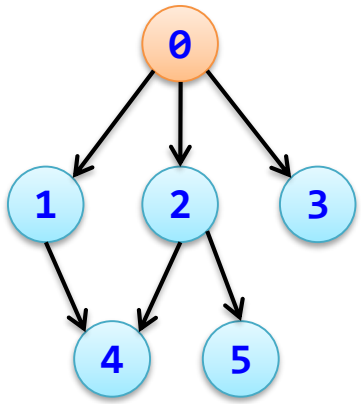
7.3 图的遍历

7.3.1 图遍历的概念

- 从图中任意指定的顶点（称为初始点）出发，按照某种搜索路径访问图中的所有顶点，且每个顶点仅被访问一次，称此过程为图的遍历。
- 如果是连通的无向图或者强连通的有向图，则遍历一次就能完成，并可按访问的先后顺序得到由所有顶点组成的一个序列。

- 根据遍历方式的不同，图的遍历方法有：一种是深度优先遍历（DFS）方法；另一种是广度优先遍历（BFS）方法。
- 为了避免同一个顶点被重复访问，可设置一个访问标志数组**visited**，初始时所有元素置为**0**，当顶点**i**访问过时，**visited[i]**置为**1**。



DFS: 0 1 4 2 5 3

BFS: 0 1 2 3 4 5

7.3.2 深度优先遍历

- 从某个起始点v出发进行深度优先遍历–DFS(v)，首先访问初始顶点v。
- 然后选择一个与顶点v邻接且没被访问过的顶点w为起始点，再从w出发进行深度优先遍历–DFS(w)，直到图中与当前顶点v邻接的所有顶点都被访问过为止。

递归过程：可借助于堆栈

图采用邻接表存储，其深度优先遍历算法如下（其中，v是起始点编号，visited是全局数组）：

```
MAXV=100                                    #全局变量，表示最多顶点个数
visited=[0]*MAXV
def DFS(G,v):                               #邻接表G中顶点v出发深度优先遍历
  print(v,end=' ')                          #访问顶点v
  visited[v]=1                              #置已访问标记
  for j in range(len(G.adjlist[v])):        #处理顶点v的所有出边顶点
    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顶点未访问
        DFS(G,w)                            #从w开始递归遍历
```

时间复杂度为O(*n*+*e*)。

或者

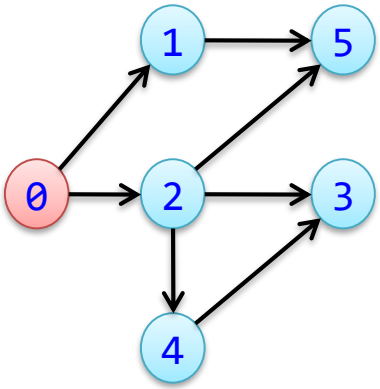
```
def DFS1(G,v):                          #邻接表G中从顶点v出发的深度优先遍历
  print(v,end=' ')                      #访问顶点v
  visited[v]=1                          #置已访问标记
  for p in G.adjlist[v]:                #处理顶点v的所有出边顶点
    w=p.adjvex                          #取顶点v的一个邻接点w
    if visited[w]==0:
       DFS1(G,w)                        #若w顶点未访问,递归访问它
```

- 上述DFS和DFS1完全相同，仅仅遍历顶点v的邻接点的语法形式不同！
- 后面的图算法中主要采用DFS的形式。

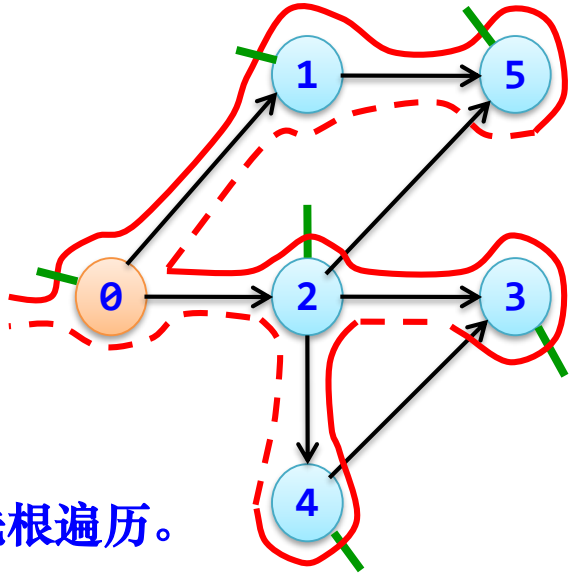
图采用邻接矩阵存储，其深度优先遍历算法如下（其中，v是起始点编号，visited是全局数组）：

```
MAXV=100                                    #全局变量，表示最多顶点个数
visited=[0]*MAXV
def DFS(g,v):                               #邻接矩阵g中顶点v出发深度优先遍历
  print(v,end=" ")                          #访问顶点v
  visited[v]=1                              #置已访问标记
  for w in range(g.n):
     if g.edges[v][w]!=0 and g.edges[v][w]!=INF:
        if visited[w]==0:                   #存在边<v,w>并且w没有访问过
           DFS(g,w)                         #从w开始递归遍历
```

时间复杂度为$O(n^2)$。




```
[
    [[1, 1], [2, 1]]                    #顶点0
    [[5, 1]]                            #顶点1
    [[3, 1], [4, 1], [5, 1]]            #顶点2
    []                                  #顶点3
    [[3, 1]]                            #顶点4
    []                                  #顶点5
]
```



DFS: 0 1 5 2 3 4

说明

类似树的先根遍历。

7.3.3 广度优先遍历

- 首先访问起始点v。
- 接着访问顶点v的所有未被访问过的邻接点v_1、v_2、…、v_t。
- 然后再按照v_1、v_2、…、v_t的次序，访问每一个顶点的所有未被访问过的邻接点。
- 依次类推，直到图中所有和初始点v有路径相通的顶点或者图中所有已访问顶点的邻接点都被访问过为止。

相邻顶点：先访问先处理 → **队列**

图采用邻接表存储，其广度优先遍历算法如下：

```
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
```

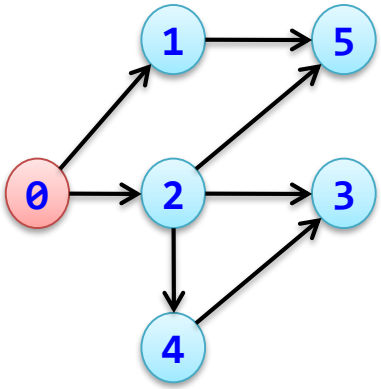
时间复杂度为$O(n+e)$。

图采用邻接矩阵存储，其广度优先遍历算法如下：

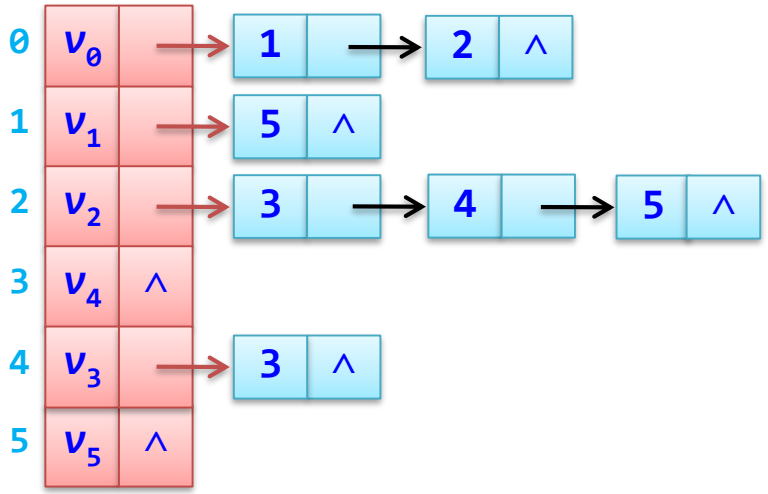
```
from collections import deque
MAXV=100                                    #全局变量，表示最多顶点个数
visited=[0]*MAXV
def BFS(g,v):                               #邻接矩阵g中顶点v出发广度优先遍历
  qu=deque()                                #将双端队列作为普通队列qu
  print(v,end=" ")                          #访问顶点v
  visited[v]=1                              #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 w in range(g.n):
      if g.edges[v][w]!=0 and g.edges[v][w]!=INF:
        if visited[w]==0:                   #存在边<v,w>并且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
```

时间复杂度为$O(n^2)$。

更直观的邻接表形式：

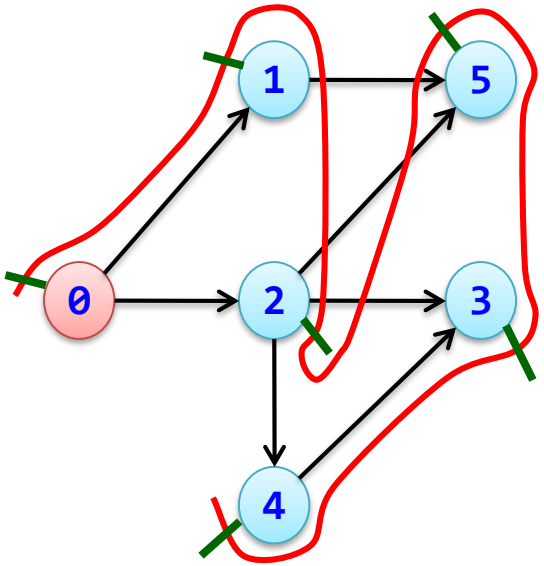


（a）一个有向图G_4



（b）图G_4的邻接表

BFS：0 1 2 5 3 4



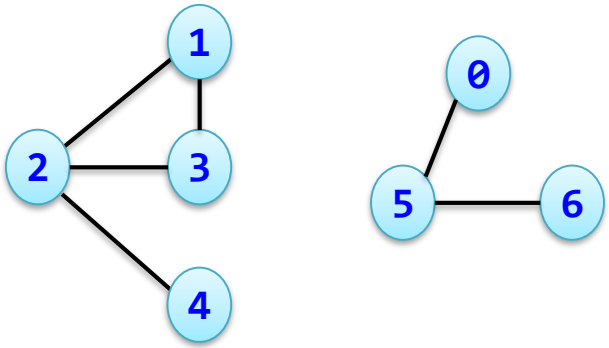
说明

类似树的层次遍历。

7.3.4 非连通图的遍历

无向图

- 连通图：一次遍历能够访问到图中的所有顶点。
- 非连通图：一次遍历只能访问到起始点所在连通分量中的所有顶点，其他连通分量中的顶点是不可能访问到的。为此需要从其他每个连通分量中选择起始点，分别进行遍历，才能够访问到图中的所有顶点。



非连通图采用邻接表存储结构，其深度优先遍历算法如下：

```
def DFSA(G):                          #非连通图的DFS
  for i in range(G.n):
    if visited[i]==0:                 #若顶点i没有访问过
        DFS(G,i)                      #从顶点i出发深度优先遍历
```

非连通图采用邻接表存储结构，其广度优先遍历算法如下：

```
def BFSA(G):                          #非连通图的BFS
  for i in range(G.n):
    if visited[i]==0:                 #若顶点i没有访问过
        BFS(G,i)                      #从顶点i出发广度优先遍历
```

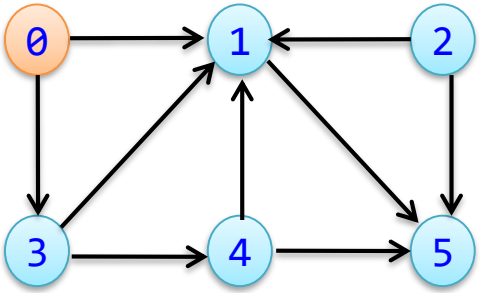
【例7.4】假设图采用邻接表存储，设计一个算法，判断一个无向图是否连通。若连通则返回true；否则返回false。

```
def DFS1(G,v):                                  #邻接表G顶点v出发深度优先遍历
  visited[v]=1                                  #置已访问标记
  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个邻接点w
     if visited[w]==0:
        DFS1(G,w)                               #若w顶点未访问,递归访问它

def Connect(G):                                 #判断无向图G的连通性
  flag=True
  DFS1(G,0)                                     #调用DSF1算法,从0出发DFS
  for i in range(G.n):
     if visited[i]==0:
        flag=False                              #存在没有访问的顶点，则不连通
        break
  return flag
```

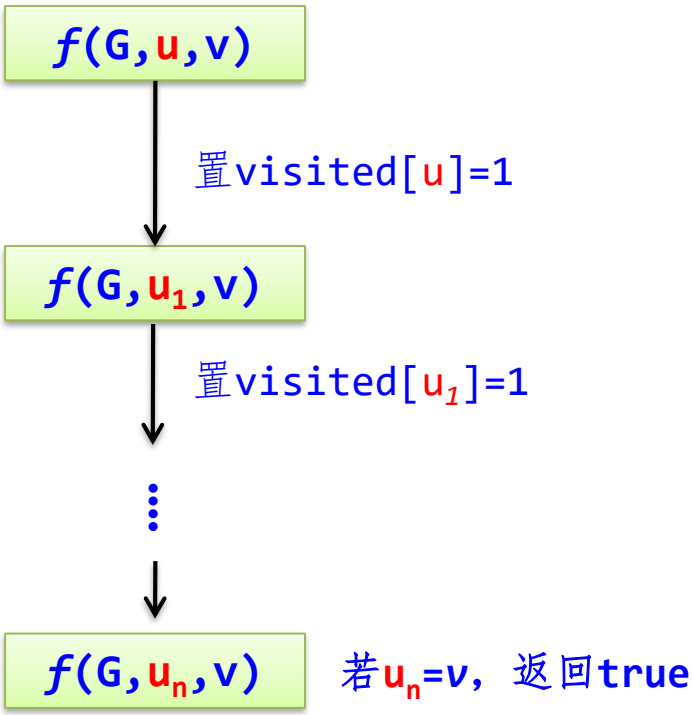
7.4 图遍历算法的应用

7.4.1 深度优先遍历算法的应用

【例7.5】假设图G采用邻接表存储，设计一个算法判断顶点u到顶点v之间是否有路径。并对于以下有向图，判断从顶点0到顶点5、从顶点0到顶点2是否有路径。

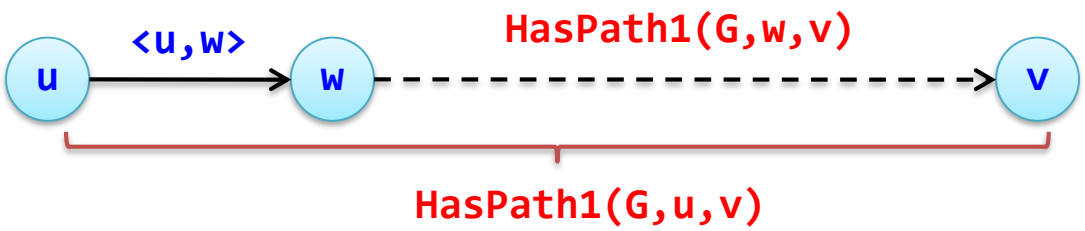


思路



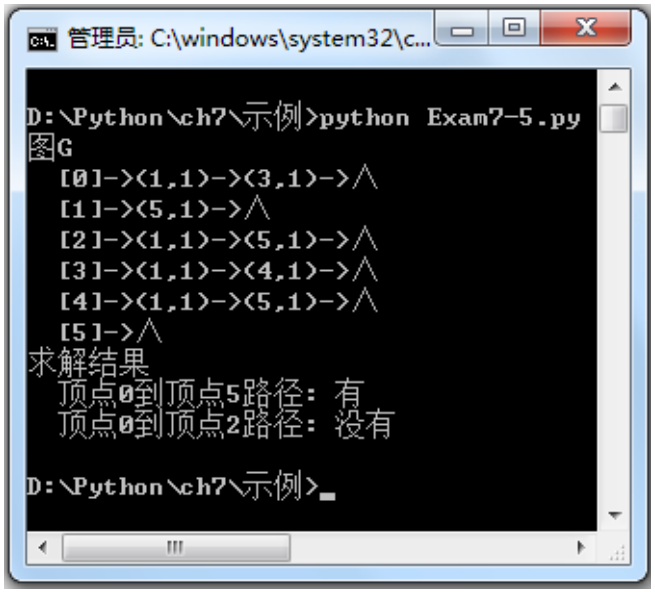
```
from AdjGraph import AdjGraph,INF
MAXV=100                                    #全局变量，表示最多顶点个数
visited=[0]*MAXV                            #全局访问标志数组

def HasPath(G,u,v):                         #判断u到v是否有简单路径
  for i in range(G.n): visited[i]=0         #初始化
  return HasPath1(G,u,v)
```

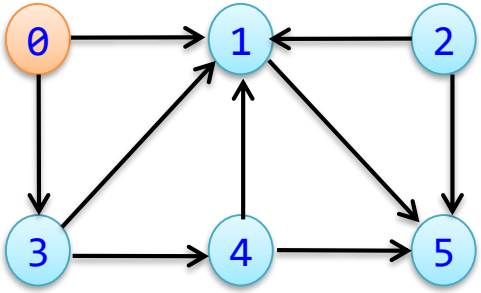
```
def HasPath1(G,u,v):                          #被HasPath方法调用
  visited[u]=1
  for j in range(len(G.adjlist[u])):          #处理顶点u的所有出边
    w=G.adjlist[u][j].adjvex                  #取顶点u的第j个邻接点w
    if w==v:                                  #找到目标点后返回真
      return True                             #表示u到v有路径
    if visited[w]==0:
      if HasPath1(G,w,v)==True: return True
  return False
```



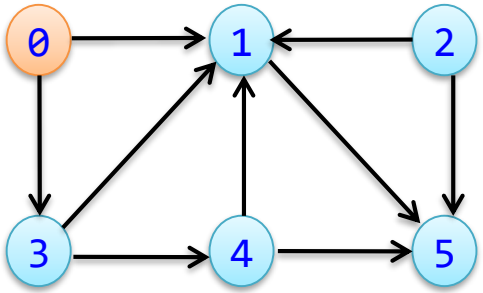
程序验证

```
#主程序
G=AdjGraph()
n,e=6,9
a=[[0,1,0,1,0,0],[0,0,0,0,0,1],[0,1,0,0,0,1],
[0,1,0,0,1,0],[0,1,0,0,0,1],[0,0,0,0,0,0]]
G.CreateAdjGraph(a,n,e)  #创建图7.14的邻接表
print("图G");G.DispAdjGraph()
u,v=0,5
print("求解结果")
print("  顶点%d到顶点%d路径: " %(u,v),end='')
print("有" if HasPath(G,u,v) else "没有")
u,v=0,2
print("  顶点%d到顶点%d路径: " %(u,v),end='')
print("有" if HasPath(G,u,v) else "没有")
```

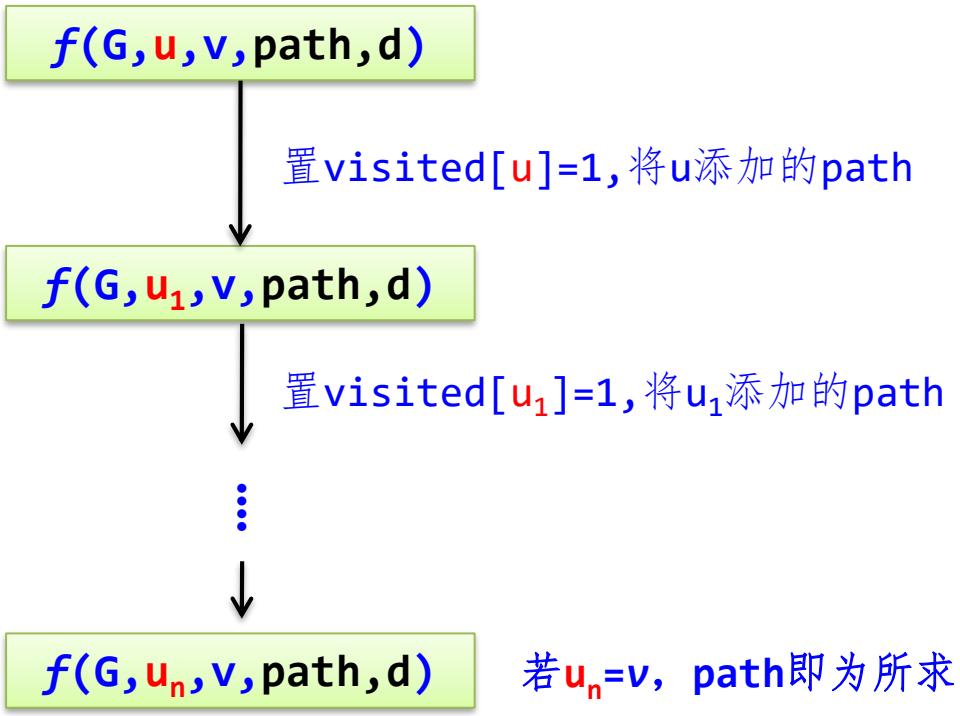




【例7.6】假设图G采用邻接表存储，设计一个算法求顶点*u*到顶点*v*之间的一条简单路径（假设两顶点之间存在一条或多条简单路径）。并对于以下有向图，求从顶点0到顶点4的一条简单路径。

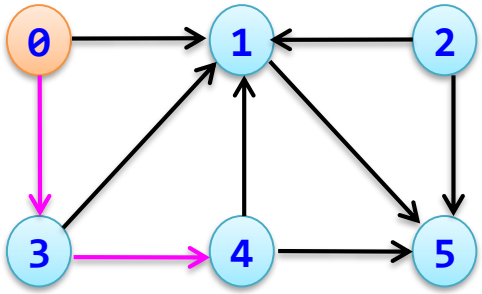


思路　path[0..d]存放一条路径



```
def FindaPath1(G,u,v):                          #解法1：求u到v的一条简单路径
  path=[-1]*MAXV
  d=-1                                          #path[0..d]存放一条路径
  for i in range(G.n): visited[i]=0             #visited数组初始化
  FindaPath11(G,u,v,path,d)
```

```
def FindaPath11(G,u,v,path,d):                  #被Findapath1调用
  visited[u]=1
  d+=1; path[d]=u                               #顶点u加入到路径中
  if u==v:                                      #找到一条路径后输出并返回
     for i in range(d+1):
        print(path[i],end=' ')
     print()
     return
  for j in range(len(G.adjlist[u])):            #处理顶点u的所有出边
      w=G.adjlist[u][j].adjvex                  #取顶点u的第j个邻接点w
      if visited[w]==0:                         #w没有访问过
         FindaPath11(G,w,v,path,d);             #递归调用
```



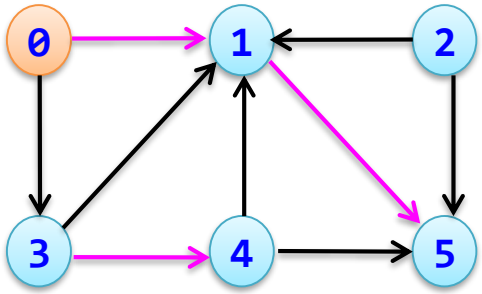
FindaPath1(G,0,4)

顶点0到顶点4路径：0 3 4

- 上述算法中是将path作为一个长度为MAXV的数组，通过path[0..*d*]表示路径的（*d*参数为int的不可变类型，具有自动回退功能）。
- 能不能直接用path列表存放路径呢？

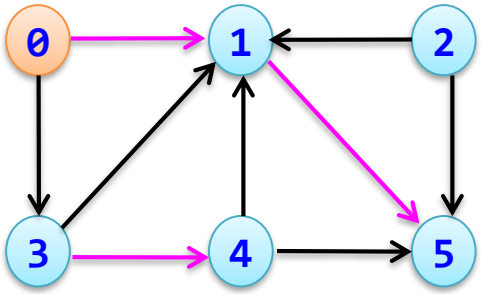
用path列表存放路径 

```
def FindaPath2(G,u,v):                          #解法2：求u到v的一条简单路径
  path=[]
  for i in range(G.n): visited[i]=0             #初始化
  FindaPath21(G,u,v,path)

def FindaPath21(G,u,v,path):                    #被Findapath2调用
  visited[u]=1
  path.append(u)                                #顶点u加入到路径中
  if u==v:                                      #找到一条路径后输出并返回
     print(path)
     return
  for j in range(len(G.adjlist[u])):            #处理顶点u的所有出边
     w=G.adjlist[u][j].adjvex                   #取顶点u的第j个邻接点w
     if visited[w]==0:                          #w没有访问过
        FindaPath21(G,w,v,path);                #递归调用
```



FindaPath2(G,0,4)

顶点0到顶点4路径：0 1 5 3 4



FindaPath2(G,0,4)

顶点0到顶点4路径：0 1 5 3 4

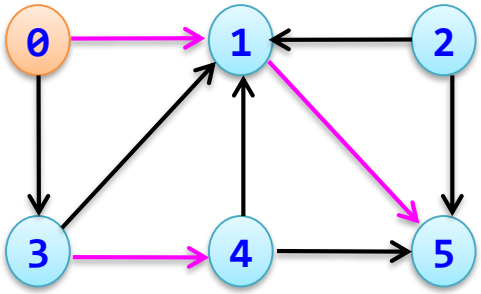
- 因为path列表是可变类型的参数，相当于全局变量，没有自动回退功能。
- 执行FindaPath2(G,0,4)语句时path中存放的是搜索轨迹，而不是从顶点0到顶点4的路径。
- 改正的方式是增加具有回退功能的代码，即访问顶点*u*时，将*u*添加到path中，如果从*u*出发没有找到终点*v*，则回退，也就是将顶点*u*从path中删除（*u*是path中最后添加的顶点）。

增加具有回退功能的代码 

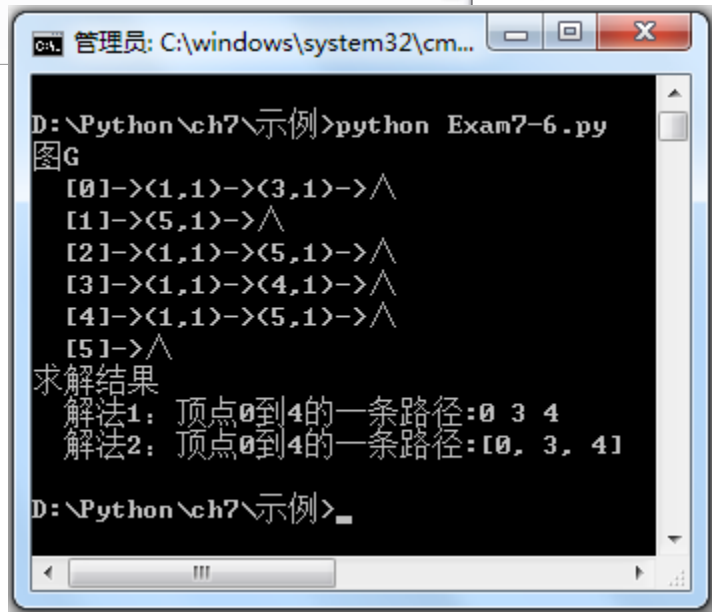
```
def FindaPath21(G,u,v,path):                    #被Findapath2调用
  visited[u]=1
  path.append(u)                                #顶点u加入到路径中
  if u==v:                                      #找到一条路径后输出并返回
     print(path)
     return
  for j in range(len(G.adjlist[u])):            #处理顶点u的所有出边
     w=G.adjlist[u][j].adjvex                   #取顶点u的第j个邻接点w
     if visited[w]==0:                          #w没有访问过
        FindaPath21(G,w,v,path);                #递归调用
  path.pop()                                    #增加的具有回退功能的代码
```


理解为：当顶点u存放的邻接点都处理后，从u回退到路径上的前一个顶点

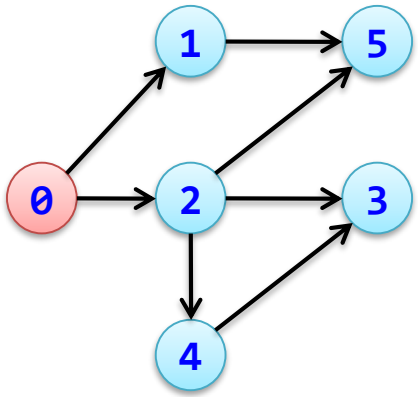
```
#主程序
G=AdjGraph()
n,e=6,9
a=[[0,1,0,1,0,0],[0,0,0,0,0,1],[0,1,0,0,0,1],
  [0,1,0,0,1,0],[0,1,0,0,0,1],[0,0,0,0,0,0]]
G.CreateAdjGraph(a,n,e)
print("图G");G.DispAdjGraph()
u,v=0,4
print("求解结果")
print("  解法1: 顶点%d到%d的一条路径:" %(u,v),end='')
FindaPath1(G,u,v)
print("  解法2: 顶点%d到%d的一条路径:" %(u,v),end='')
FindaPath2(G,u,v)
```



程序验证

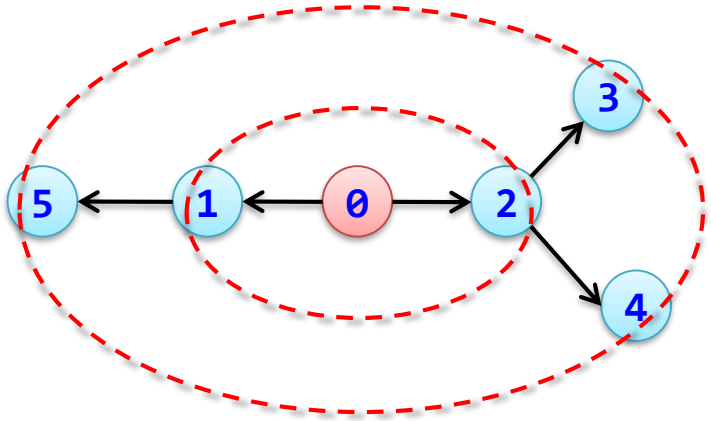


7.4.2 广度优先遍历算法的应用

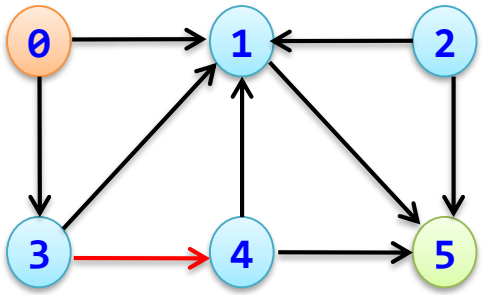


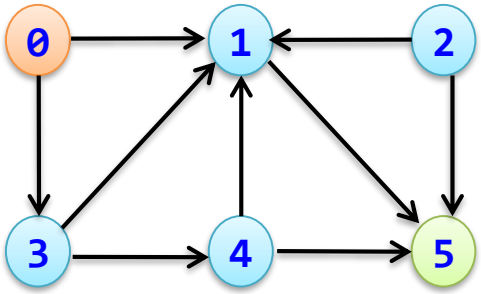
起始点0到图中其他顶点的最短路径长度：

0 : 0
0 → 1 : 1
0 → 2 : 1
0 → 1 → 5 : 2
0 → 2 → 3 : 2
0 → 2 → 4 : 2



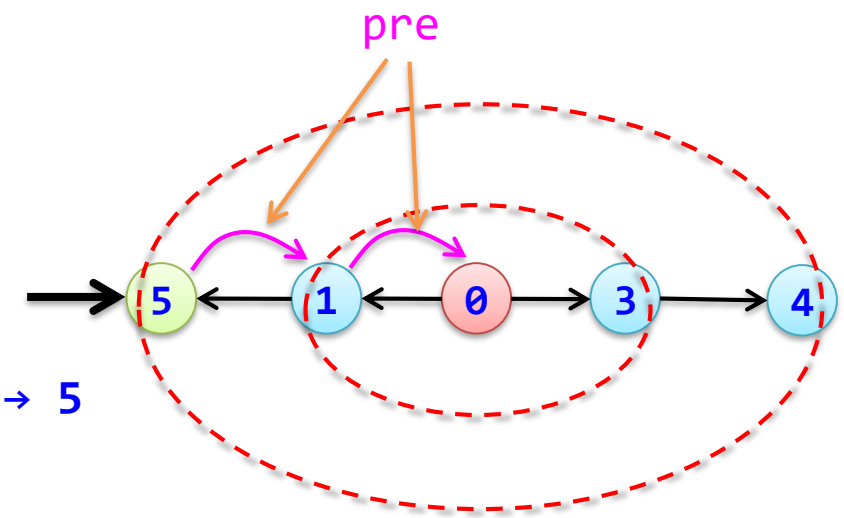
【例7.9】假设图G采用邻接表存储，设计一个算法，求不带权图G中从顶点*u*到顶点*v*的一条最短路径（假设两顶点之间存在一条或多条简单路径）。并对于以下有向图，求从顶点0到顶点5的一条最短简单路径。





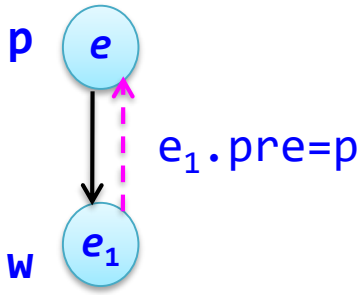
起始点0到图中其他顶点的最短路径长度：

0 : 0
0 → 1 : 1
0 → 3 : 1
0 → 1 → 5 : 2
0 → 3 → 4 : 2



5 1 0
反向：0 → 1 → 5

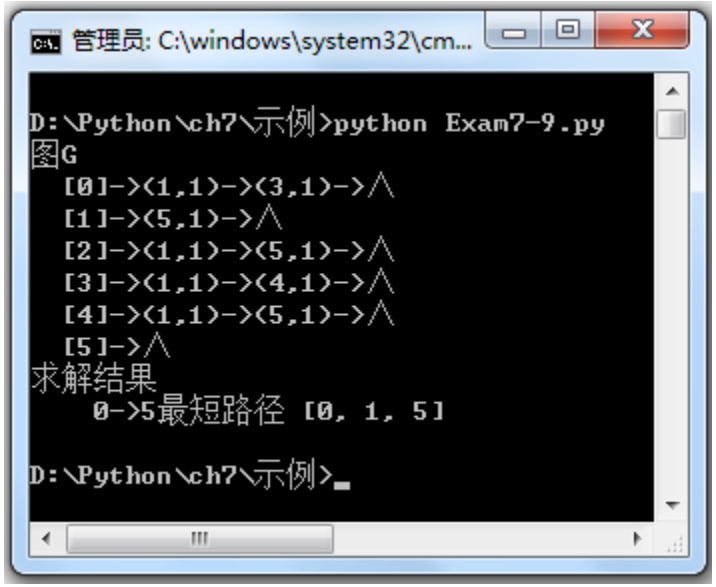
```
class QNode:                          #队列元素类
  def __init__(self,p,pre):           #构造方法
    self.vno=p                        #当前顶点编号
    self.pre=pre                      #当前结点的前驱结点
```



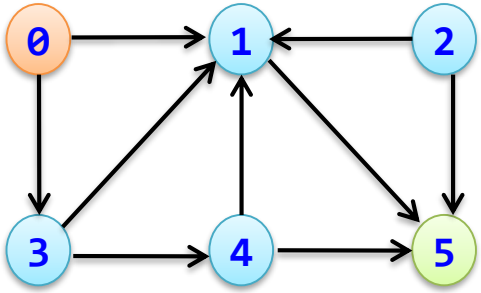
```
def ShortPath(G,u,v):                          #求u到v的一条最短简单路径
  res=[]                                       #存放结果
  qu=deque()                                   #定义一个队列qu
  qu.append(QNode(u,None))                     #起始点u(前驱为None)进队
  visited[u]=1                                 #置已访问标记
  while len(qu)>0:                             #队不空循环
     p=qu.popleft()                            #出队一个结点
     if p.vno==v:                              #当前结点p为v结点
        res.append(v)
        q=p.pre                                #q为前驱结点
        while q!=None:                         #找到起始结点为止
           res.append(q.vno)
           q=q.pre
        res.reverse()                          #逆置res构成正向路径
        return res
     for j in range(len(G.adjlist[p.vno])):    #处理顶点u的所有出边
        w=G.adjlist[p.vno][j].adjvex           #取顶点u的第j个邻接点w
        if visited[w]==0:                      #w没有访问过
           qu.append(QNode(w,p))               #置其前驱结点为p
           visited[w]=1                        #置已访问标记
```

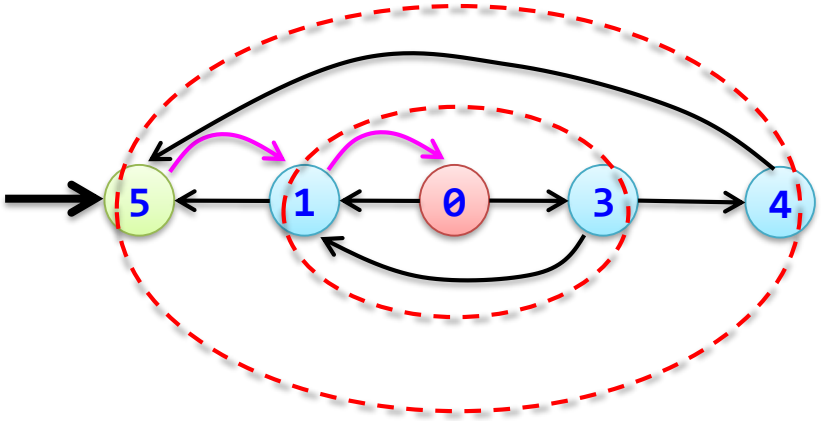
程序验证

```
#主程序
G=AdjGraph()
n,e=6,9
a=[[0,1,0,1,0,0],[0,0,0,0,0,1],[0,1,0,0,0,1],
   [0,1,0,0,1,0],[0,1,0,0,0,1],[0,0,0,0,0,0]]
G.CreateAdjGraph(a,n,e)
print("图G");G.DispAdjGraph()
u,v=0,5
print("求解结果")
print("    %d->%d最短路径" %(u,v),ShortPath(G,u,v))
```



为什么广度优先遍历找到的路径一定是最短路径呢?





- 广度优先遍历找到的路径上的每个顶点均为不同层次的顶点，所以该路径一定是最短路径。
- 深度优先遍历找到的路径上的顶点可能属于相同层次的顶点，所以不一定是最短路径。