆向边(v,u)赋给权重 $f(e)$ (容许的最大反向容量)
  - 算法对流量进行操作调整的目标是增加总流量，因此算法可能对某些特定边的流量进行缩减，以便于宏观调整，增加总的流量。为了表示边$e=(u,v)$上正流量$f(e)$的缩减，我们将其逆向边(v,u)加入到余图G_f中，并将剩余容量设置为$f(e)$，也就是说，一条边所能允许的反向流量最多将其正向流量抵消。
  - 余图中的这些反向边允许算法将已经发送出去的流量发送回去。

$$\text{残存容量} c_f(u,v) = \begin{cases} c_e - f(e) & if\ (u,v) \in E \\ f(e) & if\ (v,u) \in E \\ 0 & \text{otherwise} \end{cases}$$





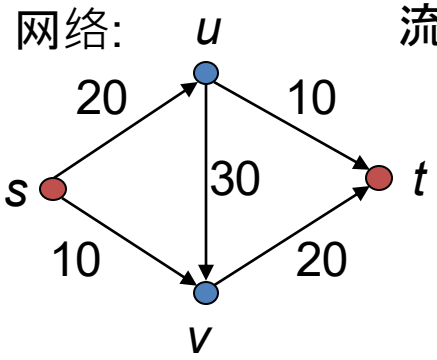


增广路径

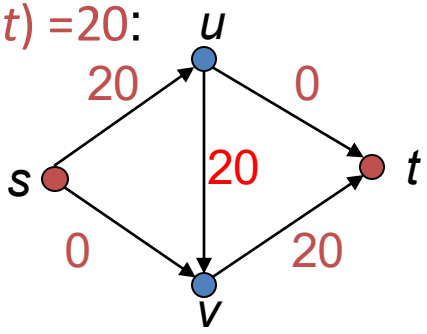
给定图G中的流f, 及其对应的余图G_f

1. 找到余图中的一条新流，该流通过一条没有重复结点的路径，并且值和该路径上的**最小**容量相等(增广路径)，该值为沿着增广路径能够为每条边增加的流量的最大值
2. 沿着路径更新余图

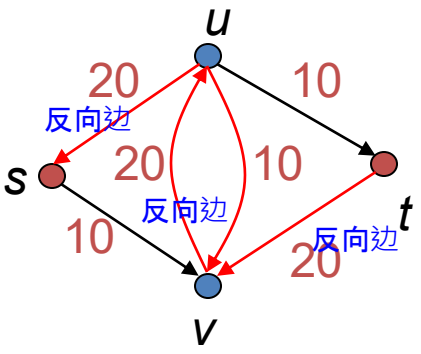
网络:



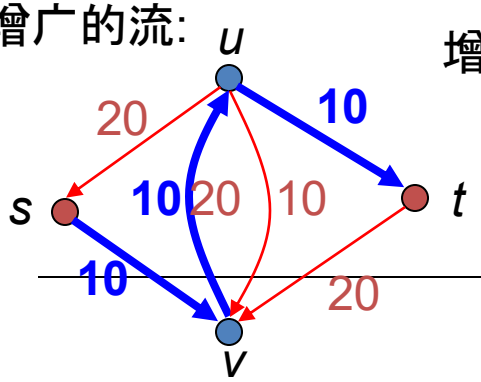
流$f^{in}(t)$ =20:



余图:

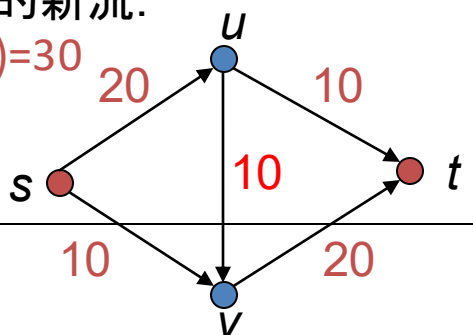


增广的流:

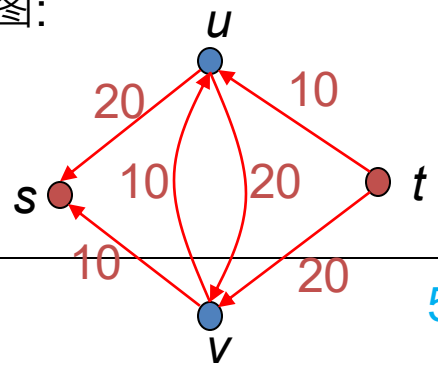


增广后的新流:

$f^{in}(t)$=30



新的余图:



Ford-Fulkerson方法

- 基本思想：循环增加流的值

```
Ford-Fulkerson(G,s,t)
1. For each edge(u,v) in G.E Do
2.     f(u,v)=0
3. While there exists a path p from s to t
in Gf Do //如果余图中存在s-t 的增广路径 P
4.     cf(p) ← min{cf(u,v):(u,v) is in p}
5.     For each edge (u,v) in p Do
6.         If (u,v) in G.E Do
7.             f(u,v) ← f(u,v)+cf(p)
8.         Else f(v,u) ← f(v,u)-cf(p)
```

对于所有e初始化 $f(e) = 0$

找到路径p的最小容量

沿着路径p增广f(),即修改流

算法正确性证明参见书26.2节，不做要求

示例

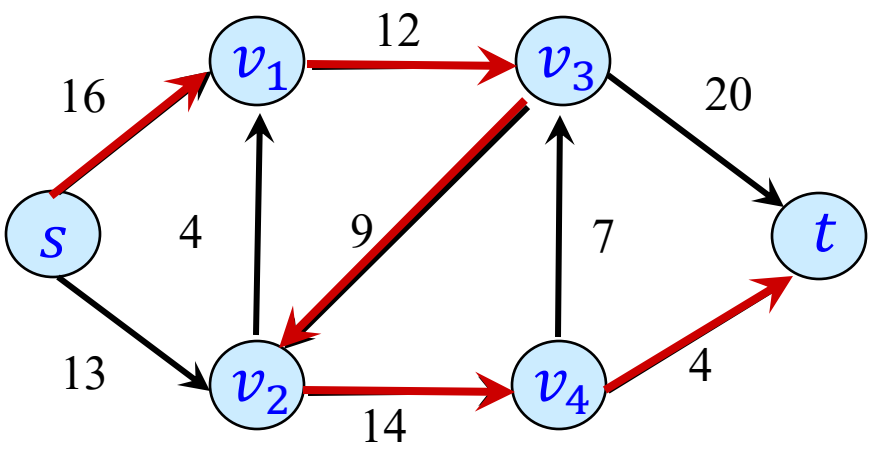
初始：没有流，
余图=输入网络图

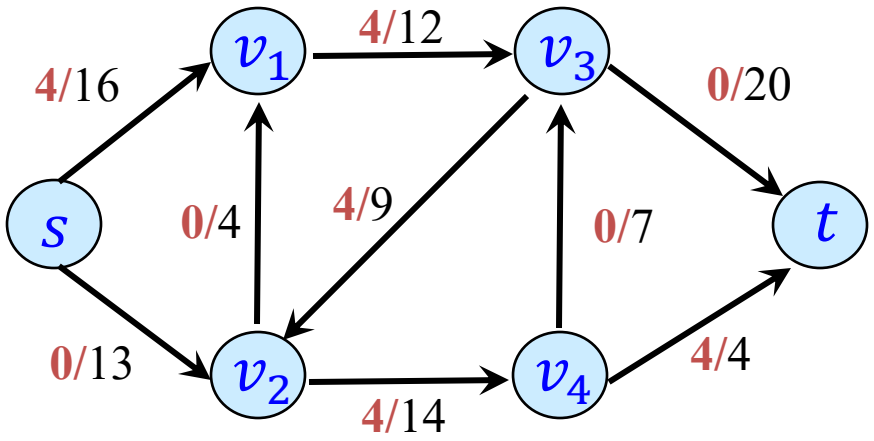
红线代表增广路径

余图

流

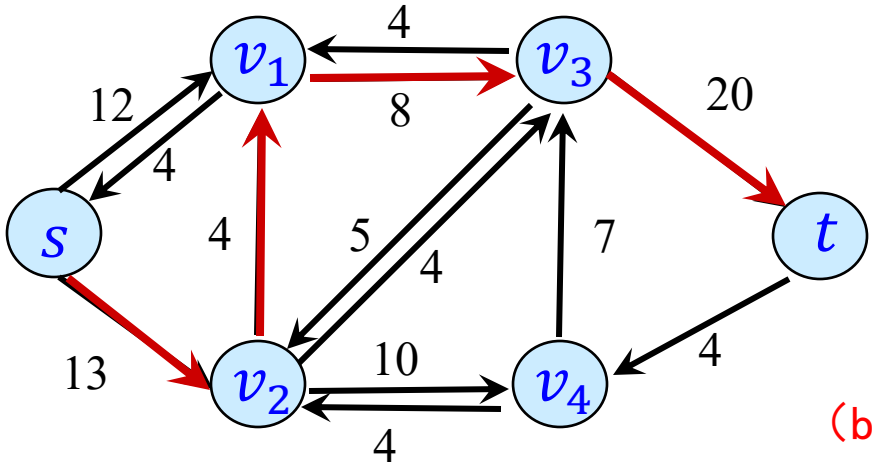
流/容量



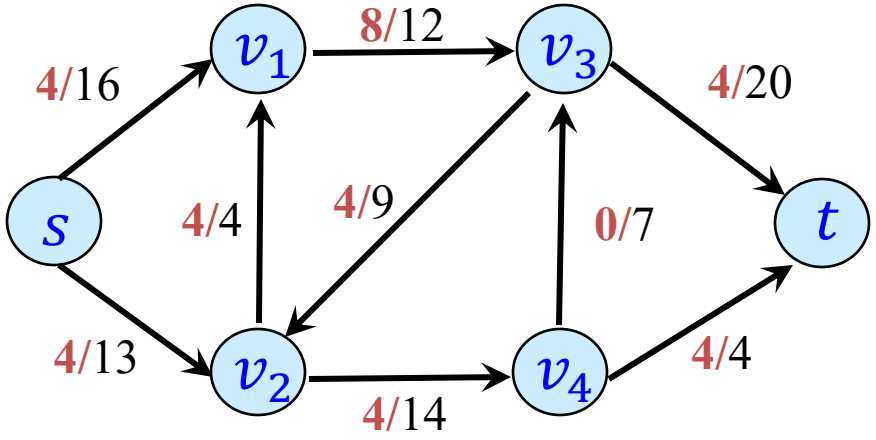


（a）输入网络

$c_f(p) = \min\{16,12,9,14,4\} = 4$



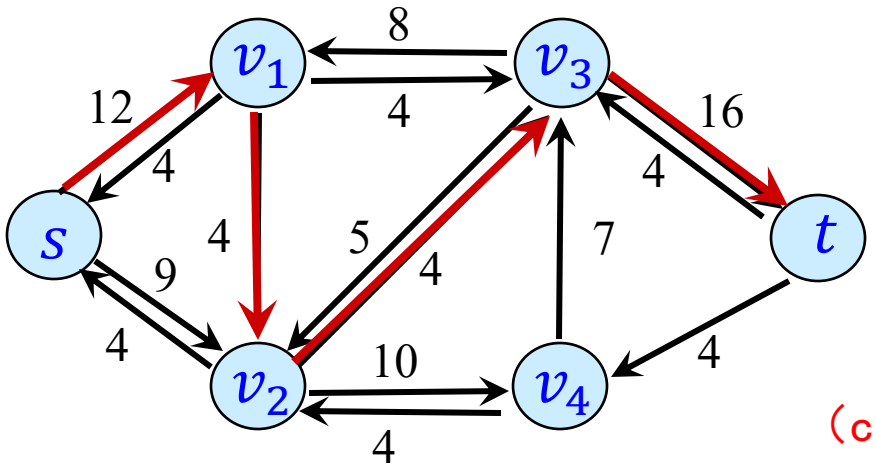
（b）



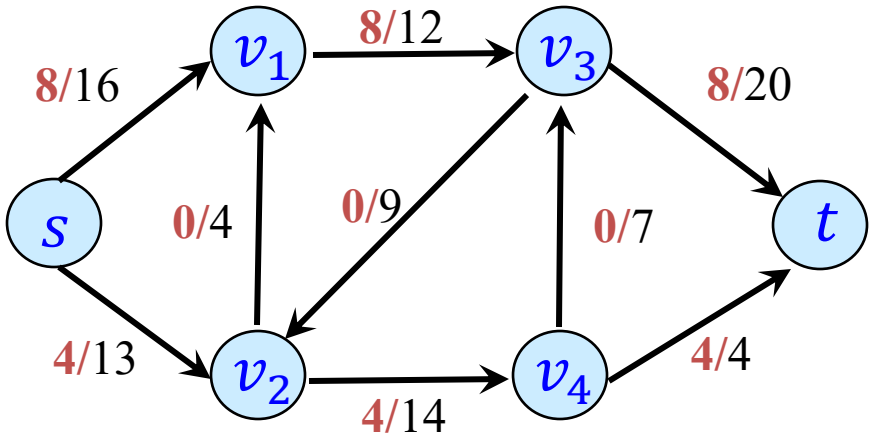
$c_f(p) = min\{13,4,8,20\} = 4$

示例

余图

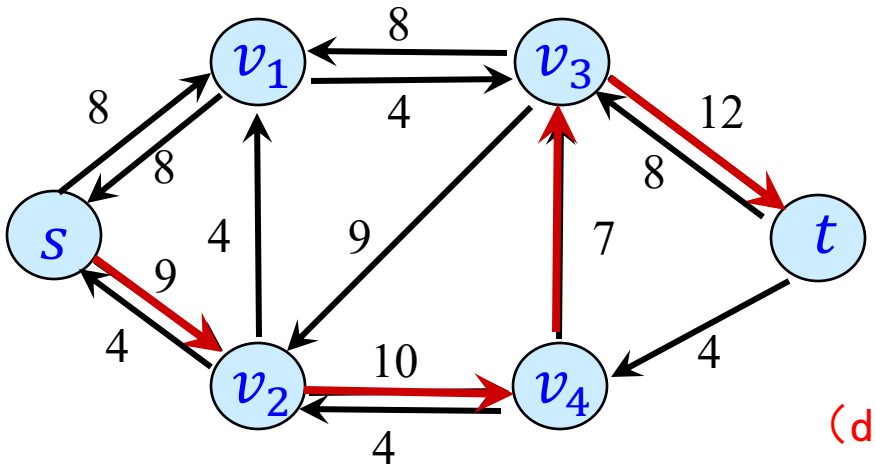


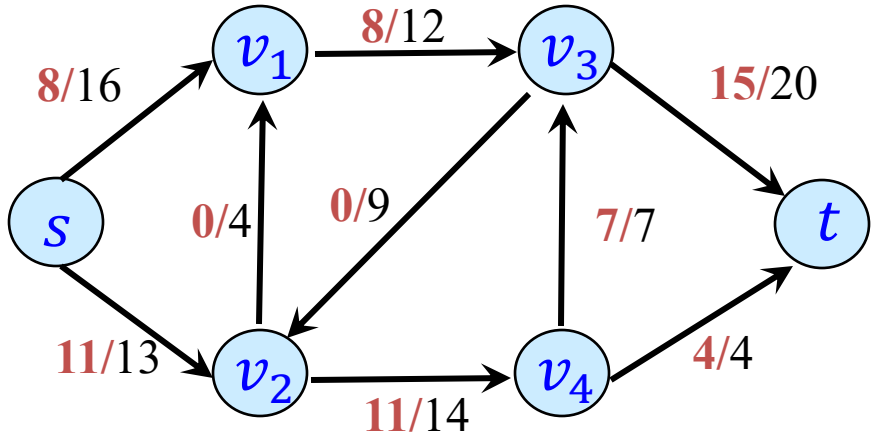
流



(c)

$c_f(p) = min\{12,4,4,16\} = 4$



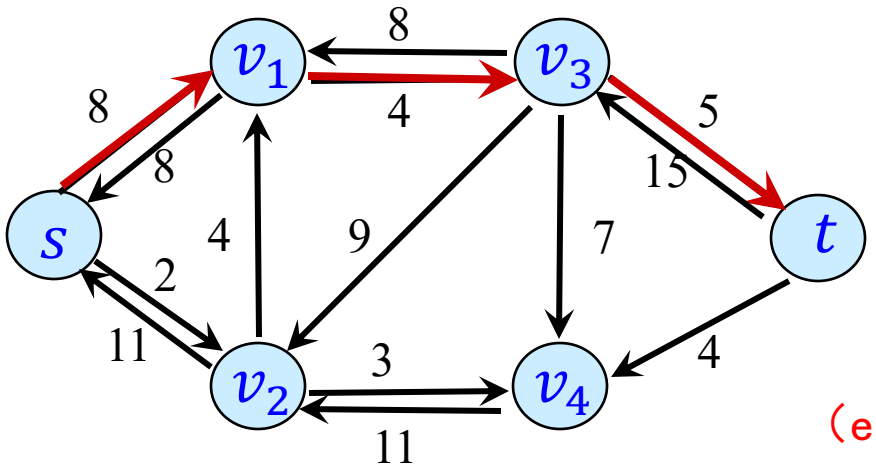


(d)

$c_f(p) = min\{9,10,7,12\} = 7$

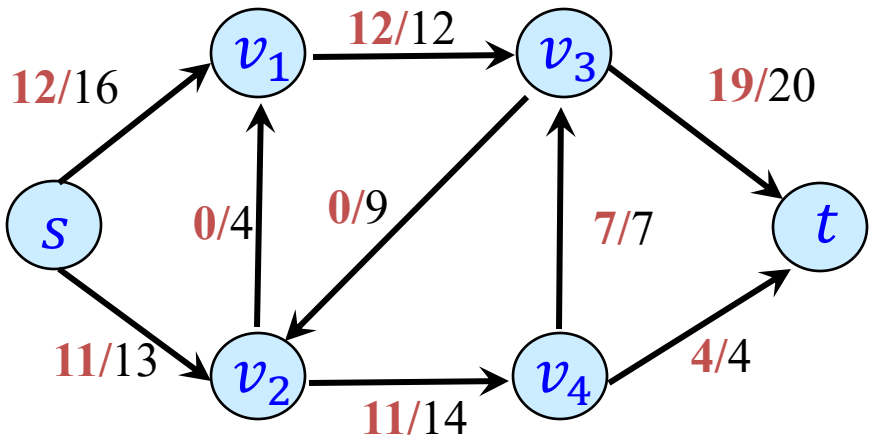
示例

余图

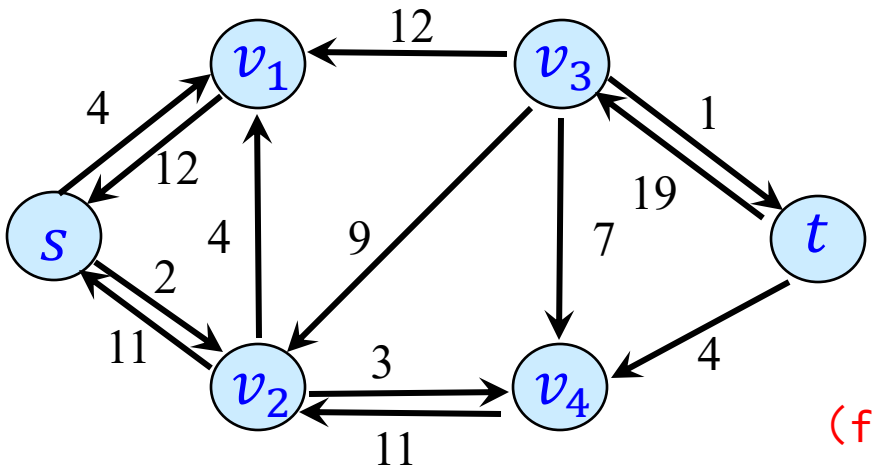


(e)

流



$$c_f(p) = min\{8,4,12\} = 4$$



(f)

没有增广路径

最大流是$12 + 11 = 23$

注意：从流图中计算最大流

Ford-Fulkerson方法

- 算法分析
  - 如何寻找增广路径是关键
    - 采用深度优先搜索或者广度优先搜索寻找增广路径：O(V+E')=O(E),其中E'为当时余图的边数, E为原图边数
  - 可终止性：最大流问题的容量常常为整数，如果不是，乘以一个系数转换为整数
    - 流量值在每次迭代中至少增加1，那么最多经过C 轮迭代算法停止，其中C 是最大流的值
  - 时间复杂性：O（EC）

```
Ford-Fulkerson(G,s,t)
1. For each edge(u,v) in G.E Do
2.       f(u,v)=0
3. While there exists a path p from s to t in Gf
Do
4.       cf(p)←min{cf(u,v):(u,v) is in p}
5.       For each edge (u,v) in p Do
6.          If (u,v) in G.E Do
7.             f(u,v)←f(u,v)+cf(p)
8.          Else f(v,u)←f(v,u)-cf(p)
```

9.7 匹配问题

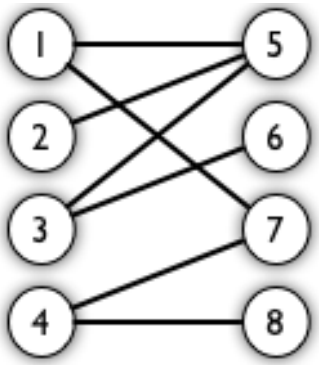
匹配

- 给定一个无向图G=(V,E)，一个匹配是一个边的子集 $M \subseteq E$，使得对于所有结点$v \in V$，M中最多有一条边与结点v相连，即M中任意两条边都没有公共顶点。M中的边称为匹配边，M中的点称为匹配点。不在M中的边和点则称为未匹配边，未匹配点。
- 极大匹配
  - 不存在 $e \notin M$ 满足 $M \cup \{e\}$ 也是匹配
- 最大匹配
  - $|M|$ 最大的匹配：对于任意匹配M’，有$|M| \geq |M'|$
- 完美匹配
  - $|M| = |V|/2$：每个结点都是M中边的顶点

二分图匹配

二分图

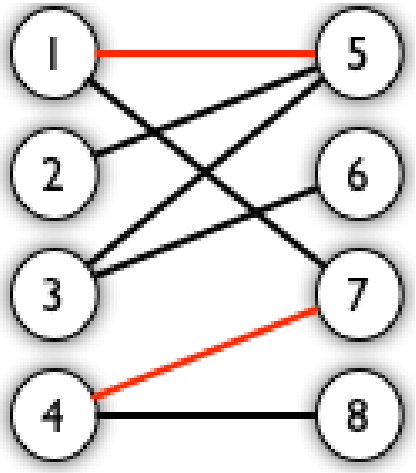
- 二分图又称作二部图，是图论中的一种特殊模型。 设G=(V,E)是一个无向图，如果顶点V可分割为两个互不相交的子集(A,B)，并且图中的每条边（i，j）所关联的两个顶点i和j分别属于这两个不同的顶点集(i∈A,j∈B)，则称图G为一个二分图



- 在二分图中寻找最大匹配有着许多实际应用：比如把一个机器集合L和同时执行的任务集合R相匹配。E中有边(u,v)说明一台特定的机器u∈L能够完成一项特定的任务v∈R. 最大匹配能够让尽可能多的机器运行起来。

交替路和增广路

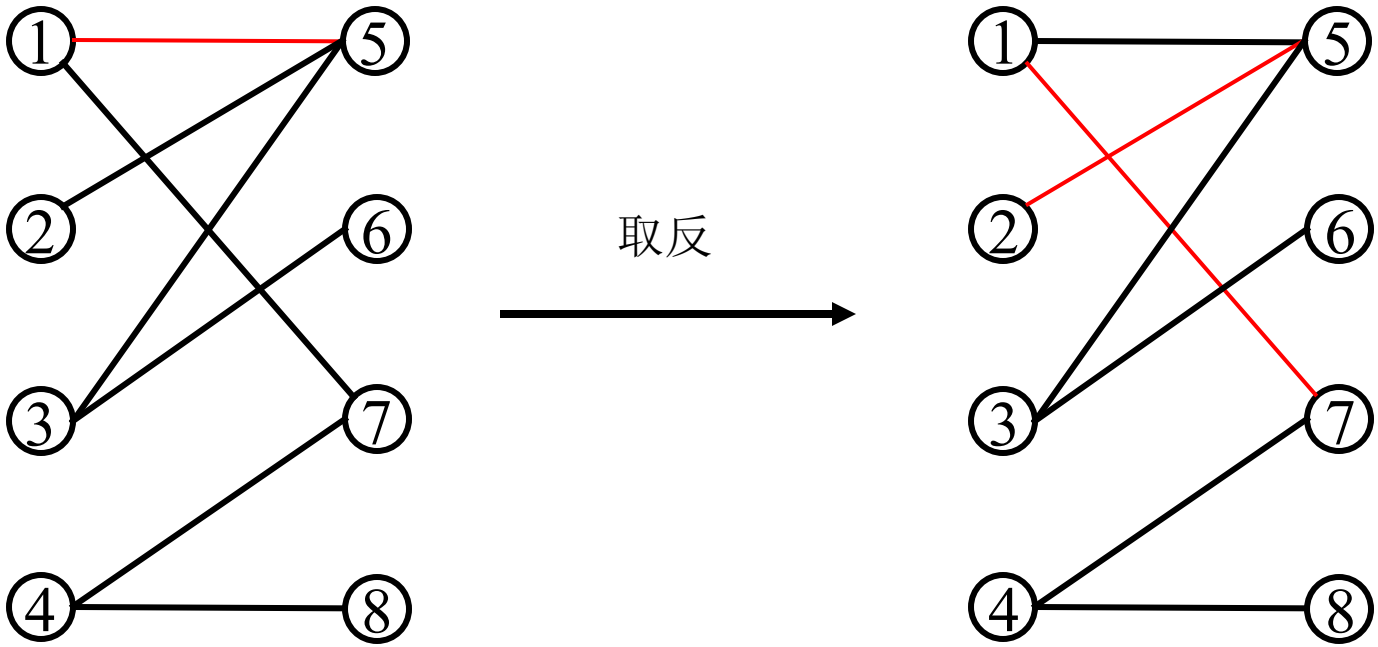
- 交替路：从一个未匹配点出发，依次经过非匹配边、匹配边、非匹配边…形成的路径叫交替路。
- 增广路：从一个未匹配点出发，走交替路，如果终点为另一个未匹配点（出发的点不算），则这条交替路称为增广路。



$8 \rightarrow 4 \rightarrow 7 \rightarrow 1 \rightarrow 5 \rightarrow 2$

关于增广路的推论

- 增广路的路径长度必定为奇数，第一条边和最后一条边都不属于M。
- 增广路经过取反操作，可以得到一个更大的匹配。
- M为二分图G的最大匹配当且仅当不存在相对于M的增广路径。



增广路：2→5→1→7

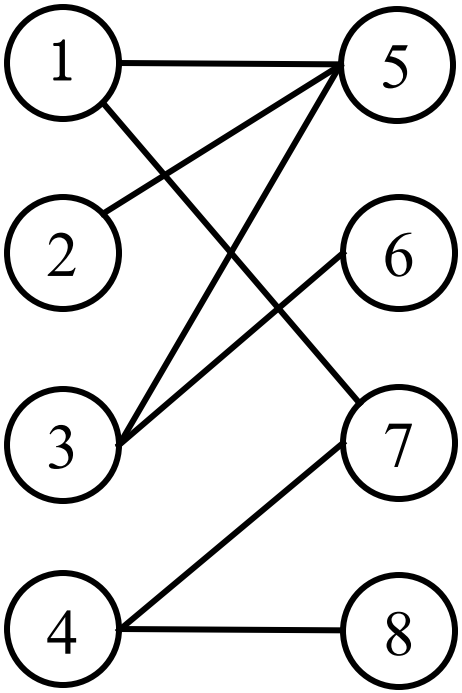
注意：只含有一条不属于M的边也是增广路

匈牙利算法

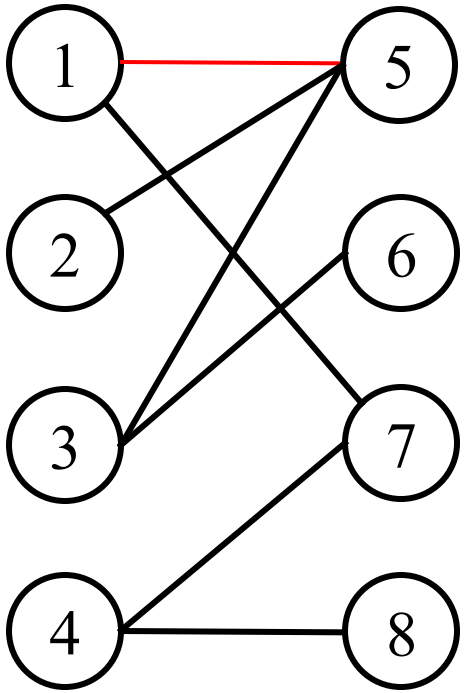
- 求二分图的最大匹配。
- 基本步骤：
- 1. 置M为空；
- 2. 找出一条增广路径P，通过取反操作，得到更大的匹配。
- 3. 重复步骤2，直到找不出增广路为止.

书26.3提供了另外一种基于Ford-Fulkerson的二分图最大匹配解法

例子



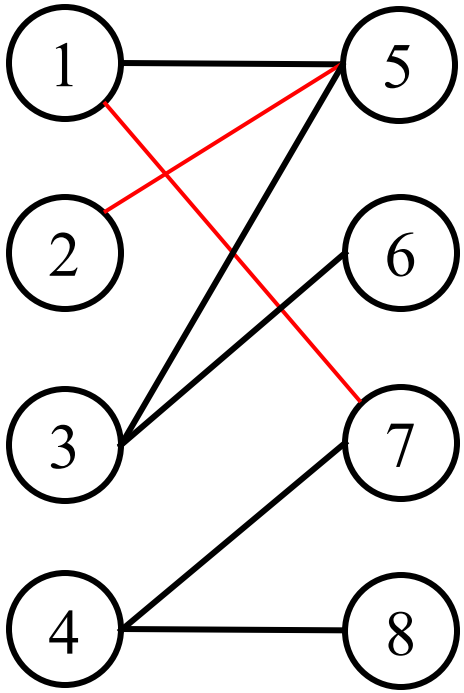
增广路:1→5



M={(1,5)}

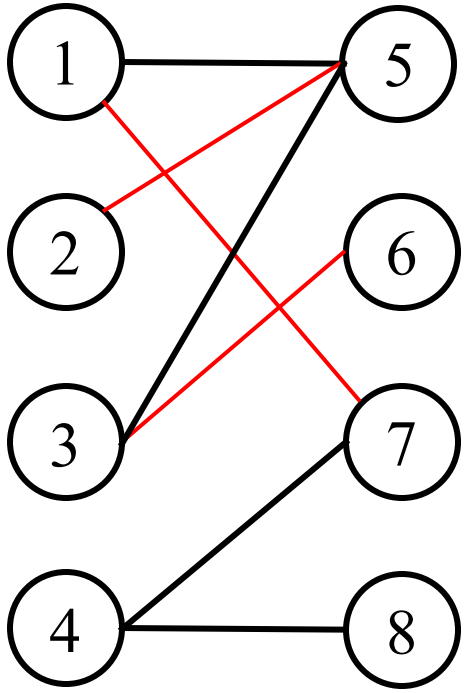
增广路：2→5→1→7

例子



M={(2,5),(1,7)}

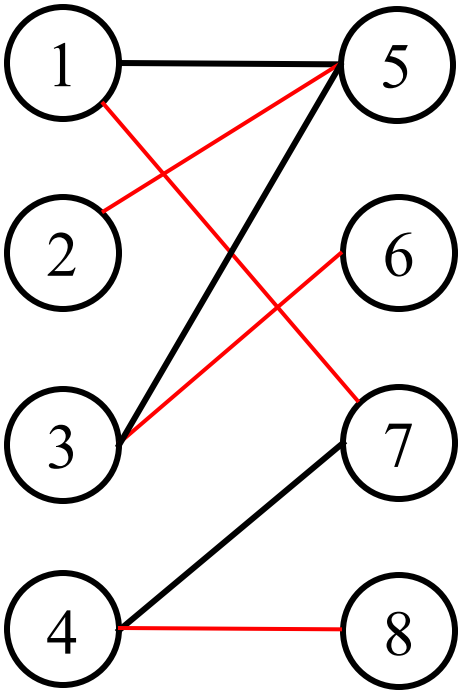
增广路：3→6



M={(2,5),(1,7),(3,6)}

增广路：4→8

例子



M={(2,5),(1,7),(3,6),(4,8)}

此时图中已无增广路，故该二分图的最大匹配为4

应用

现要给4个工人A, B, C, D分配任务，每个工人可完成不同的任务，但最多只能接受一个任务。共有4个任务，每个任务也只能分配给一个工人，问最多可以分配多少个任务给工人？

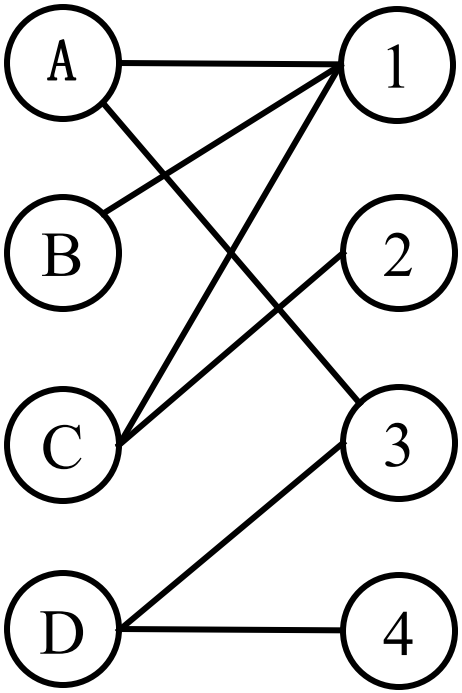
	任务1	任务2	任务3	任务4
A	1	0	1	0
B	1	0	0	0
C	1	1	0	0
D	0	0	1	1

1代表能完成，0代表不能完成

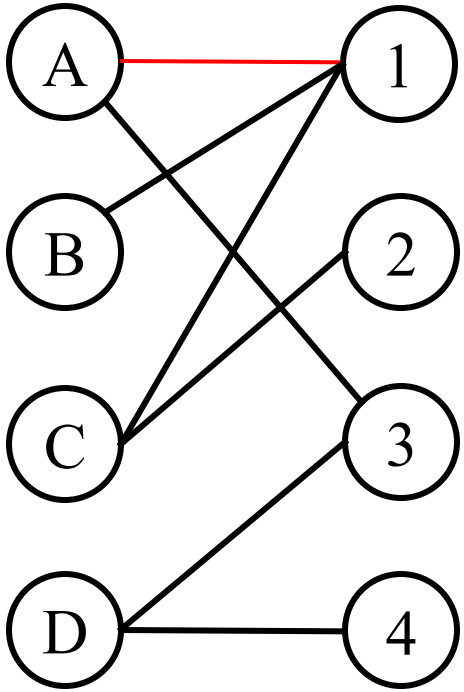
应用

- 分析：数学建模
  - 该问题可以转化为一个图模型。工人和任务可以看作两个不相交的点集合，将工人和他能完成的任务相连(比如工人A能完成任务1，则图中含有边A1)，因此图中的每条边的两个顶点分别落在两个集合里，所以此图是一个二分图。
  - 又因为“每个工人只能接受一个任务，每个任务只能分配给一个工人”，则意味着我们要寻找一个边集合，使得任意两条边没有公共顶点，这就是该图的匹配。
  - “最多可以分配多少个任务”就是要寻找最大匹配，可以用匈牙利算法求解。

应用



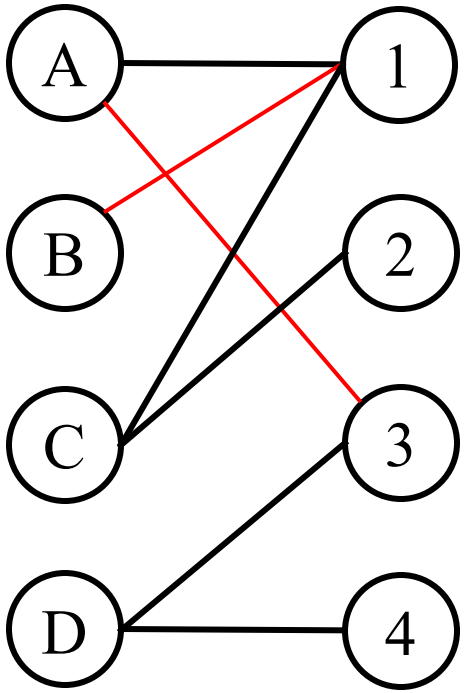
增广路:A→1



M={(A,1)}

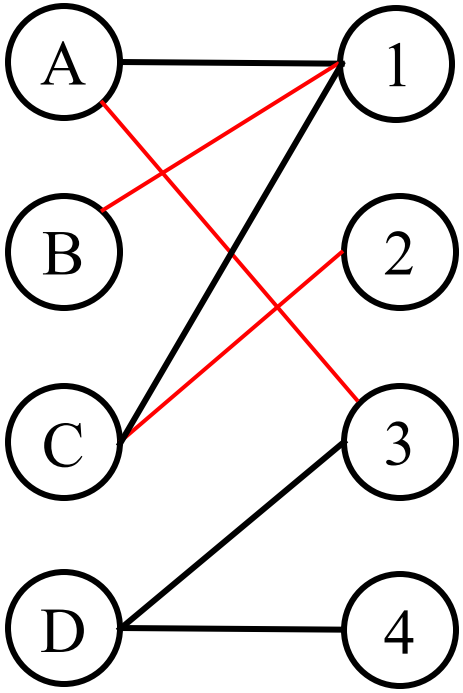
增广路：B→1→A→3

应用



M={(A,3),(B,1)}

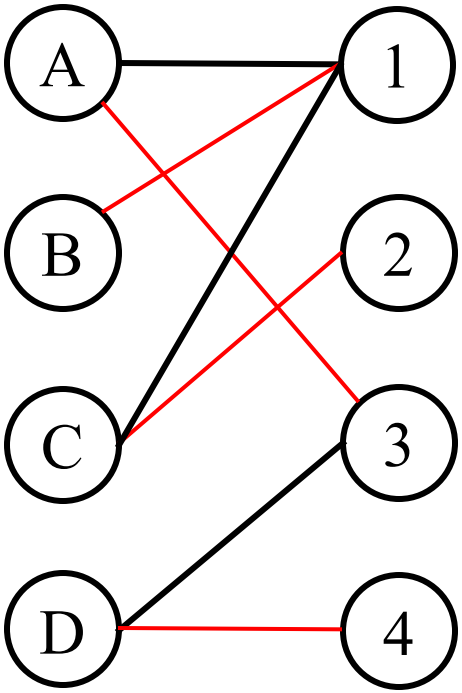
增广路：C→2



M ={(A,3),(B,1),(C,2)}

增广路：D→4

应用



M ={(A,3),(B,1),(C,2),(D,4)}

此时图中已无增广路，故该二分图的最大匹配为4。
所以最多能分配4个任务。