7.6 最短路径

7.6.1 最短路径的概念

7.6.1 最短路径的概念

- 图中从A站点到B站点距离最短的路径；
- 图中从A站点到B站点最便宜的路径；
- 图中从A站点到B站点最快的路径；所谓最快，可以用经停站点最少的路径表示；

以上问题都可抽象为：

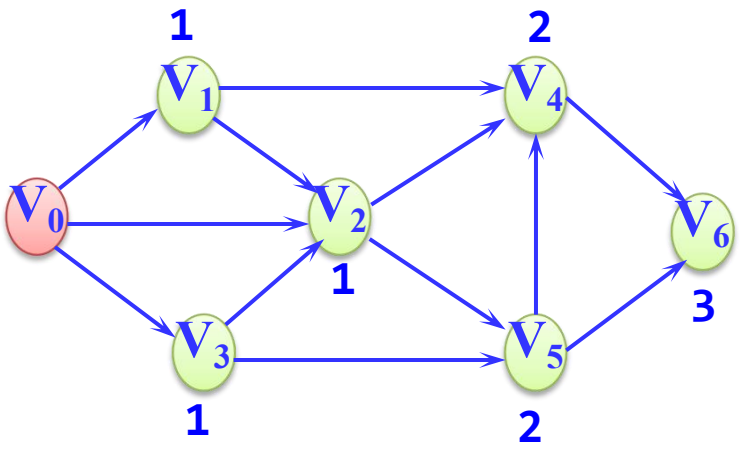
在网络中，求两个不同顶点之间的所有路径中，边的权值之和最小的路径，即为两顶点间的最短路径（Shortest Path）。

问题分类：

- 单源最短路径问题：从某固定源点出发，求其到所有其他顶点的最短路径；
  - （有向）无权图
  - （有向）有权图
- 多源最短路径问题：求图中任意两顶点间的最短路径。

无权图的单源最短路径算法

- 按照递增（非递减）的顺序找出到各个顶点的最短路径：



0：V_0
1：V_1，V_2和 V_3
2：V_4 和 V_5
3：V_6

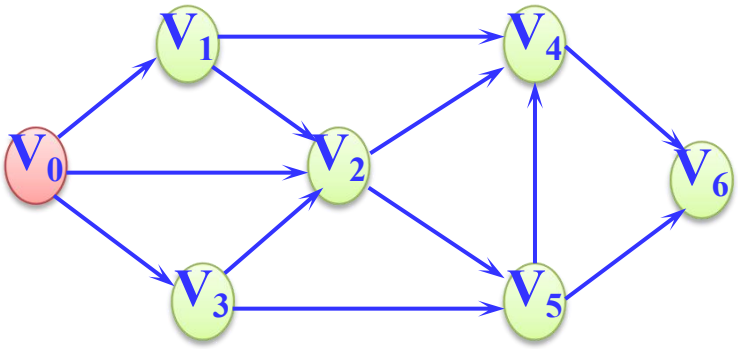
```
from collections import deque
MAXV=100                                    #全局变量，表示最多顶点个数
visited=[0]*MAXV
def BFS(G,v):                               #邻接表G中顶点v出发广度优先遍历
  qu=deque()                                #将双端队列作为普通队列qu
  print(v,end=" ")                          #访问顶点v
  visited[v]=1                              #置已访问标记
  qu.append(v)                              #v进队
  while len(qu)>0:                          #队不空循环
    v=qu.popleft()                          #出队顶点v
    for j in range(len(G.adjlist[v])):      #处理顶点v的所有出边
      w=G.adjlist[v][j].adjvex              #取顶点v的第j个出边邻接点w
      if visited[w]==0:                     #若w未访问
        print(w,end=" ")                    #访问顶点w
        visited[w]=1                        #置已访问标记
        qu.append(w)                        #w进队
```

dist[w]=s 到w 的最短距离； 可兼职visited[w]的功能！
dist[s]=0;
path[w]=s到w的路上，w的前驱顶点。这样即可得到最短路径！

数组初始化：
dist[w]=-1 对图中所有顶点w；非法距离；这样dist可兼职visited数组的功能；
path[w]=-1！或V

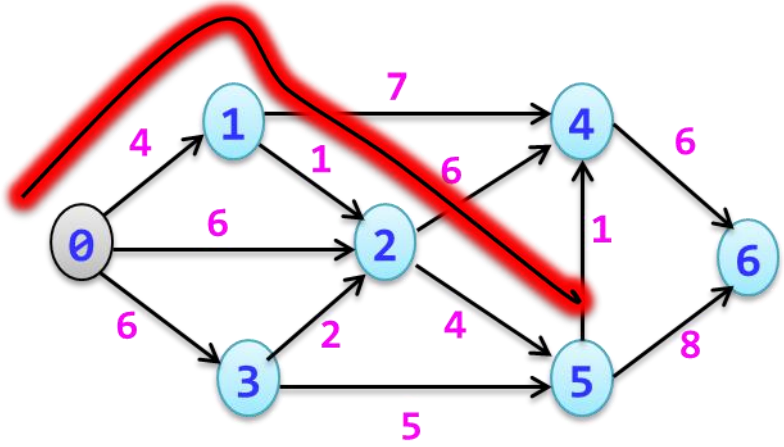
```
def BFS(G,v):                              #邻接表G中顶点v出发广度优先遍历
  qu=deque()                               #将双端队列作为普通队列qu
  qu.append(v)                             #v进队
  while len(qu)>0:                         #队不空循环
    v=qu.popleft()                         #出队顶点v
    for j in range(len(G.adjlist[v])):     #处理顶点v的所有出边
      w=G.adjlist[v][j].adjvex             #取顶点v的第j个出边邻接点w
      if dist[w]==-1:                      #w尚未访问
        dist[w]=dist[v]+1                  #更新源点到w的最短距离！
        path[w]=v                          #记录w的前驱；
        qu.append(w)                       #w进队
```

T=O(|V|+|E|)



	0	1	2	3	4	5	6
dist	0	-1	-1	-1	-1	-1	-1
path	-1	-1	-1	-1	-1	-1	-1

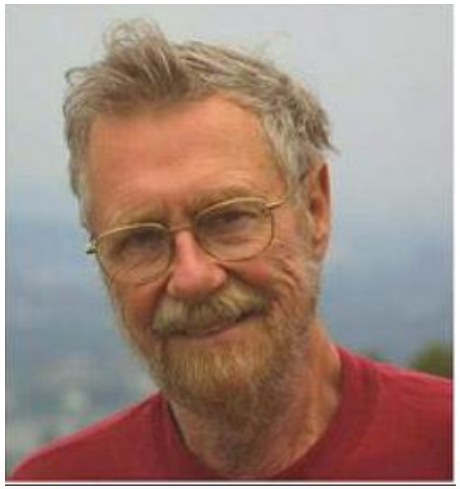
带权图的最短路径算法



0→5 的距离？

采用广度优先遍历可以求最短路径？ 不适合

7.6.2 狄克斯特拉算法



Edsger Wybe Dijkstra

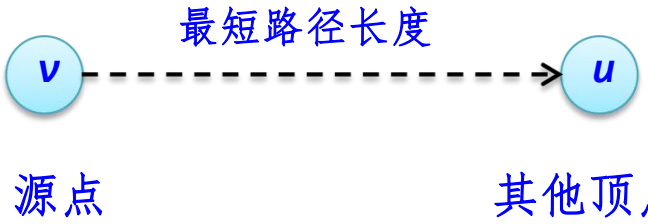
1930年5月11日~2002年8月6日

- 提出“goto有害论”
- 提出信号量和PV原语
- 解决了“哲学家聚餐”问题
- Dijkstra最短路径算法和银行家算法的创造者
- 第一个Algol 60编译器的设计者和实现者
- THE操作系统的设计者和开发者
- 1972年获得图灵奖
- 与D.E.Knuth并称为我们这个时代**最伟大的计算机科学家**的人，与癌症抗争多年，于2002年8月6日在荷兰Nuenen自己的家中去世，享年72岁。

真正统治世界的十大算法

1. 归并排序，快速排序和堆排序
2. 傅立叶变换与快速傅立叶变换
3. Dijkstra算法
4. RSA算法（一种加密算法）
5. 安全哈希算法
6. 整数因式分解
7. 链接分析（Google的Page Rank算法）
8. 比例积分微分算法
9. 数据压缩算法（以哈夫曼算法为基础）
10. 随机数生成算法

1．狄克斯特拉算法过程

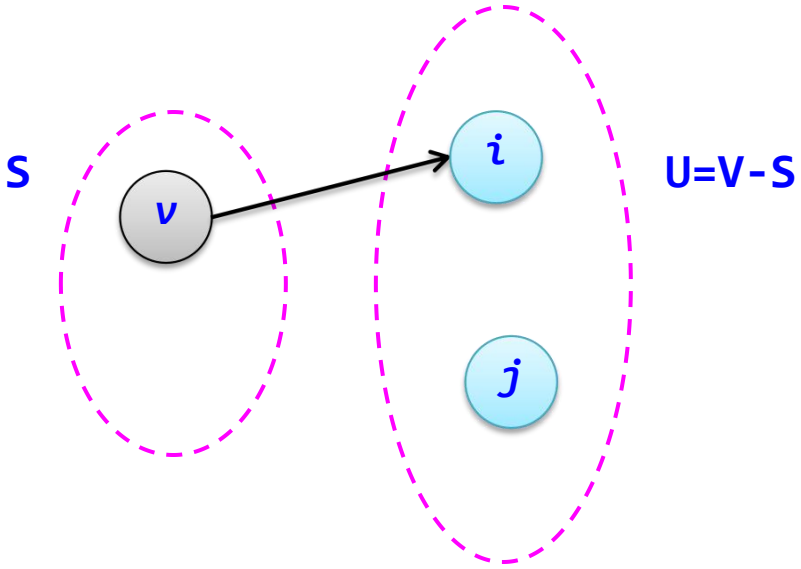


- 按递增（非递减）的顺序找出源点到各个顶点的最短路；（贪心算法）
- 真正的最短路必须只经过s中的顶点（为什么？）反正；
- 令S={源点+已经确定了最短路径的顶点v_i}；

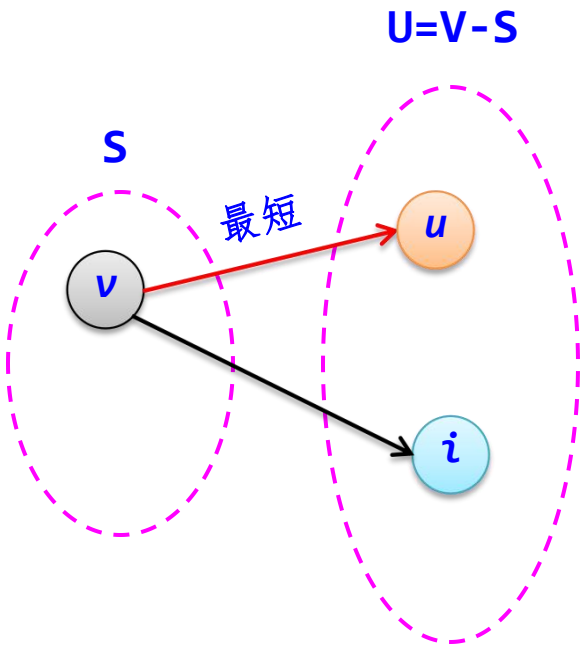
给定带权图G和源点v，Dijkstra算法的步骤如下：

（1）初始化：S={v}，顶点集U包含除v外的其他顶点。

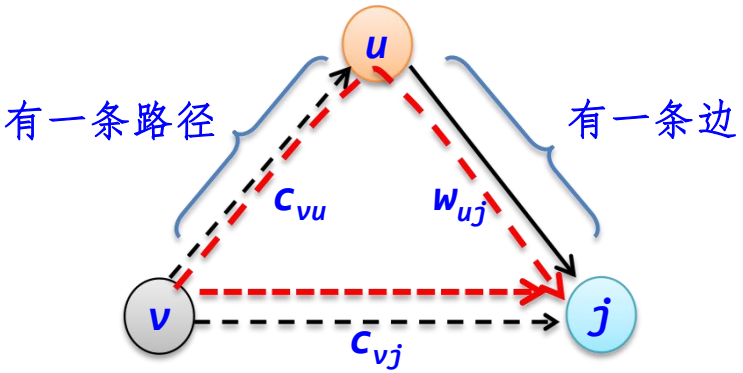
- 顶点v到自己的最短路径长度为0。
- 源点v到U中顶点i的最短路径长度为该边权值（v到i有边<v，i>）。
- 源点v到U中顶点i的最短路径长度为∞（若v到i没有边）。



（2）从U中选取一个顶点*u*，它是源点*v*到U中最短路径长度最小的顶点，然后把*u*加入S中（此时求出了*v*到*u*的最短路径长度）。



（3）以顶点u为新考虑的中间点，修改顶点u的出边邻接点j的最短路径长度，此时源点v到顶点j的最短路径有两条，即一条经过顶点u，一条不经过顶点u：



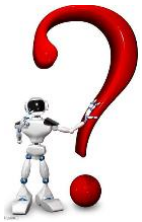
比较 ⟹ 求两条路径中求最小者

（4）重复步骤（2）和（3）直到S包含所有的顶点即U为空。

2．狄克斯特拉算法设计

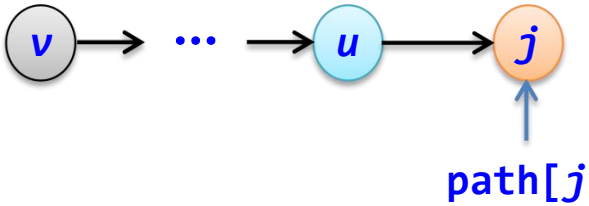
狄克斯特拉算法设计要点：

- 判断顶点i属于哪个集合，设置一个数组S，S[i]=1表示顶点i属于S集合，S[i]=0表示顶点i属于U集合。
- 保存最短路径长度：数组dist[0..n-1]，dist[i]用来保存从源点v到顶点i的最短路径长度。dist[i]的初值为<v，i>边上的权值，若顶点v到顶点i没有边，则权值定为∞。以后每考虑一个新的中间点u时，dist[i]的值可能被修改变小。
- 保存最短路径：数组path[0..n-1]，其中path[i]存放从源点v到顶点i的最短路径。



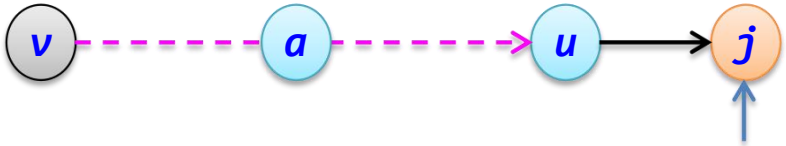
为什么能够用一个一维数组保存多条最短路径呢？

path[*j*]只保存源点*v*到顶点*j*的最短路径上顶点*j*的前驱顶点



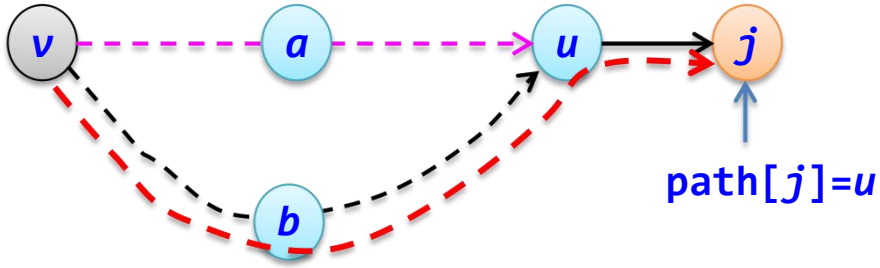
- 如果该路径恰好包含源点*v*到*u*的最短路径 ⇨ **OK**
- 如果不是 ⇨ ✕

证明



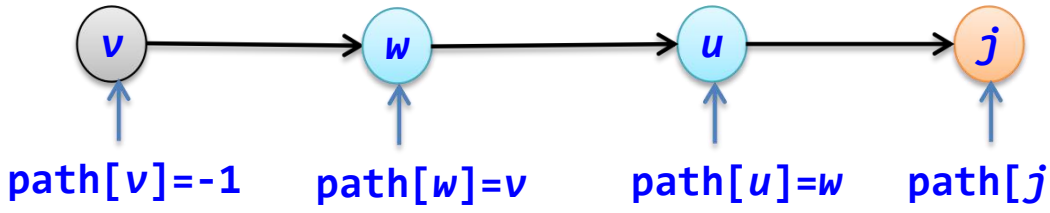
假设 $v \to a \to u \to j$ 是v到j的最短路径

但 $v \to a \to u$ 不是v到u的最短路径



该路径更短，与假设矛盾！

path表示

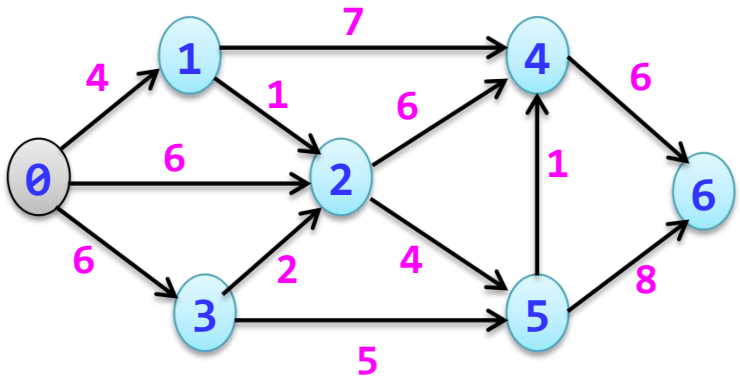


由path反推最短路径

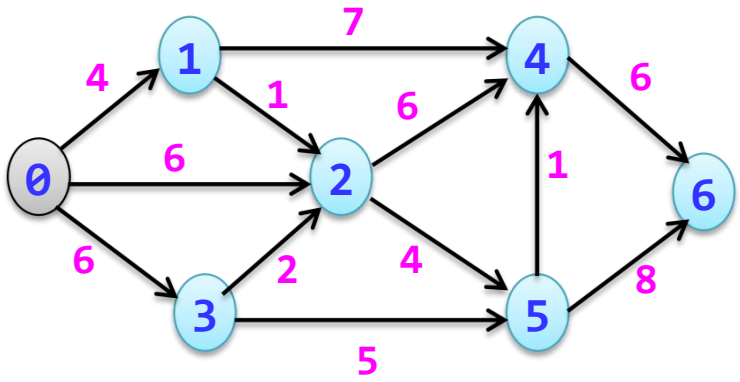
path[j]=u
path[u]=w
path[w]=v（到源点为止）

逆路径：j u w v
反向：v → w → u → j

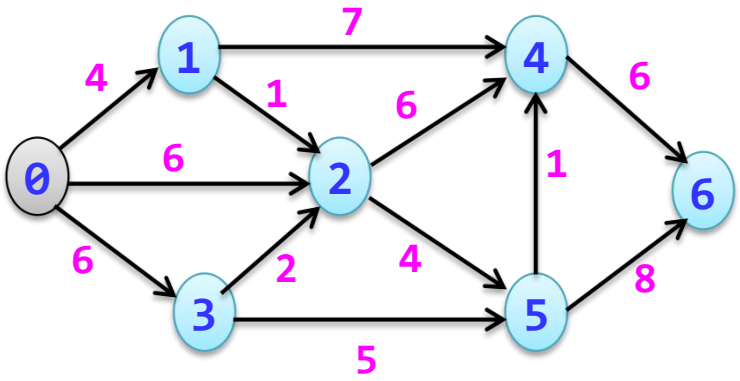
示例



S	U	dist[]	path[]
		0 1 2 3 4 5 6	0 1 2 3 4 5 6
{0}	{1,2,3,4,5,6}	{0, 4, 6, 6, ∞, ∞, ∞}	{0, 0, 0, 0, -1, -1, -1}
		最小的顶点：1	
{0,1}	{2,3,4,5,6}	{0, 4, 5, 6, 11, ∞, ∞}	{0, 0, 1, 0, 1, -1, -1}
		最小的顶点：2	
{0,1,2}	{3,4,5,6}	{0, 4, 5, 6, 11, 9, ∞}	{0, 0, 1, 0, 1, 2, -1}



S	U	dist[]	path[]
		0 1 2 3 4 5 6	0 1 2 3 4 5 6
{0,1,2}	{3,4,5,6}	{0, 4, 5, 6, 11, 9, ∞}	{0, 0, 1, 0, 1, 2, -1}
		最小的顶点：3	
{0,1,2,3}	{4,5,6}	{0, 4, 5, 6, 11, 9, ∞}	{0, 0, 1, 0, 1, 2, -1}
		最小的顶点：5	
{0,1,2,3,5}	{4,6}	{0, 4, 5, 6, 10, 9, 17}	{0, 0, 1, 0, 5, 2, 5}



S	U	dist[]	path[]
		0 1 2 3 4 5 6	0 1 2 3 4 5 6
{0,1,2,3,5}	{4,6}	{0, 4, 5, 6, 10, 9, 17}	{0, 0, 1, 0, 5, 2, 5}
		最小的顶点：4	
{0,1,2,3,5,4}	{6}	{0, 4, 5, 6, 10, 9, 16}	{0, 0, 1, 0, 5, 2, 4}
		最小的顶点：6	
{0,1,2,3,5,4,6}	{}	{0, 4, 5, 6, 10, 9, 16}	{0, 0, 1, 0, 5, 2, 4}

最终结果

利用dist和path求最短路径长度和最短路径

❶ 求0 ⇨ 6的最短路径长度：

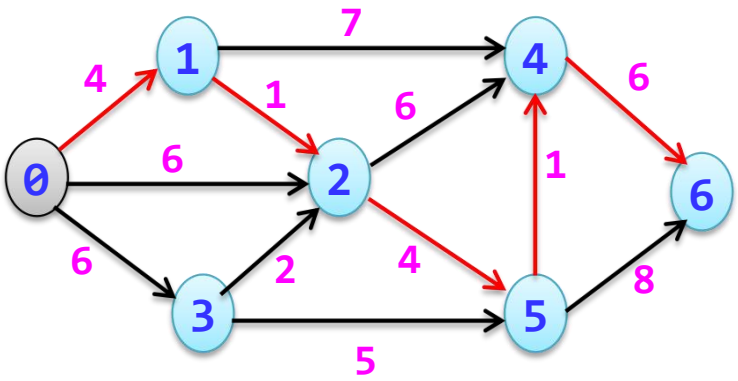
```
      0   1   2   3   4   5   6
dist={0,  4,  5,  6, 10,  9, 16}
```

从顶点0 ⇨6的最短路径长度为16

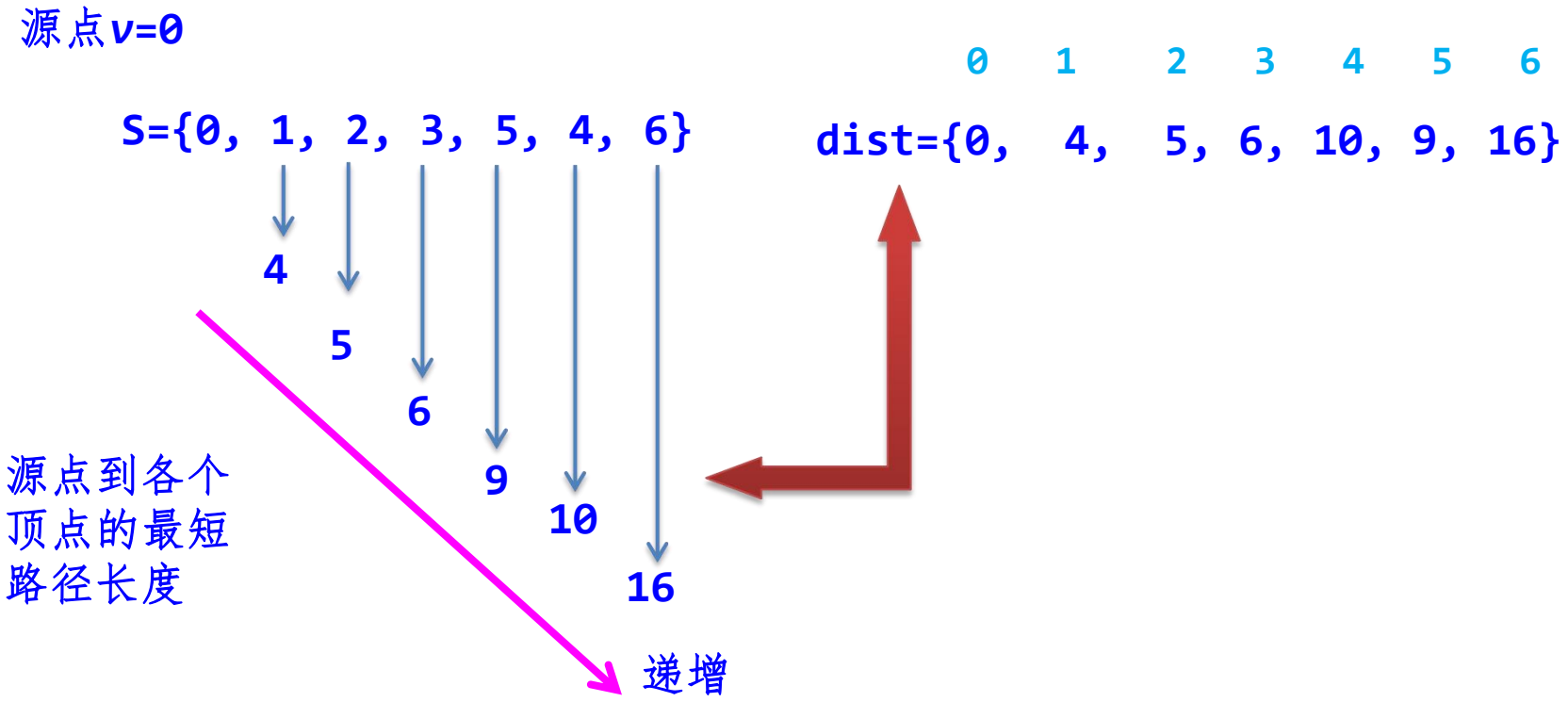
❷ 求0 ⇨ 6的最短路径：

```
      0   1   2   3   4   5   6
path={0,  0,  1,  0,  5,  2,  4}
```

path[6]=4
path[4]=5
path[5]=2
path[2]=1
path[1]=0到源点



观察求解结果



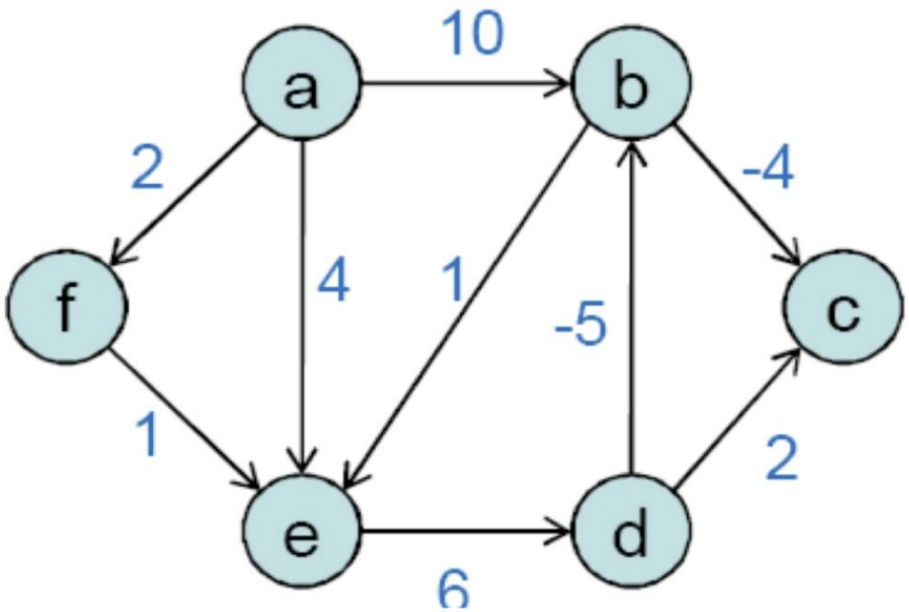
结论：

❶ 按顶点进入S的先后顺序，最短路径长度越来越长。

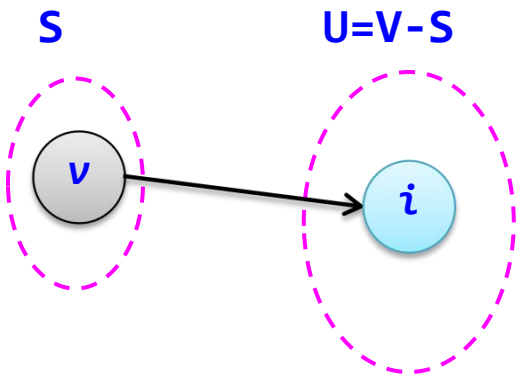
❷ 一个顶点一旦进入S后，其最短路径长度不再改变（调整）。

Dijkstra算法不适合含负权的图求最短路径

?



```
def Dijkstra1(g,v):                          #求从v到其他顶点的最短路径
    dist=[-1]*MAXV                           #建立dist数组
    path=[-1]*MAXV                           #建立path数组
    S=[0]*MAXV                               #建立S数组
    for i in range(g.n):
        dist[i]=g.edges[v][i]                #最短路径长度初始化
        if g.edges[v][i]<INF:                #最短路径初始化
            path[i]=v                        #v到i有边，置路径上顶点i的前驱为v
        else:                                #v到i没边时，置路径上顶点i的前驱为-1
            path[i]=-1
    S[v]=1                                   #源点v放入S中
```



- dist[i]=g.edges[v][i]
- path[i]=v

```
u=-1
for i in range(g.n-1):                  #循环向S中添加n-1个顶点
   mindis=INF                           #mindis置最小长度初值
   for j in range(g.n):                 #选取不在S中且具有最小距离的顶点u
      if S[j]==0 and dist[j]<mindis:
         u=j
         mindis=dist[j]
   S[u]=1                               #顶点u加入S中

   for j in range(g.n):                 #修改不在s中的顶点的距离
     if S[j]==0:                        #仅仅修改S中的顶点j
        if g.edges[u][j]<INF and dist[u]+g.edges[u][j]<dist[j]:
           dist[j]=dist[u]+g.edges[u][j]
           path[j]=u
DispAllPath(dist,path,S,v,g.n)   #输出所有最短路径及长度
```

算法的时间复杂度为$O(n^2)$。

```
def DispAllPath(dist,path,S,v,n):                    #输出从顶点v出发的所有最短路径
  for i in range(n):                                 #循环输出从顶点v到i的路径
    if S[i]==1 and i!=v:
      apath=[]
      print("  从%d到%d最短路径长度: %d \t路径:" %(v,i,dist[i]),end=' ')
      apath.append(i)                                #添加路径上的终点
      k=path[i];
      if k==-1:                                      #没有路径的情况
         print("无路径")
      else:                                          #存在路径时输出该路径
         while k!=v:
            apath.append(k)                          #顶点k加入到路径中
            k=path[k]
         apath.append(v)                             #添加路径上的起点
         apath.reverse()                             #逆置apath
         print(apath)                                #输出最短路径
```

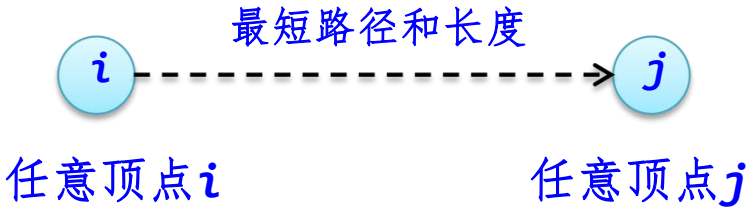
3*．改进的狄克斯特拉算法设计

前面的狄克斯特拉算法从两个方面进行优化，这里仅仅输出最短路径长度：

- 采用邻接表存储图，可以更快地查找到顶点u的所有邻接点并进行调整，时间为O(MAX(图中顶点的出度))。
- 求目前一个最短路径长度的顶点u时，Dijkstra1算法采用简单比较方法，可以改为采用优先队列（小根堆）求解。由于最多e条边进队，对应的时间为$O(\log_2 e)$。

算法的最坏时间复杂度为$O(e\log_2 e)$

多源最短路径算法



- 调用Dijkstra 算法n次；
- Floyd算法；

7.6.3 弗洛伊德算法

Robert W.Floyd（1936—2001）

- 弗洛伊德1936年6月8日生于纽约。说他“自学成才”并不是说他没有接受过高等教育，他是芝加哥大学的毕业生，但学的不是数学或电气工程等与计算机密切相关的专业，而是文学，1953年获得文学士学位。
- 弗洛伊德通过勤奋学习和深入研究，在计算机科学的诸多领域：算法，程序设计语言的逻辑和语义，自动程序综合，自动程序验证，编译器的理论和实现等方面都作出创造性的贡献。
- 弗洛伊德是1978年获得图灵奖。

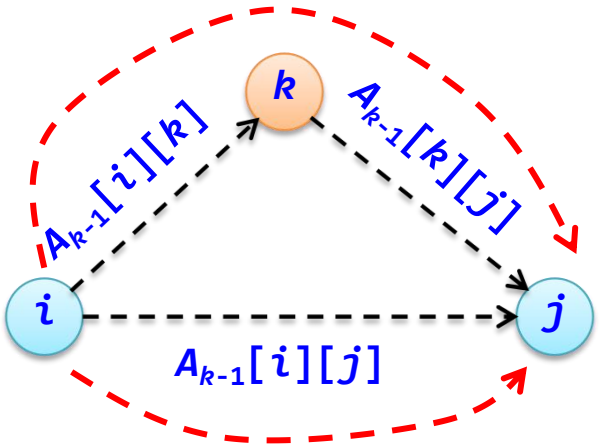
- 假设有向图G=(V,E)采用邻接矩阵g表示，另外设置一个二维数组A用于存放当前顶点之间的最短路径长度，即分量$A[i][j]$表示当前顶点i到顶点j的最短路径长度。
- 弗洛伊德算法的基本思想是递推产生一个矩阵序列A_0、A_1、…、A_k、…、A_{n-1}，其中$A_k[i][j]$表示从顶点i到顶点j的路径上所经过的顶点编号不大于k的最短路径长度。

按顶点编号顺序进行迭代

归纳起来，弗洛伊德思想可用如下的表达式来描述：

$A_{-1}[i][j]=g.edges[i][j]$

$A_k[i][j]=MIN\{A_{k-1}[i][j],A_{k-1}[i][k]+A_{k-1}[k][j]\}$　$0 \leqslant k \leqslant n-1$



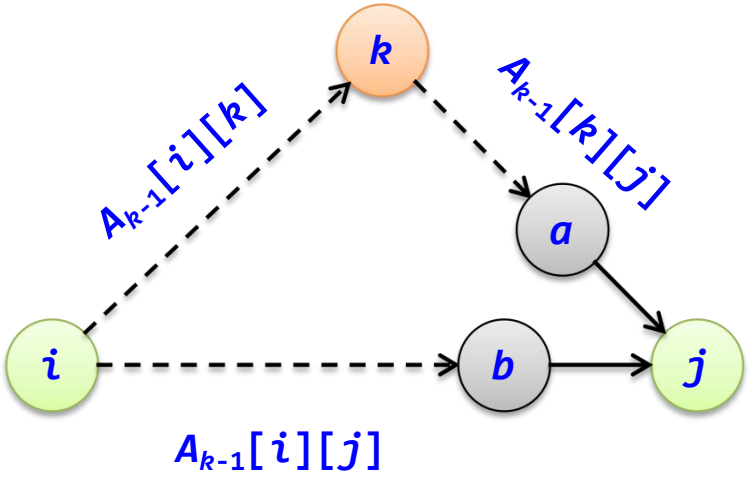
两条路径中选最小者：

$A_k[i][j]=$ MIN {

$A_{k-1}[i][j]$,

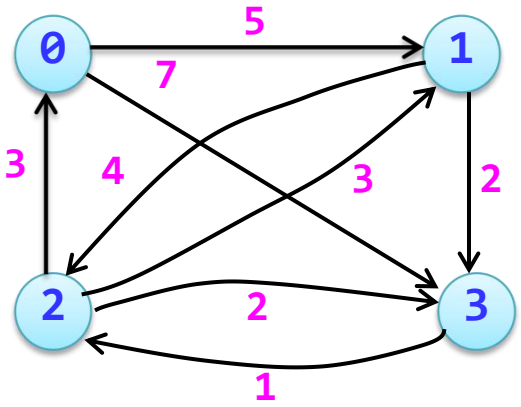
$A_{k-1}[i][k]+A_{k-1}[k][j]$ }

- 另外用二维数组path保存最短路径，它与当前迭代的次数有关。
- $path_k[i][j]$在考虑顶点k时得到的从顶点i到顶点j的最短路径的前驱顶点。



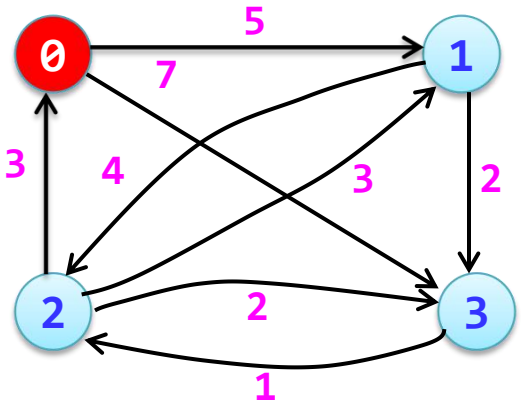
- $path_{k-1}[i][j]=b$，$path_{k-1}[k][j]=a$
- 若$A_{k-1}[i][j]>A_{k-1}[i][k]+A_{k-1}[k][j]$，选择经过顶点$k$的路径，即$path_k[i][j]=a=path_{k-1}[k][j]$。
- 否则不改变。

示例



$$\begin{bmatrix} 0 & 5 & \infty & 7 \\ \infty & 0 & 4 & 2 \\ 3 & 3 & 0 & 2 \\ \infty & \infty & 1 & 0 \end{bmatrix}$$

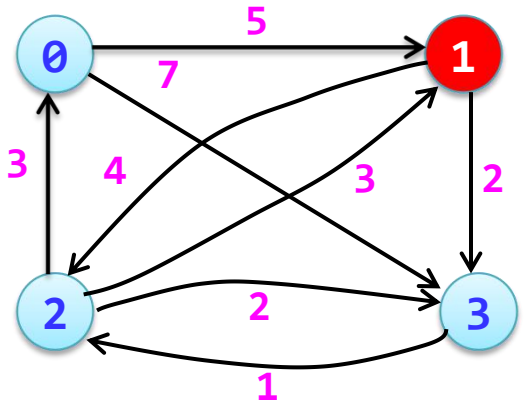
A_{-1}				$path_{-1}$			
0	5	∞	7	-1	0	-1	0
∞	0	4	2	-1	-1	1	1
3	3	0	2	2	2	-1	2
∞	∞	1	0	-1	-1	3	-1



在考虑顶点0时：

没有任何最短路径得到修改！

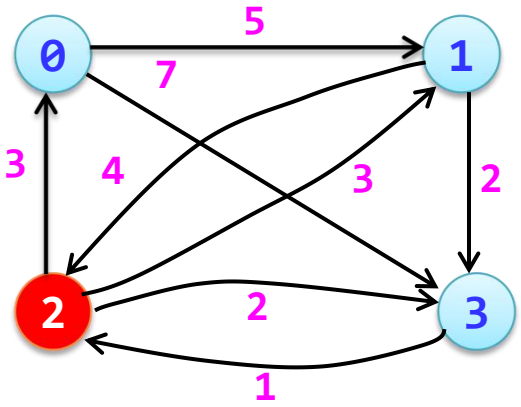
A_0				$path_0$			
0	5	∞	7	-1	0	-1	0
∞	0	4	2	-1	-1	1	1
3	3	0	2	2	2	-1	2
∞	∞	1	0	-1	-1	3	-1



在考虑顶点1时：

- 0→2：由无路径改为0→1→2，长度为9，path[0][2]改为1

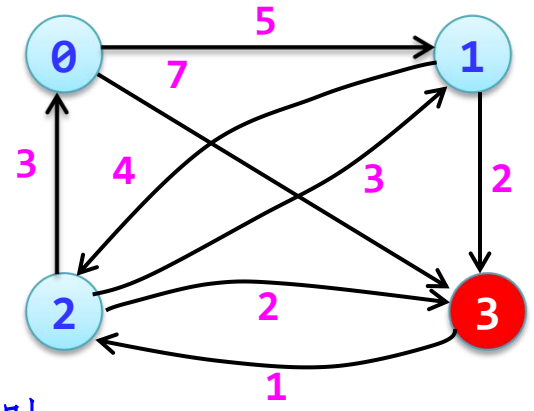
A_1				$path_1$			
0	5	9	7	-1	0	1	0
∞	0	4	2	-1	-1	1	1
3	3	0	2	2	2	-1	2
∞	∞	1	0	-1	-1	3	-1



在考虑顶点2时：

- 1→0：由无路径改为1→2→0，长度为7， path[1][0]改为2
- 3→0：由无路径改为3→2→0，长度为4， path[3][0]改为2
- 3→1：由无路径改为3→2→1，长度为4， path[3][1]改为2

A_2				$path_2$			
0	5	9	7	-1	0	1	0
7	0	4	2	2	-1	1	1
3	3	0	2	2	2	-1	2
4	4	1	0	2	2	3	-1



在考虑顶点3时：

- 0→2：由0→1→2改为0→3→2，长度为8，path[0][2]改为3
- 1→0：由1→2→0改为1→3→2→0，长度为6，path[1][0]改为2
- 1→2：由1→2改为1→3→2，长度为3，path[1][2]改为3

A_3				$path_3$			
0	5	8	7	-1	0	3	0
6	0	3	2	2	-1	3	1
3	3	0	2	2	2	-1	2
4	4	1	0	2	2	3	-1

A_3				$path_3$			
0	5	8	7	-1	0	3	0
6	0	3	2	2	-1	3	1
3	3	0	2	2	2	-1	2
4	4	1	0	2	2	3	-1

❶ 求1 ⇨ 0的最短路径长度：

由A_3数组可以直接得到两个顶点之间的最短路径长度，如$A_3[1][0]=6$，说明顶点1到0的最短路径长度为6。

A_3				$path_3$			
0	5	8	7	-1	0	3	0
6	0	3	2	2	-1	3	1
3	3	0	2	2	2	-1	2
4	4	1	0	2	2	3	-1

❷ 求1 ⇨ 0的最短路径：

path[1][0]=2
path[1][2]=3
path[1][3]=1（找到起点）

得到的顶点序列为0、2、3、1
顶点1到0的最短路径为1→3→2→0

2．弗洛伊德算法设计

```
def Floyd(g):                                  #输出所有两个顶点之间的最短路径
  A=[[0]*MAXV for i in range(MAXV)]            #建立A数组
  path=[[0]*MAXV for i in range(MAXV)]         #建立path数组
  for i in range(g.n):                         #给数组A和path置初值即求A_-1[i][j]
    for j in range(g.n):
      A[i][j]=g.edges[i][j]
      if i!=j and g.edges[i][j]<INF:
         path[i][j]=i                          #i和j顶点之间有边时
      else:
         path[i][j]=-1                         #i和j顶点之间没有边时
```

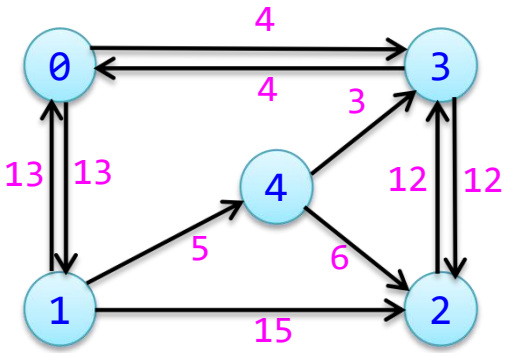
```
for k in range(g.n):                          #求Ak[i][j]
   for i in range(g.n):
      for j in range(g.n):
         if A[i][j]>A[i][k]+A[k][j]:
            A[i][j]=A[i][k]+A[k][j]
            path[i][j]=path[k][j]             #修改最短路径
Dispath(A,path,g)                             #生成最短路径和长度
```

```
def Dispath(A,path,g):                          #输出所有的最短路径和长度
  for i in range(g.n):
     for j in range(g.n):
       if A[i][j]!=INF and i!=j:                #若顶点i和j之间存在路径
          print("  顶点%d到%d的最短路径长度: %d\t路径:" %(i,j,A[i][j]),end='')
          k=path[i][j]
          apath=[j]                             #路径上添加终点
          while k!=-1 and k!=i:                 #路径上添加中间点
             apath.append(k)                    #顶点k加入到路径中
             k=path[i][k]
          apath.append(i)                       #路径上添加起点
          apath.reverse()                       #逆置apath
          print(apath)                          #输出最短路径
```

上述弗洛伊德算法中有三重循环，其时间复杂度为$O(n^3)$。

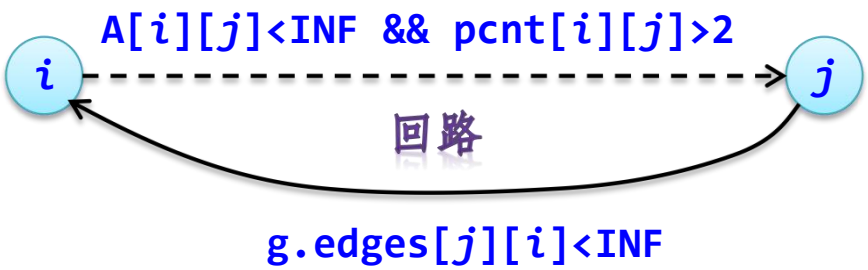
- 弗洛伊德算法适合含负权的图求最短路径
- 不适合含负权回路的图求最短路径

【例7.11】假设一个带权有向图G采用邻接矩阵存储结构表示（所有权值为正整数），设计一个算法求其中的一个最小环的长度，要求这样的环至少包含3个顶点，并求出如下图中满足要求的最小环长度。



解： 采用Floyd算法求存放任意两个顶点之间最短路径长度的数组A。

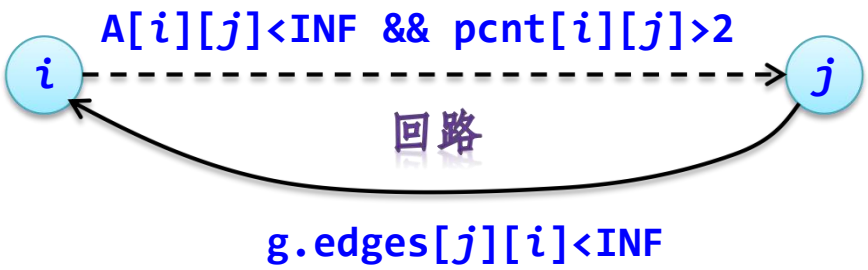
- 设置一个二维数组pcnt，pcnt[i][j]表示从顶点i到j的最短路径上包含的顶点个数。
- 当求出A和pcnt后，如果i≠j并且A[i][j]<INF && pcnt[i][j]>2，表示顶点i到j有一条包含3个或者更多顶点的最短路径，若g.edges[j][i]<INF，表示顶点j到i有一条边，则构成一个至少包含3个顶点的环，其长度为A[i][j]+g.edges[j][i]。
- 通过比较求出长度最小环的长度。

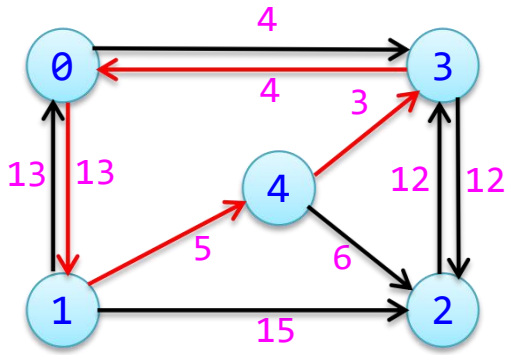
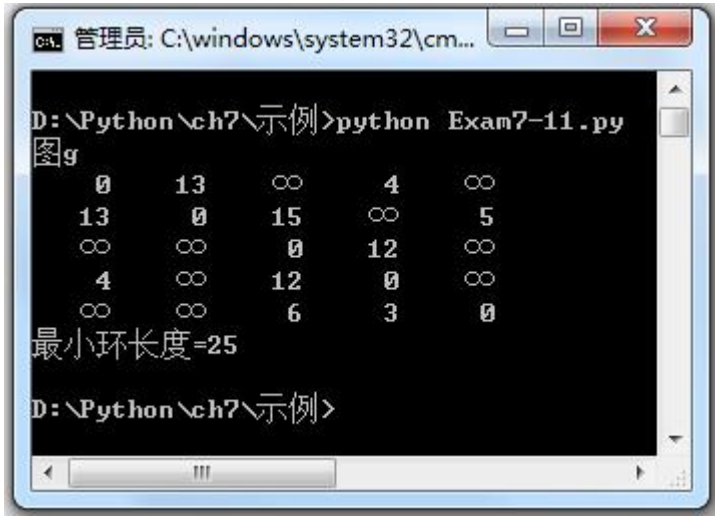


```
from MatGraph import MatGraph,INF,MAXV
A=[[0]*MAXV for i in range(MAXV)]                    #全局变量,A数组
pcnt=[[0]*MAXV for i in range(MAXV)]                 #全局变量,路径中顶点个数

def Floyd(g):                                        #Floyd算法
  for i in range(g.n):                               #数组A和pcnt初始化
     for j in range(g.n):
        A[i][j]=g.edges[i][j]
        if i!=j and g.edges[i][j]<INF:
           pcnt[i][j]=2;                             #<i,j>作为路径，含2个顶点
        else:
           pcnt[i][j]=0;                             #没有路径，顶点个数为0
  for k in range(g.n):                               #求Ak[i][j]和pcnt[i][j]
     for i in range(g.n):
        for j in range(g.n):
           if A[i][j]>A[i][k]+A[k][j]:
              A[i][j]=A[i][k]+A[k][j]
              pcnt[i][j]=pcnt[i][k]+pcnt[k][j]-1
```

```
def Mincycle(g):                          #找一个最小环长度
    minl=INF                              #最小环长初始化为INF
    Floyd(g)
    for i in range(g.n):
      for j in range(g.n):
          if i!=j and A[i][j]<INF and pcnt[i][j]>2 and g.edges[j][i]<INF:
             minl=min(minl,A[i][j]+g.edges[j][i])
    return minl
```



程序验证

```
#主程序
g=MatGraph()
n,e=5,10
a=[[0,13,INF,4,INF],[13,0,15,INF,5], \
   [INF,INF,0,12,INF],[4,INF,12,0,INF],[INF,INF,6,3,0]]
g.CreateMatGraph(a,n,e)
print("图g")
g.DispMatGraph()
print("最小环长度=%d" %(Mincycle(g)))
```