广度优先遍历

BFS（Breadth First Search)

访问方式如下：

(1) 从选中的某一个未访问过的顶点出发，访问并对该顶点加已访问标志

(2) 依次对该顶点的未被访问过的第1个、第2个、第3个……第 k 个邻接点 v_1 、v_2、v_3…… v_k进行访问且加已访问标志

(3) 依次对顶点 v_1 、v_2、v_3…… v_k转向操作(2)

(4) 如果还有顶点未被访问过，选中其中一个顶点作为起始顶点，再次转向(1)。如果所有的顶点都被访问到，遍历结束

广度优先遍历

BFS（Breadth First Search）

每个结点要经历的三个阶段（状态）

- 未被发现 (Undiscovered)
- 被发现（Discovered）
- 遍历结束（Finished）

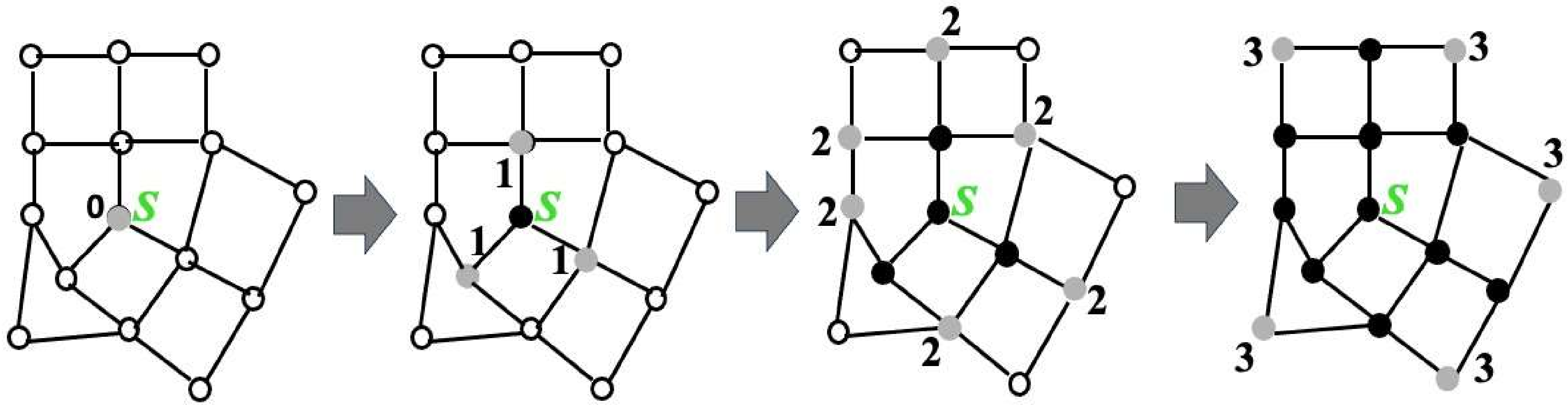
结点及其所有邻接结点已被发现

高等教育出版社

广度优先遍历

BFS（Breadth First Search）

- Undiscovered
- Discovered
- Finished



数值表示从起点到各结点的最短路径长度！（边无权重或权重均为1）

广度优先遍历算法

算法7-13: 按广度优先遍历图中结点 BFS(*graph*)

输入： 图*graph*

输出： 图*graph*的广度优先遍历序列

```
1.  for v ← 0 to graph.n_verts-1 do //初始化各顶点的已访问标志为未访问
2.  |  visited[v] ← false
3.  end
4.  for v ← 0  to graph.n_verts-1 do
5.  |  if visited[v] = false then
6.  |  |  visited[v] ← true    //标记起点
7.  |  |  prev[v] ← -1   //起点无前驱
8.  |  |  dist[v] ← 0  //离起点的最小距离（最少边数）
9.  |  |  BFS(graph, v, visited)
10. |  end
11. end
```

1-3：时间复杂度 O（n）

4-11：依赖于第6行BFS时间复杂度

广度优先遍历算法

算法7-14: 按广度优先遍历图中结点 BFS(*graph, v, visited*)

输入： 图*graph*，出发顶点*v*，已访问标志数组*visited*

输出： 图*graph*中从顶点*v*出发的广度优先遍历序列

```
InitQueue(queue)
EnQueue(queue, v)    //起点入队
while IsEmpty(queue) = false do
|  u ← DeQueue(queue)
|  Visit(graph, u)              //访问结点
|  p ← graph.ver_list[u].adj
|  while p ≠ NIL do             //遍历邻接结点
|  |  if visited[p.dest] = false then   //邻接结点未发现！        Undiscovered
|  |  |  visited[p.dest] ← true
|  |  |  prev[p.dest] ← u
|  |  |  dist[p.dest] ← dist[u] + 1   //设置离起点的最小距离    Discovered
|  |  |  EnQueue(queue, p.dest)   //邻接结点入队
|  |  end
|  end          Finished
end
```

广度优先遍历算法

算法7-14: 按广度优先遍历图中结点 BFS(*graph, v, visited*)

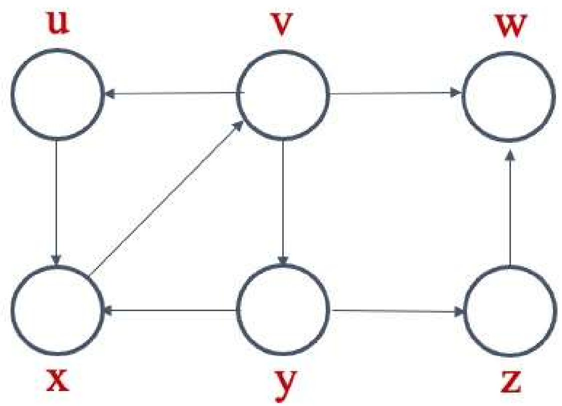
输入： 图*graph*，出发顶点*v*，已访问标志数组*visited*

输出： 图*graph*中从顶点*v*出发的广度优先遍历序列

```
1.  InitQueue(queue)
2.  EnQueue(queue, v)    //起点入队
3.  while IsEmpty(queue) = false do
4.  | u ← DeQueue(queue)
5.  | Visit(graph, u)            //访问结点
6.  | p ← graph.ver_list[u].adj
7.  | while p ≠ NIL do          //遍历邻接结点
8.  | | if visited[p.dest] = false then   //邻接结点未发现!
9.  | | | visited[p.dest] ← true
10. | | | prev[p.dest] ← u
11. | | | dist[p.dest] ← dist[u] + 1   //设置离起点的最小距离
12. | | | EnQueue(queue, p.dest)   //邻接结点入队
13. | | end
14. | end
15. end
```

- 每个结点入队一次、出队一次
- 每个结点被查询（第8行代码）的次数等于其入度（？）
- 时间复杂度：O(n+m)
- 空间复杂度：O(n)？

广度优先遍历算法示例

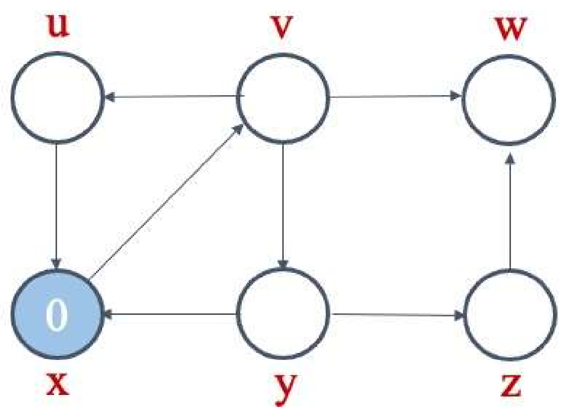


queue:

visited	F	F	F	F	F	F
prev						
	u	v	x	y	w	z

广度优先遍历算法示例

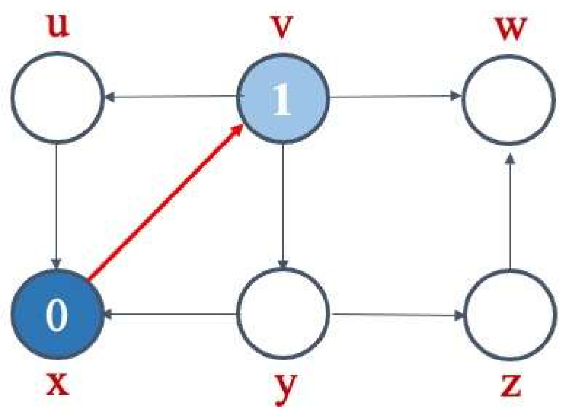
queue: x



	u	v	x	y	w	z
visited	F	F	T	F	F	F
prev			-1			

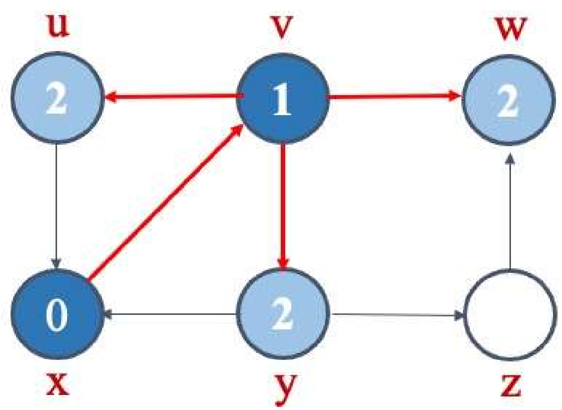
广度优先遍历算法示例

queue: v



	u	v	x	y	w	z
visited	F	T	T	F	F	F
prev		x	-1			

广度优先遍历算法示例

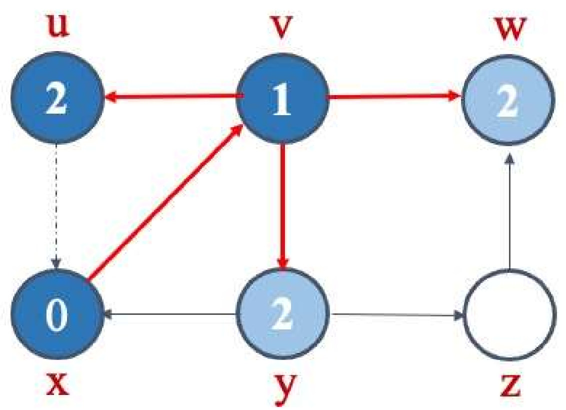


queue: u, w, y

visited	T	T	T	T	T	F
prev	v	x	-1	v	v	
	u	v	x	y	w	z

广度优先遍历算法示例

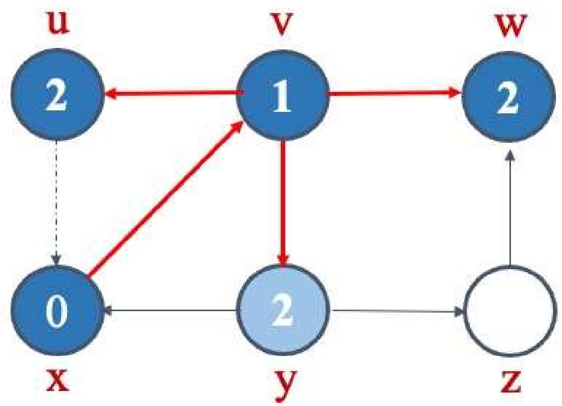
queue: w, y



visited	T	T	T	T	T	F
prev	v	x	-1	v	v	
	u	v	x	y	w	z

广度优先遍历算法示例

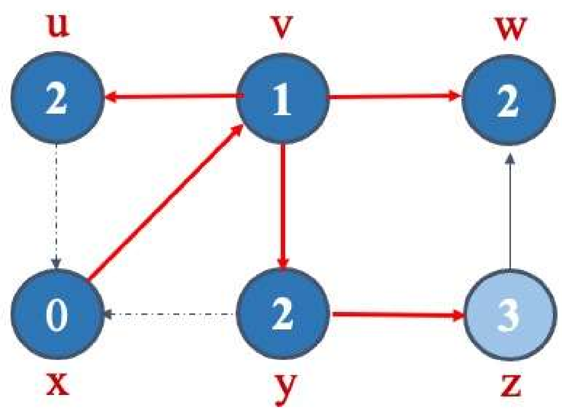
queue: y



	u	v	x	y	w	z
visited	T	T	T	T	T	F
prev	v	x	-1	v	v	

广度优先遍历算法示例

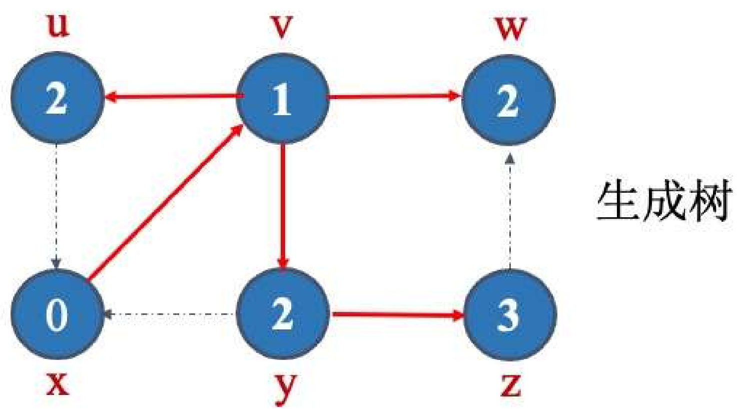
queue: z



visited	T	T	T	T	T	T
prev	v	x	-1	v	v	y
	u	v	x	y	w	z

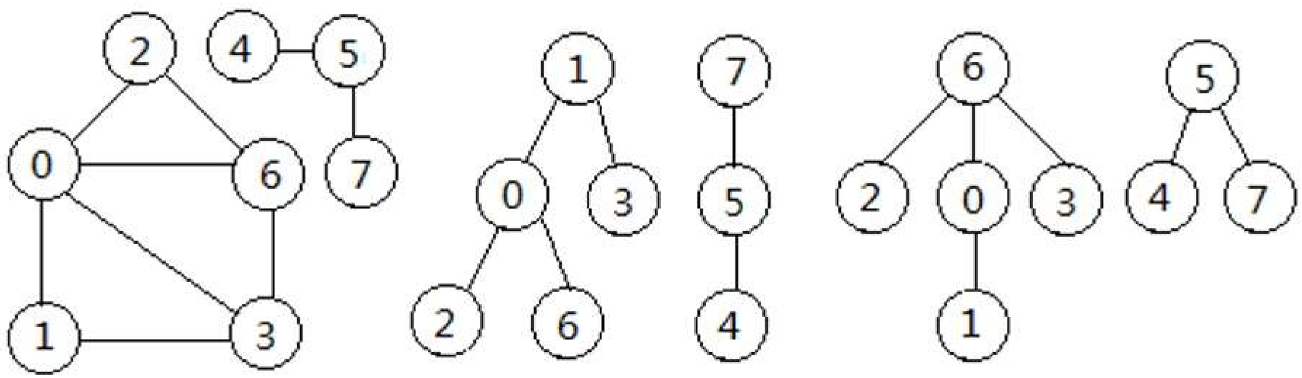
广度优先遍历算法示例

queue:



visited	T	T	T	T	T	T
prev	v	x	-1	v	v	y
	u	v	x	y	w	z

广度优先遍历



(a) G18　　(b) G18的广度优先遍历　　(c) G18的广度优先遍历

1. 广度优先遍历结果是不唯一的。
2. 它不是一个**递归过程**：由对图的遍历，转向对点的访问。

多选题 1分

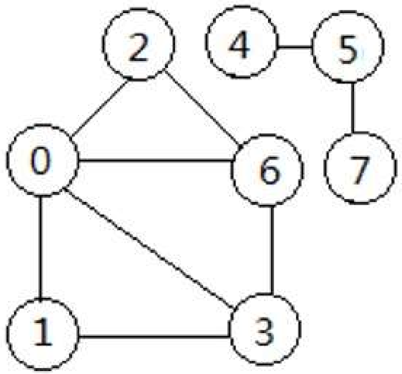
下面哪些序列可以是对右图广度优先遍历的结果?

A <0, 6, 2, 3, 1, 7, 5, 4>

B <2, 6, 0, 1, 3, 5, 4, 7>

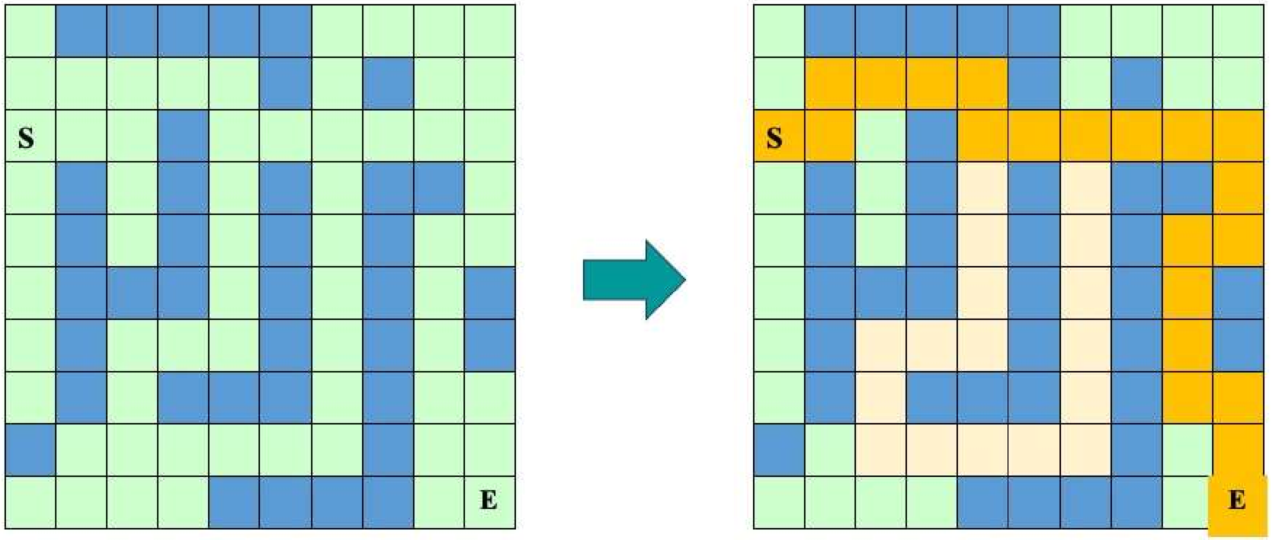
C <5, 4, 7, 3, 0, 1, 2, 6>

D <7, 5, 4, 0, 1, 3, 2, 6>



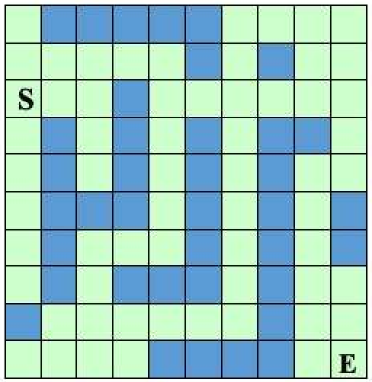
广度优先遍历的应用：走迷宫

问题描述：表示迷宫的二维数组M[1..n, 1..m], 其中M[i][j]=0表示方格(i, j)可以通过，而M[i][j]=1表示该方格设有障碍，不能通过($1 \le i \le n, 1 \le j \le m$)。从标有S的方格（起点）出发，每步可以移到当前位置上下左右相邻的、且可通过的方格，查找走到终点（标有E的方格）的最短路径。



高等教育出版社

深度优先遍历的应用：走迷宫



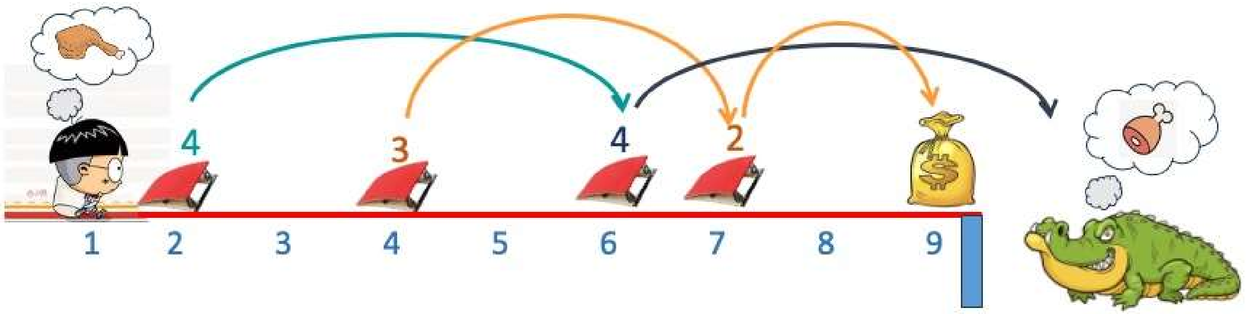
算法: BFS-FindMaze($M, n, m, s_x, s_y, t_x, t_y, visited$)
输入： 二维数组M[1..n, 1..m], 起点位置(s_x, s_y), 终点(t_x, t_y)
输出： 查找从起点至终点的最短路径
关键数据结构： adj ← {(0, -1), (0, 1), (-1, 0), (1, 0)} //上下左右的（相对）位置

```
visited[sx][sy] ← true   //标记起点
EnQueue(queue, (sx, sy) )   //起点入队（结构体或类）
prev[sx][sy] ← -1
while IsEmpty(queue) = false do
|  (x, y) ← DeQueue(queue)   //取出队首位置
|  if x = tx 且 y = ty then
|  |  break   //走到终点，结束查找（? ）
|  end
|  for k ← 0 to 3 do   //依次查找上下左右相邻位置
|  |  nx ← x + adj[k][0]
|  |  ny ← y + adj[k][1]   //相邻位置有效、非障碍且未被访问过
|  if 1 ≤ nx ≤ n 且 1 ≤ ny ≤ m 且 M[nx][ny] ≠ 1 且 visited[nx][ny] = false then
|  |  prev[nx][ny] ← k   //记录到达<nx. ny>的最短路径
|  |  visited[nx][ny] ← true
|  |  EnQueue(queue, (nx, ny) )
|  end
end
```

- 时间复杂度：O(mn)
- 空间复杂度：O(mn)

广度优先遍历的应用：跳吧！小明

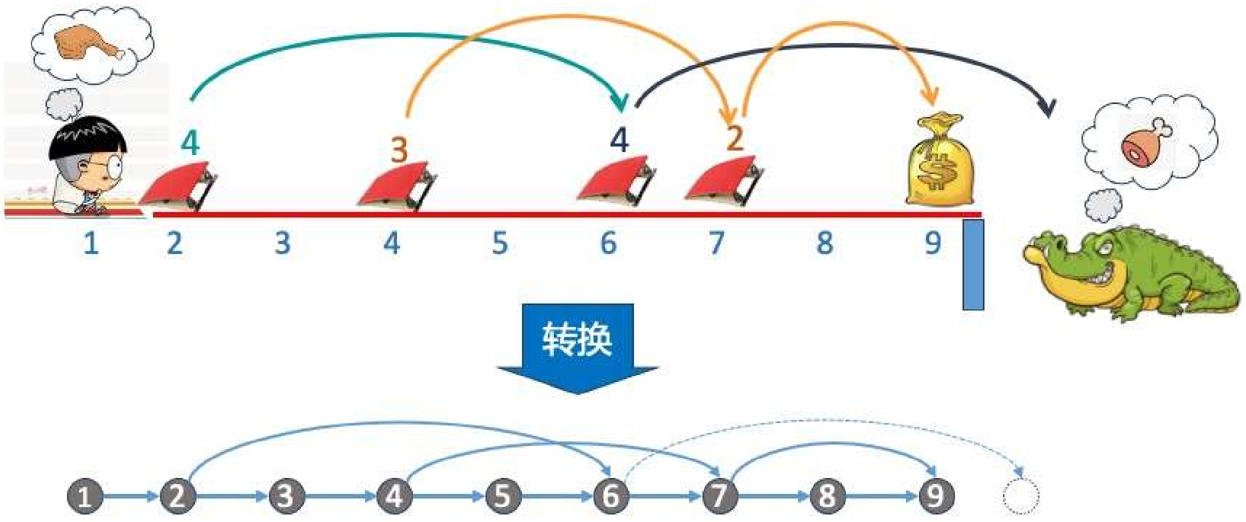
问题描述：有一条长度为N的直路，小明从左端位置1出发，每步走1的距离，一直走到终点N为止。在直线的某些位置“设置”有跳板，如果踩跳板（也可不踩直接通过），能够向前跳跃一定的距离，求刚好到达终点N的最少步数。



- 路径0：1 -> 2 -> 3 -> 4 ->5 -> 6 -> 7 -> 8 -> 9
- 路径1：1 -> 2 -> 3 -> 4 ---> 7 --> 9 步数5
- 路径2：1 -> 2 ----> 6 ---->10 。。。 小明。。终
- 路径3：1 -> 2 ----> 6 -> 7 --> 9 步数4：最短路径

广度优先遍历的应用：跳吧！小明

问题描述：有一条长度为N的直路，小明从左端位置1出发，每步走1的距离，一直走到终点N为止。在直线的某些位置"设置"有跳板，如果踩跳板（也可不踩直接通过），能够向前跳跃一定的距离，求刚好到达终点N的最少步数。



求从起点1到终点N的最短路径

广度优先遍历的应用：跳吧！小明

算法: BFS-XiaoMing(*n, Jump, visited*)
输入： 路径长度n, Jump[1..n], Jump[i]表示第i个位置上的跳板的弹跳距离，Jump[i]=0则无跳板
输出： 计算小明从起点1走到终点n的最少步数（时间）

```
visited[1] ← true   //标记起点
prev[1] ← -1
step[1] ← 0   //从起点开始，步数是0
EnQueue(queue, 1)
while IsEmpty(queue) = false do
|  p ← DeQueue(queue)   //取出队首位置
|  if p = n then
|  |  break          //走到终点，结束计算
|  end
|  next_p ← p + Jump[p]          //先试踩跳板
|  for i ← 1 to 2 do
|  |  if next_p ≤ n 且 visited[next_p] = false then   //不能越界！
|  |  |  visited[next_p] ← true
|  |  |  prev[next_p] ← p
|  |  |  step[next_p] ← step[p] + 1   //到达next_p的最少步数
|  |  |  EnQueue(queue, next_p)
|  |  end
|  |  next_p ← p + 1          //再试直接通过
|  end
```

小明

- 时间复杂度：O(n)
- 空间复杂度：O(n)

深度和广度优先遍历结果特点

1. 图的深度优先遍历和广度优先遍历既适用于有向图，也适用于无向图。
2. 遍历中，已访问过的邻接点将不再被访问，故遍历结果只能是树形结构。

无回路

深度和广度优先遍历比较

1. 深度优先遍历的特点是“一条路跑到黑”，如果面临的问题是能找到一个解就可以，深度优先遍历一般是首选。其搜索深度一般比广度优先遍历要搜索的宽度小很多。
2. 广度优先遍历的特点是层层扩散。如果面临的问题是要找到一个距离出发点最近的解，那么广度优先遍历是最好的选择。
3. 广度优先遍历需要程序员自己写个队列，代码比较长。而且这个队列要能同时存储一整层顶点，如果是一棵满二叉树，每层顶点的个数是呈指数级增长的，所以耗费的空间会比较大。

无向图的连通性

重要性质:

如果无向图是连通的，那么选定图中任何一个顶点，从该顶点出发，通过遍历，就能到达图中其他所有顶点。

方法是:

只需在以上的深度优先、广度优先遍历实现算法中增加一个计数器，记录外循环体中，进入内循环的次数，根据次数是否可以判断出该图是否连通？如果不连通有几个连通分量？每个连通分量包含哪些顶点?

无向图的连通性

算法7-15: 图的连通性判断 IsConnect(*graph*)

输入： 图*graph*

输出： 图*graph*的连通性。若不连通，还输出连通分量的数量。

```
1.  for v ← 0 to graph.n_verts-1 do   //初始化各顶点的访问标志为未访问
2.      visited[v]←false
3.  end
4.  count ← 0   //计数连通分量
5.  for v ← 0 to graph.n_verts-1 do
6.  | if visited[v] = false then
7.  | |  count ← count + 1   //连通分量数目加1
8.  | |  BFS(graph, v, visited)
9.  | end
10. end
```

高等教育出版社

无向图的连通性

```
if count =1 then
|  ret ← true    //只有一个连通分量，图连通
else
|  print count
|  ret ← false
end
return ret
```

时间复杂度 O（n+m）

六度空间理论

1967年哈佛大学心理学教授-斯坦利·米尔格拉姆（Stanley Milgram），设计并实施了一次连锁信件实验。

具体做法：

将设计好的信件随机发送给居住在内布拉斯加州的160个人，信中写上了一个波士顿股票经纪人的名字，要求每个收信人收到信后，再将这个信寄给自己认为比较接近该股票经纪人的朋友，要求后面收到信的朋友也照此操作。

最后发现，有信件在经历了不超过六个人之后就送到了该股票经纪人手中。

六度空间理论

由此提出了“小世界理论”，也称“六度空间理论”或“六度分隔理论（Six Degrees of Separation）”。

该理论假设：世界上所有互不相识的人只需要很少的中间人就能建立起联系，具体说来就是，在社会性网络中，你和世界上任何一个陌生人之间所间隔的人不会超六个，即最多通过六个人你就能够认识任何一个陌生人。

也就是说：马云是我二舅的表弟的同学的。。。这句话真的没骗人！

该理论目前仍然是数学界的的一大猜想，它从来没有得到过严谨的数学证明。

六度空间理论的验证方法

- 图中顶点代表人，顶点之间的边代表人与人之间相识。
- 根据六度空间思想，该理论转化为无向图中任何两点之间的最短距离不会超过六。

六度空间理论的验证算法

<u>**算法7-16:** 验证六度空间理论SixDegreesOfSeparation(*graph,v*)</u>

输入： 图*graph*，起始顶点*v*

输出： 图中以顶点*v*为起始顶点，最短距离不大于6的顶点个数和图中顶点总数的比值

1. **for** *v* ← 0 **to** *graph.n_verts*-1 **do** //初始化各顶点的访问标志为未访问
2. | *visited*[v] ← *false*
3. **end**
4. *count* ← 0
5. InitQueue(*ver_queue*) //结点队列
6. InitQueue(*level_queue*) //结点所在层数队列
7. EnQueue(*ver_queue, v*)
8. EnQueue(*level_queue, 0*)

六度空间理论的验证算法

```
while IsEmpty(ver_queue) = false do
| cur_ver ←DeQueue(ver_queue)
| cur_level ←DeQueue(level_queue)
| if cur_level ≤ 6 then
| | if visited[cur_ver] = false then  //未访问过该结点则对它加访问标志
| | | visited[cur_ver] ← true
| | | count ← count + 1
| | | p ←graph.ver_list[cur_ver].adj //向cur_ver的下一层搜索
| | | while p ≠ NIL do
| | | | if visited[p.dest] = false then
| | | | | EnQueue(ver_queue, p.dest)
| | | | | EnQueue(level_queue, cur_level+1)
| | | | end
```

高等教育出版社

六度空间理论的验证算法

```
| | | | p ← p.next
| | | end
| | end
| else //已完成6层搜索，算法结束
| | break
| end
end
return count/graph.n_vers
```

无向连通图的BFS 算法，时间复杂度 O（n+m）

有向图的连通性

- 有向图的强连通分量问题解决起来比较复杂。
- 对一个强连通分量来说，要求每一对顶点间相互可达。
- 以上的深度、广度优先遍历都只是计算了单向路径。

有向图的连通性

依然可利用有向图的深度优先遍历DFS，通过以下算法获得：

1. 对有向图G进行深度优先遍历，按照遍历中回退顶点的次序给每个顶点进行编号。最先回退的顶点的编号为1，其它顶点的编号按回退先后逐次增大1。
2. 将有向图G的所有有向边反向，构造新的有向图Gr。
3. 选取未访问顶点中编号最大的顶点，以该顶点为起始点在有向图Gr上进行深度优先遍历。如果没有访问到所有的顶点，再次返回3，反复如此，直至所有的顶点都被访问到。

深度优先遍历算法的扩展

算法7-11: 按深度优先遍历图中结点 DFS(*graph*)

输入： 图*graph*

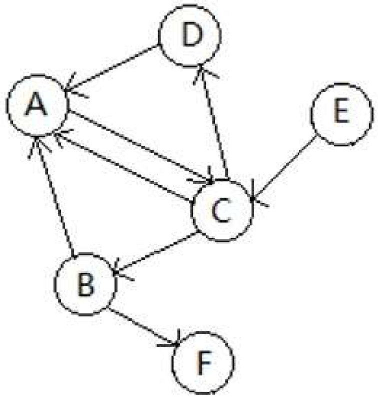
输出： 1. 各结点*u*第一次被“发现”的时刻 ***dfn***[*u*]
2. 各结点*u*遍历结束的时刻 ***fin***[*u*]
（1、2 可用于割点、强连通成分等）
3. 各结点*u*在深度优先遍历次序中的前驱结点 ***prev***[*u*] ---路径、生成树等

关键数据结构： 顺序表visited

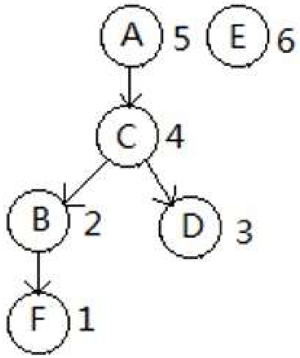
全局变量： time 时间戳

按 *fin*[*u*] 的升序对每个顶点进行编号

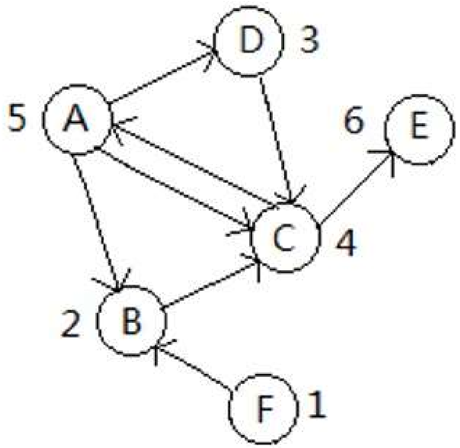
有向图的连通性



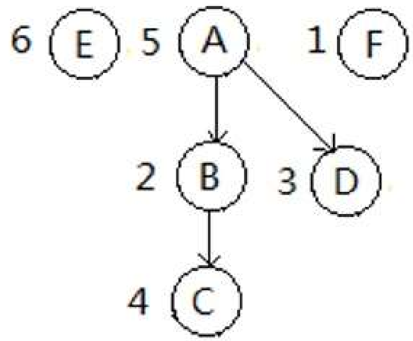
(a) G19



(b) G19的深度优先遍历



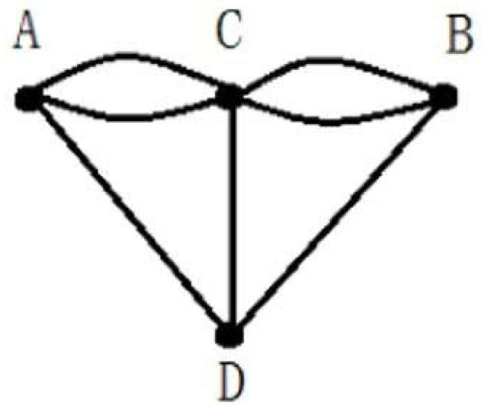
(c) G19的边逆向得Gr



(d) Gr的深度优先遍历

哥尼斯堡七桥问题求解

1. 18世纪数学家欧拉将著名的格尼斯堡七桥问题抽象为以下数学问题：从图中的任意一个顶点出发是否存在一条路径，它能经过每条边一次且仅经过一次后回到出发顶点。



2. 欧拉解决了七桥问题、给出了解决相关问题的欧拉定理。

相关术语

欧拉路径： 如果图中的一条路径经过了图中每条边一次且仅一次，这条路径称欧拉路径。

欧拉回路： 如果一条欧拉路径的起点和终点相同，这条路径是一个回路，称欧拉回路。

欧拉图： 具有欧拉回路的图称欧拉图(简称**E图**)，

半欧拉图： 具有欧拉路径但不具有欧拉回路的图称半欧拉图。

一笔画： 从图中一个顶点出发进行深度优先搜索，一直往前走，没有任何回溯，观察是否有一条路径能走遍图中所有的边且每条边都只走了一次，这就是一笔画问题。

欧拉路径和欧拉回路的应用场景

- **数学**：网络分析、社交网络分析、电子电路设计等
- **计算机科学**：图形算法、网络优化、数据结构等
- **物理学**：流体力学、电磁学、量子力学等
- **化学**：分子结构分析、化学反应分析等

欧拉定理

- 一个无向连通图中，如果度为奇数的顶点超过了2个，则欧拉路径是不存在的。
- 一个无向连通图中，如果除了两个顶点的度是奇数而其他顶点的度都是偶数，则从一个度为奇数的顶点出发一定能找到一条经过每条边一次且仅一次的路径回到另外一个度为奇数的顶点。
- 一个无向连通图中，如果顶点的度都是偶数，则从任意一个顶点出发都能找到经过每条边一次且仅一次并回到原来的顶点的路径（回路）。

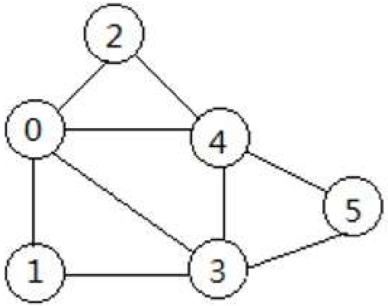
思考：如果结点的度都是偶数，那么从任一结点出发，每次选择任意一条未经过的边走下去，一定能回到起点且回路一定是欧拉回路?

高等教育出版社

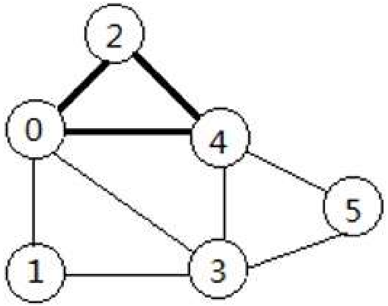
欧拉回路求解方法

1. 任选一个顶点v，从该顶点出发开始深度优先搜索，搜索路径上都是由未访问过的边构成，搜索中访问这些边（加标记！），直到回到顶点v且v没有尚未被访问的边（**思考**：如果还有边，怎么操作？），此时便得到了一个回路（非简单回路！），此回路为当前结果回路。
2. 在搜索路径上另外找一个尚有未访问边的顶点，继续如上操作，找到另外一个回路，将该回路拼接在当前结果回路上，形成一个大的、新的当前结果回路。
3. 如果在当前结果回路中，还有中间某结点有尚未访问的边，回到2；如果没有任何中间顶点尚余未访问的边，访问结束，当前结果回路即欧拉回路。

欧拉回路求解方法

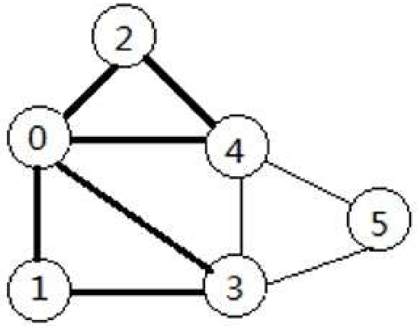


(a)G20

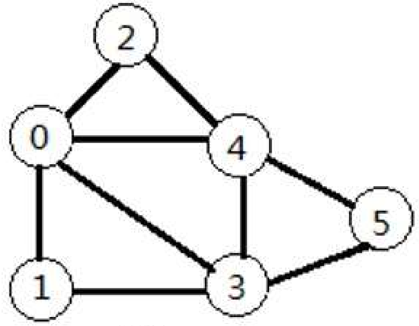


(b)

第1个回路：2->0->4->2
当前结果回路：2->0->4->2



(c)



(d)

第2个回路：0->1->3->0
当前结果回路：2->0->1->3->0->4->2

第3个回路：3->4->5->3
当前结果回路：2->0->1->3->4->5->3->0->4->2

欧拉回路求解算法

<u>**算法7-17:** 从给定点出发获得一条回路 GetCircuit(*graph*, *start*)</u>
输入： 无向图*graph*，起始顶点*start*
输出： 图*graph*中从*start*出发的一条回路的单链表

1. *new_node* ← **new** EulerNode(*start*, NIL) //从*start*顶点开始，构造回路的第一个结点
2. *circuit* ← **new** CircPtrNode
3. *circuit.first* ←*new_node*   //指向单链表的头结点
4. *circuit.last* ←*new_node*   //指向单链表的尾结点
5. *p* ← *graph.ver_list*[*start*].*adj*
6. *head* ← *start*

欧拉回路求解算法

```
while p ≠ NIL 且 p.dest ≠ start do
|  tail ← p.dest
|  RemoveEdge(graph,head, tail)
|  RemoveEdge(graph,tail, head)      从邻接表中删除经过的边（保证只走一次！）
|  new_node ←new EulerNode(tail, NIL)
|  circuit.last.next ← new_node
|  circuit.last ← circuit.last.next
|  p ←graph.ver_list[tail].adj
|  head ← tail
end
return circuit
```

欧拉回路求解算法

算法7-18: 求欧拉回路 EulerCircle(*graph*)

输入： 无向连通图*graph*

输出： 图*graph*的一个欧拉回路；若不存在，则返回NIL

```
for v ← 0 to graph.n_verts-1 do  //计算每个顶点的度，判断是否存在欧拉回路
|  p ← graph.ver_list[v].adj
|  degree[v] ← 0
|  while p ≠ NIL do
|  |  degree[v] ← degree[v] + 1
|  |  p ←p.next
|  end
|  if degree[v]%2=1 then
|  |  return NIL //存在度为奇数的顶点，该无向连通图无欧拉回路
|  end
end
```

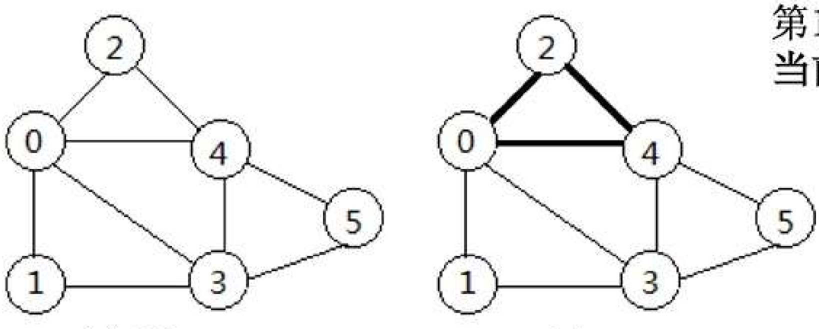
欧拉回路求解算法

```
tmp_graph ← clone(graph)  //复制原图的副本（因为要删边）
circuit ← GetCircuit(tmp_graph, 0) //从0下标顶点开始，构造第一个当前结果回路
p ← circuit.first.next //寻找新的回路，并入当前结果回路中
while p≠NIL do
|  if tmp_graph.ver_list[p.ver].adj ≠ NIL then //找到第1个起始顶点
|  |  next_circuit ←GetCircuit(tmp_graph, p.ver)
|  |  next_circuit.last.next ←p.next
|  |  p.next ←next_circuit.first.next
|  |  delete next_circuit.first
|  end
|  p ← p.next
end
return circuit
```

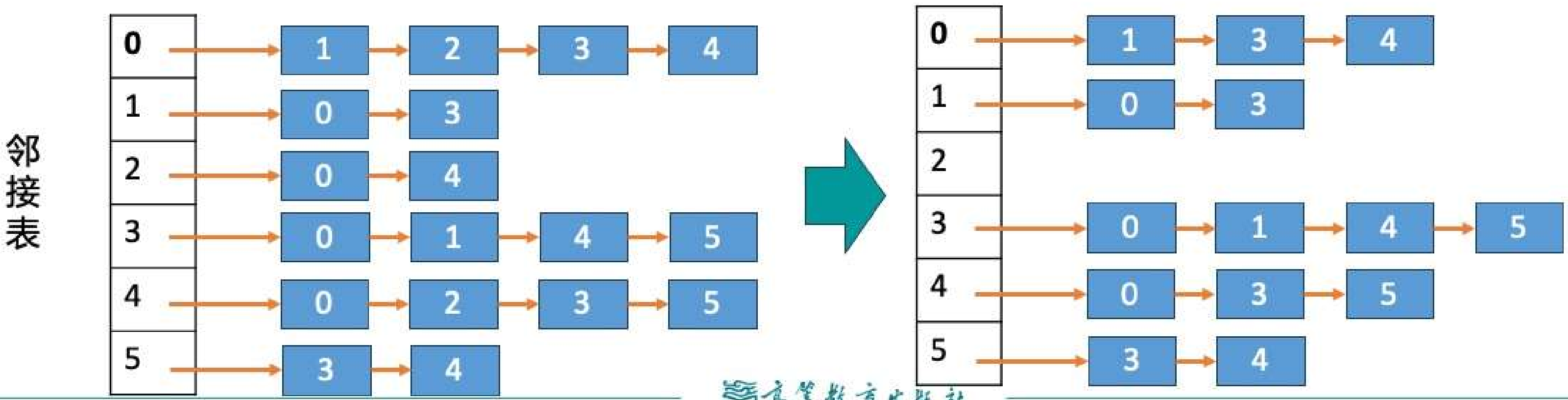
访问到每条边一次，顶点共m次，时间复杂度 O（m）

欧拉回路求解方法

第1个回路：2->0->4->2
当前结果回路：2->0->4->2



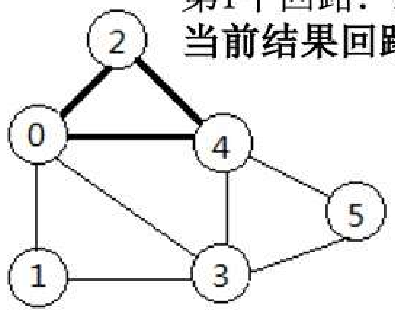
(a)G20 (b)



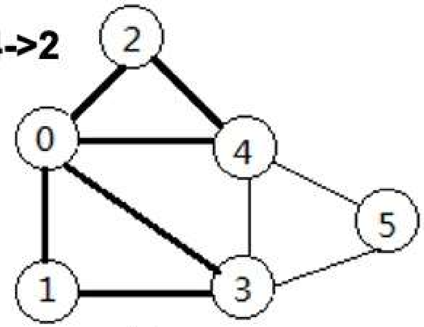
欧拉回路求解方法

第1个回路：2->0->4->2

当前结果回路：**2->0->4->2**



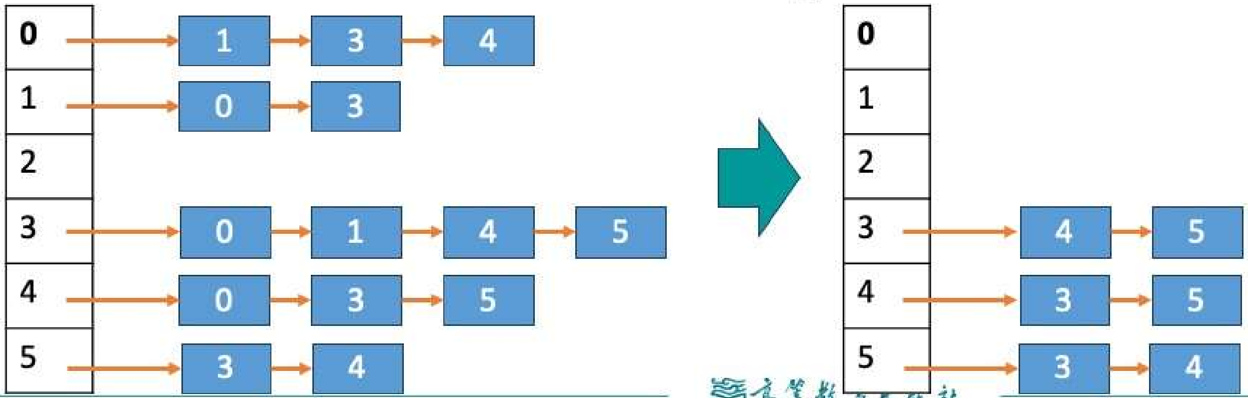
(b)



(c)

第2个回路：0->1->3->0

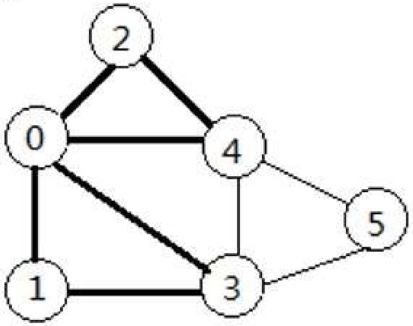
当前结果回路：**2->0->1->3->0->4->2**



欧拉回路求解方法

第2个回路：0->1->3->0

当前结果回路：2->0->1->3->0->4->2

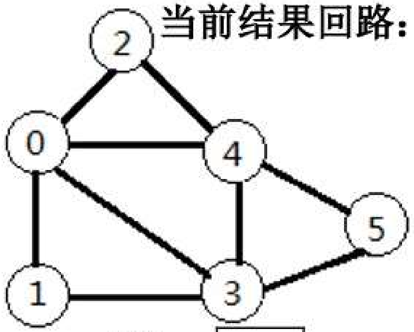


（c）

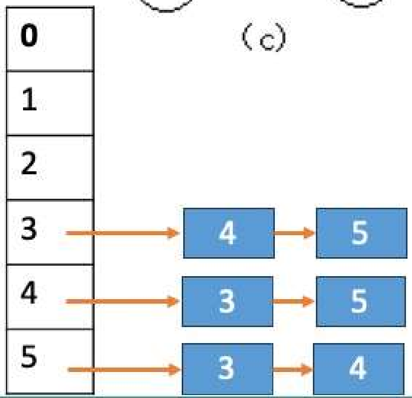
第3个回路：3->4->5->3

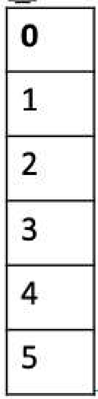
当前结果回路：2->0->1->3->4->5->3->0->4->2

欧拉回路



（d）





再无边可走!

双连通分量

相关术语：

点割集： 在一个无向图G=(V,E)中，若存在一个顶点集合W，从G中删除W中的所有顶点以及W中所有顶点相关联的边之后，图的连通分量增多，则这个顶点集合W称**点割集**。

割点： 当W中只含有一个顶点时，这个顶点称**割点**。

特殊地：当G是一个无向连通图时，删除割点后，得到的图不再连通(含两个或两个以上连通分量)。

双连通分量

边割集： 在一个无向连通图G=(V,E)中，若存在一个边的集合F，删除F中的所有边后，得到的图不再连通，则这个边集合F称**边割集**。

割边： 当F中只含有一条边时，这条边称**割边（或桥）**。

边双连通图： 若一个无向图连通图中去掉任意一条边都不会改变此图的连通性，即不存在桥，则称该无向图为边双连通图。

边双连通分量： 图中的每一个极大边双连通子图称该无向图的边双连通分量。

无向连通图的割点和割边

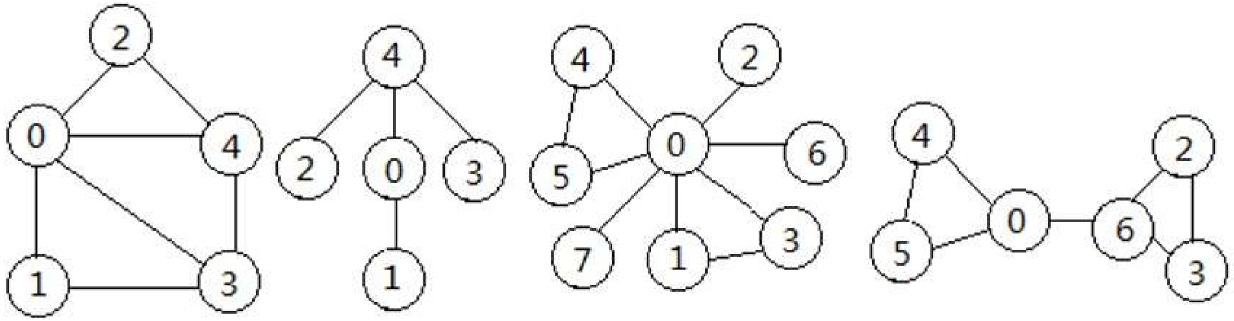


图 G18-1(a) 图 G18-1(b) 图 G18-1(c) 图 G18-1(d)

(a)割点：无
(b)割点：4、0
(c)割点：0
(d)割点：0、6

(a)割边： 无
(b)割边： (2,4) (0,4) (3,4) (0,1)
(c)割边： (0,2) (0,6) (0,7)
(d)割边： (0,6)

Tarjan算法

Robert Endre Tarjan是一位计算机科学家，他发明了很多算法，统称为Tarjan算法。

Tarjan算法最著名的有三个，分别是求解：

1）有向图的强连通分量，

2) 无向图的双连通分量，

3）最近公共祖先问题。

基于求无向图的**双连通分量**算法也解决了求无向连通图的**割点**和**割边**问题。