第7章 图

郑新、徐鹏飞、李健
北京师范大学人工智能学院
2025-2026学年 第一学期

考核要点

- 考核大纲
  - 图的基本概念;
  - 图的存储及基本操作;
  - 图的遍历和连通性
- 复习要点
  - 掌握图的基本概念、基本性质、存储结构（邻接矩阵、邻接表、邻接多重表和十字链表）（A）
  - 重点掌握基于存储结构上的遍历操作(深度优先搜索DFS与广度优先搜索BFS)
  - 掌握无向图的连通性（A）；理解有向图的连通性（B）

提纲

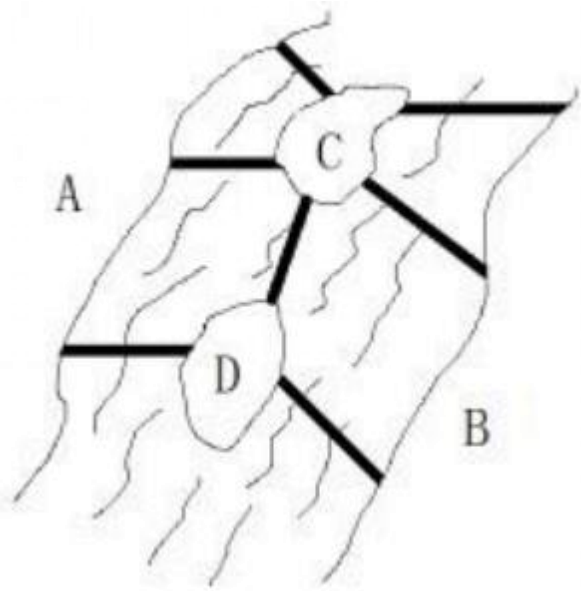
- 7.1 问题引入及求解
- 7.2 图的定义与结构
- 7.3 图的存储实现
- 7.4 图的遍历
- 7.5 图的连通性
- 7.6 图的应用
- 7.7 拓展延伸*
- 7.8 应用场景

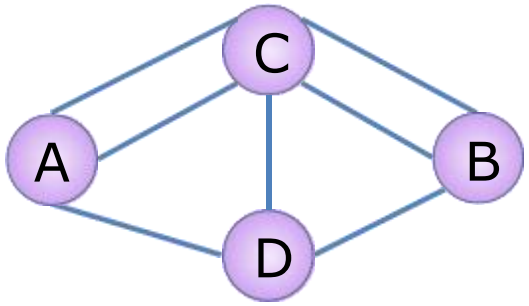
7.1 问题导入

问题：18世纪的哥尼斯堡，一条河流穿城而过，城市除被一分为二外，还包含了河中的两个小岛，河上有七座桥把这些陆地和岛屿联系了起来，可否从一个陆地或岛屿出发，一次经过全部的七座桥且每座桥只走一遍，最后还能回到出发点?

问题分析：2个问题，如何判断是否有解？如果有解，如何找到解？

问题抽象：将陆地和岛屿抽象成元素（顶点），桥抽象成元素之间的关系（边）。





问题转化：是否存在从任意一个顶点出发，经过每条边一次且仅一次，最后回到该顶点的路径（一笔画问题）。

教学工具 → 图

提纲

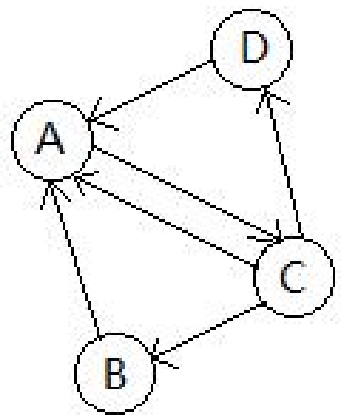
- 7.1　问题引入及求解
- 7.2　图的定义与结构
- 7.3　图的存储实现
- 7.4　图的遍历
- 7.5　图的连通性
- 7.6　图的应用
- 7.7　拓展延伸*
- 7.8　应用场景

7.2 图的定义

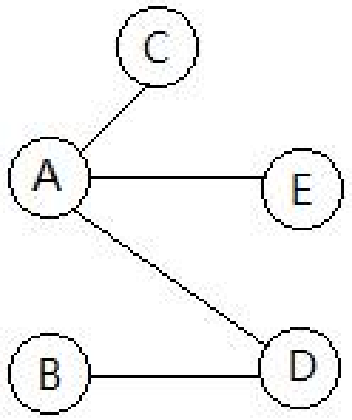
图：可以用一个二元组G = (V,E)表示，其中V是顶点的非空集合，E是两个顶点间边（弧）的集合。

G1是由顶点集合V = {A,B,C,D}和边的集合E={<B,A>，<A,C>，<C,A>，<C,D>，<D,A>，<C,B>}构成。

G2是由顶点集合V ={A,B,C,D,E}和边集合E ={ (A,C)，(A,E)，(D,B)，(D,A)}构成。



G1



G2

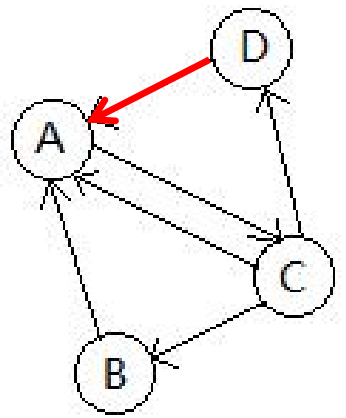
图的术语

有向边：边带有方向，用带尖括号的顶点对来表示，如<D,A>，表示由D射向A的边，A为**弧头**，D为**弧尾**。

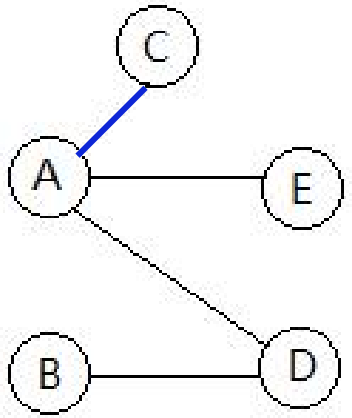
有向图：由顶点集和有向边集合组成的图。G 1 就是一个有向图。

无向边：边不带有方向，用带圆括号的顶点对来表示，如 (C,A) ，表示C和A之间有条边。

无向图：由顶点集和无向边集合组成的图。G 2 就是一个无向图。



G1



G2

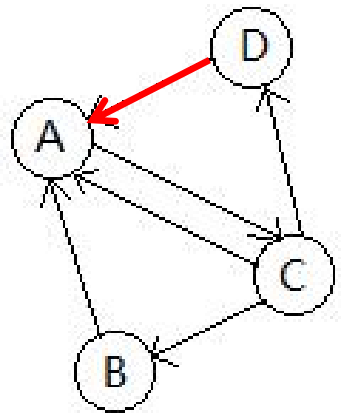
图的术语

邻接：图的顶点间有边相连，称顶点间有邻接关系。(vi,vj)是一条无向边，称vi和vj邻接、vj和vi邻接、边(vi,vj)邻接于顶点vi和vj；<vi,vj>是条有向边，称vi**邻接到**vj、或vj和vi邻接、边<vi,vj>邻接于顶点vi和vj。

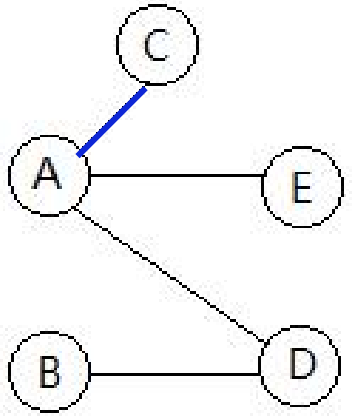
出度：有向图中一个顶点的**出度**是指由该顶点射出的有向边的条数。

入度：有向图中一个顶点的**入度**是指射入该顶点的有向边的条数。

度：无向图中一个顶点的**度**是指邻接于该顶点的边的总数。



G1



G2

图的术语

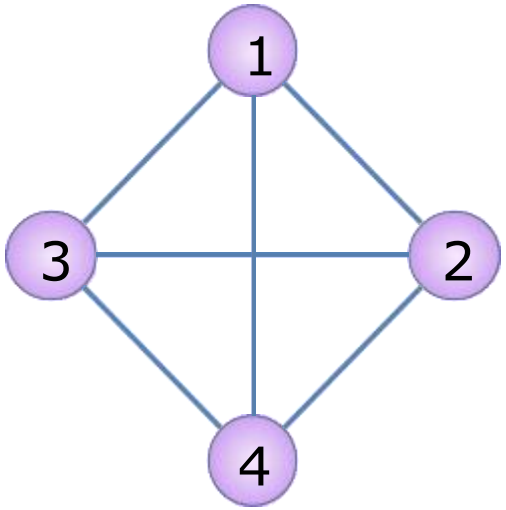
无向完全图：当无向图中边的条数达到最大，为 n（n－1）／2 时的图。

有向完全图：当有向图中边的条数达到最大，为 n（n－1）时的图。

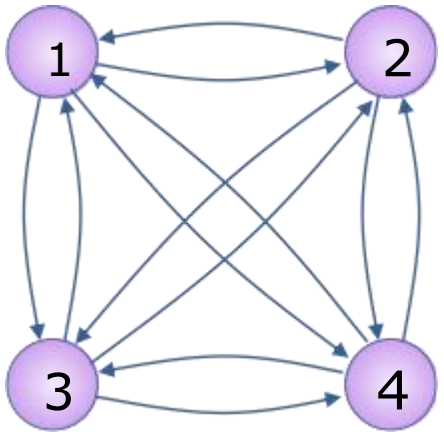
加权有向图：边上带有权重的有向图。

加权无向图：边上带有权重的无向图。

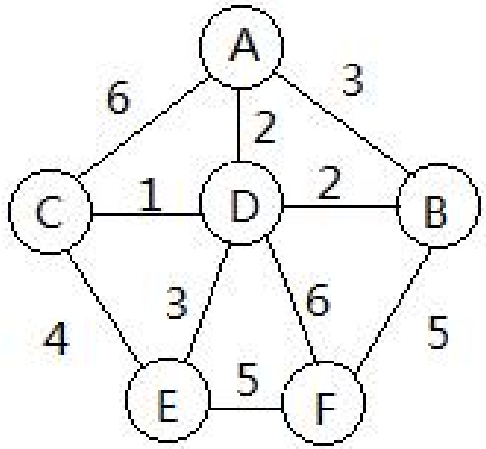
网络：加权有向图和加权无向图，统称为网络。



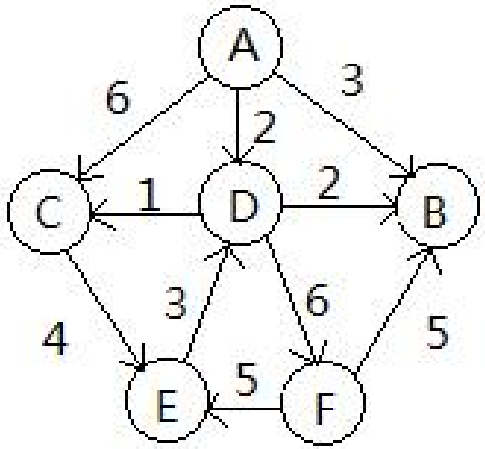
无向完全图



有向完全图



加权无向图G3



加权有向图G4

图的术语

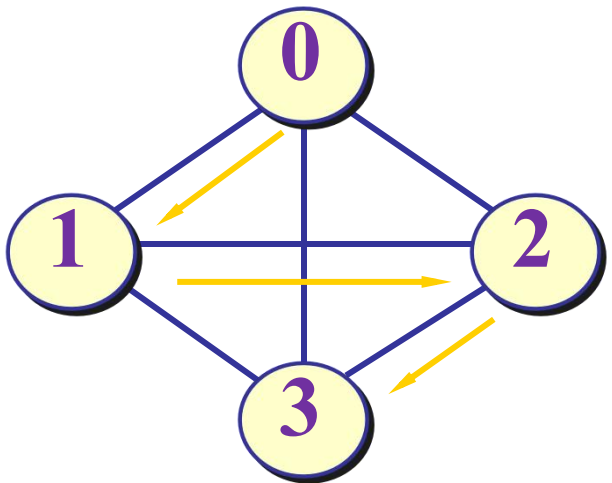
路径：如果可以从顶点v_i出发经过若干条无向边或者有向边到达顶点v_j，称顶点v_i到顶点v_j之间存在着一条路径。

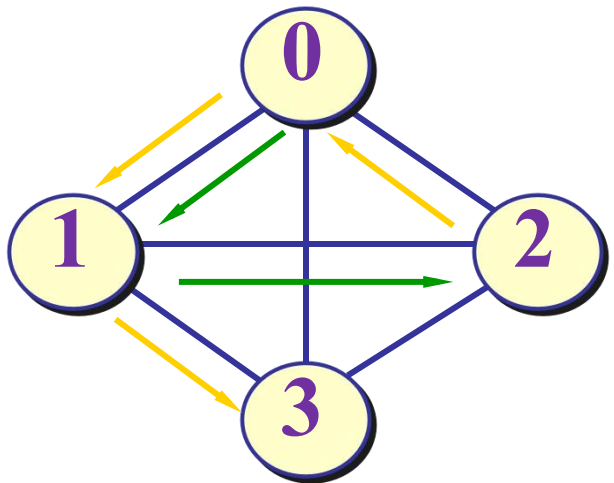
路径的长度：是顶点v_i到顶点v_j之间的这条路径上无向边或有向边的条数。

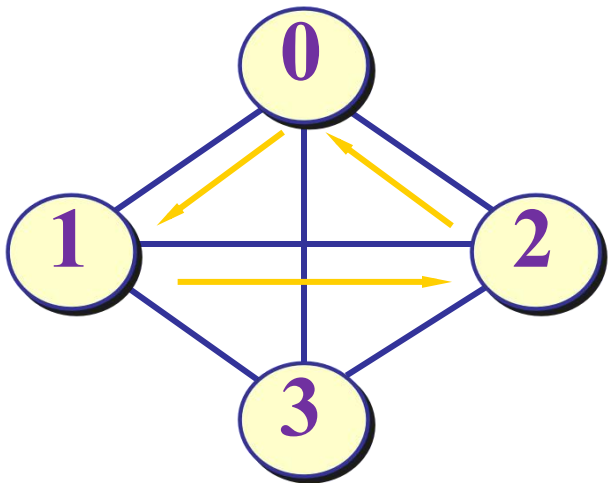
如果边上有权重，路径长度也可以用路径上所有边的权重之和来表示。

简单路径：一条路径上除了第一个顶点和最后一个顶点可能相同之外，其余各顶点都不相同。

简单回路或简单环：简单路径上第一个顶点和最后一个顶点相同。

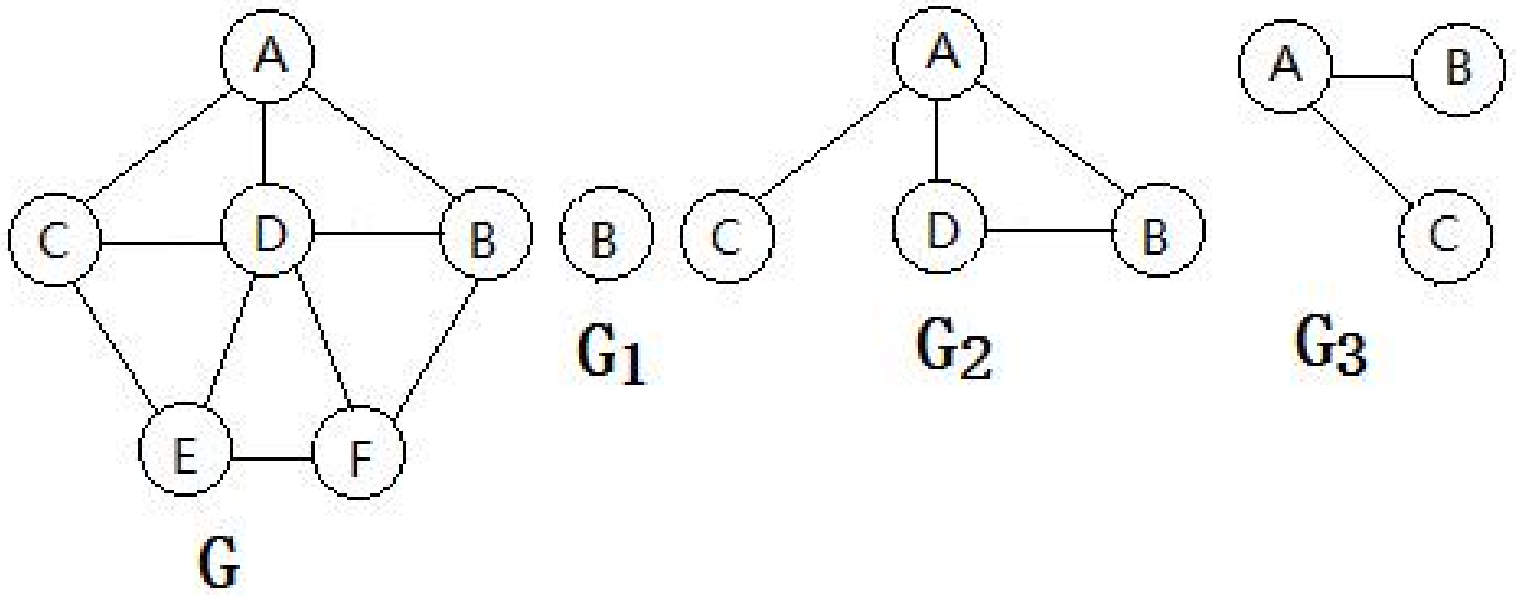






图的术语

子图：假设有两个图G = (V,E)，G’ = (V’,E’)，且V’是V的子集，E’是E的子集，称G’是G的子图



图G1、图G2、图G3均是图G的子图。

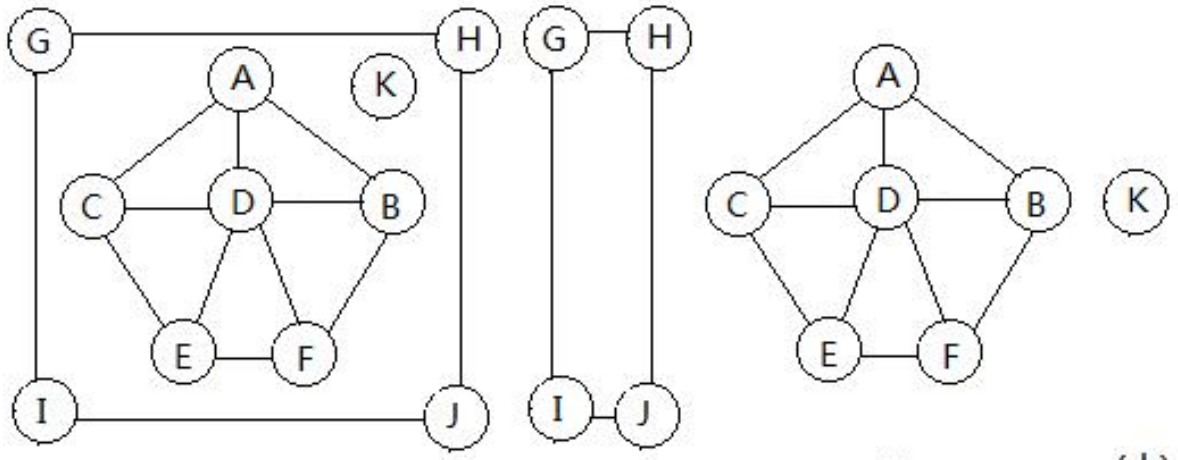
图的术语

连通：在一个图中，如果顶点v_i到v_j之间有路径存在，称顶点v_i到v_j是连通的。

连通图：在一个无向图中，如果任意两个顶点对之间都是连通的，称该无向图G是连通图。

极大连通子图：将该子图外的任意一个顶点增加进子图都会造成子图不连通，且该子图包含了其中顶点间所有的边，该子图称极大连通子图。

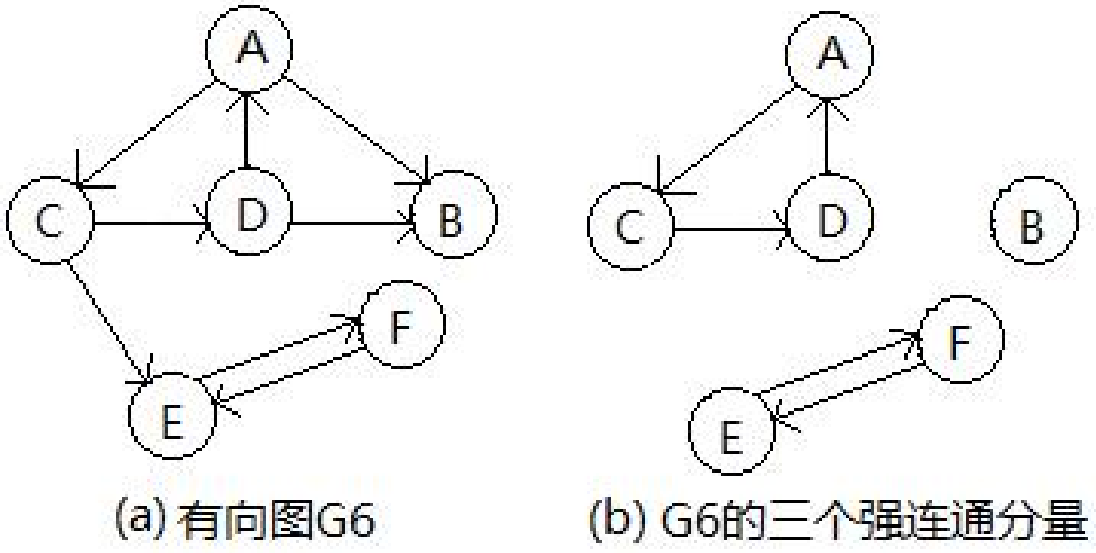
连通分量：无向图的极大连通子图称连通分量。



图的术语

强连通图：在一个有向图G中，如果任意两个顶点对之间都是连通的，称有向图G是强连通图。

强连通分量：有向图的极大连通子图，称强连通分量。

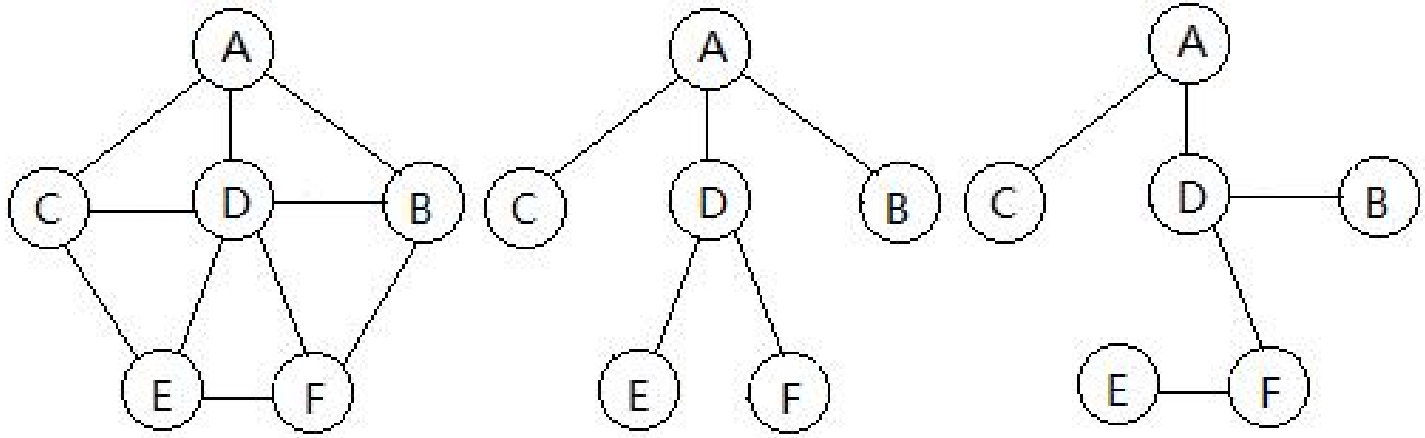


(a) 有向图G6　　(b) G6的三个强连通分量

图的术语

生成树：连通图的**极小连通子图**，该子图包含连通图的所有n个顶点，但只含它的n-1条边。如果去掉一条边，这个子图将不连通；如果增加一条边，必存在回路。

不唯一性：一个连通图的生成树并不保证唯一。



(a) G7　　(b)G7的两个生成树

线性、树、图结构的比较

- 线性结构中，每个元素只有一个直接前驱和一个直接后继。
- 树形结构中，每个数据元素有一个直接前驱，但可以有多个直接后继。
- 图形结构中，数据元素之间的关系是任意的。每个数据元素可以和任意多个数据元素相关，有任意多个直接前驱和直接后继。在无向图中，甚至是互为前驱后继。

图结构的ADT

ADT Graph {

数据对象:

$\{v_i | v_i \in \text{ElemSet}, i=1,2,3,\cdots\cdots n, n>0\}$ 或 Φ; ElemSet为顶点集合。

数据关系:

$\{<v_i, v_j>$或$(v_i, v_j) | v_i、v_j \in \text{ElemSet}$, 且$P(v_i, v_j)$, i, j=1,2,3,……n$\}$,

其中：$<v_i, v_j>$表示从顶点v_i到顶点v_j的一条边，

(v_i, v_j)表示顶点v_i与顶点v_j互连，

$P(v_i, v_j)$定义了$<v_i, v_j>$或(v_i, v_j)的意义或信息。

基本操作:

InitGraph(*graph*，*kMaxVertex*，*no_edge_value*，*directed*)：初始化一个空的图*graph*。其中*kMaxVertex*是最多可能的顶点数；*no_edge_value*是当顶点间不存在边时，在图中给顶点关系赋予的权值；*directed* 为**true**时图是有向的，为**false**时图是无向的。

CreateGraph(*graph*)：构造一个图*graph*。

DestroyGraph(*graph*)：释放图*graph*占用的所有空间。

NumberOfVex(*graph*)：返回图*graph*中顶点的个数。

NumberOfEdge(*graph*)：返回图*graph*中边的条数。

ExistEdge(*graph, u, v*)：判断图*graph*中顶点*u*到*v*之间是否存在边，有返回**true**，无返回**false**。

GetPosition(*graph,v*)：返回顶点*v*在图*graph*中的位置，无则返回NIL。

GetValue(*graph,v*)：返回图*graph*中顶点*v*的值

PutValue(*graph,v,value*)：为图*graph*中顶点*v*赋值*value*。

FirstAdjVex(*graph,v*)：返回图*graph*中顶点*v*的第一个邻接顶点，若*v*无邻接顶点返回NIL。

NextAdjVex(*graph,u,v*)：返回图*graph*中*u*顶点相对*v*顶点的下一个邻接顶点，无则返回NIL。

InsertVex(*graph,v*)：在图*graph*中插入顶点*v*。

InsertEdge(*graph,u,v,weight*)：在图*graph*中顶点*u*和*v*之间插入一条边，权值为*weight*。

RemoveVex(*graph,v*)：在图*graph*中删除顶点*v*及所有邻接于顶点*v*的边。

RmoveEdge(*graph,u,v*)：在图*graph*中删除顶点*u*和*v*之间的边。

DFS(*graph*)：按深度优先遍历图*graph*中顶点。

DFS(*graph,v,visited*)：从顶点v开始深度优先遍历，visited记录顶点访问标记

BFS(*graph*)：按广度优先遍历图*graph*中顶点。

BFS(*graph, v, visited*)：从顶点v开始广度优先遍历，visited记录顶点访问标记

}

提纲

- 7.1 问题引入及求解
- 7.2 图的定义与结构
- 7.3 图的存储实现
- 7.4 图的遍历
- 7.5 图的连通性
- 7.6 图的应用
- 7.7 拓展延伸*
- 7.8 应用场景

图的存储

- 图的存储既要考虑到顶点的存储又要考虑到边的存储。
- 如果按照线性结构和树结构的存储思路，找到一个类似的、既能同时存储顶点又能存储表示顶点间关系的边的结构就非常困难。不妨换个思路，将顶点和边的存储独立开来，顶点归顶点存、边归边存。
- 顶点用一维数组存，边用二维矩阵存 ---邻接矩阵存储法。

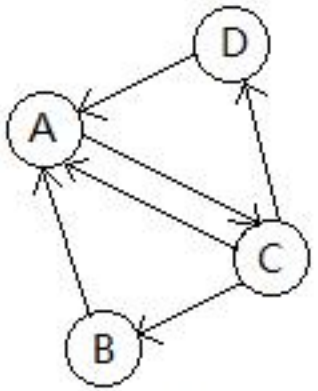
  顶点用一维数组存，边用单链表存 ---邻接表存储法。

邻接矩阵

- 在一维数组中存储顶点信息，在二维矩阵中存储边的信息。
- 如果非加权图中，存在一条自顶点vi到vj 的有向边或无向边，那么在二维矩阵（如A）中，a[i][j] = 1，否则 a[i][j] = 0。

$$A[u][v]=\begin{cases}1 & (u,v),<u,v>\in E\\ 0 & (u,v),<u,v>\notin E\end{cases}$$

- 按照简单图的定义，主对角线上元素a[i][i] = 0，即顶点到自身没有边相连。

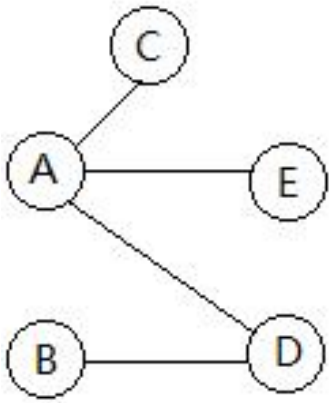


G8

0	1	2	3
A	B	C	D

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

G8的邻接矩阵



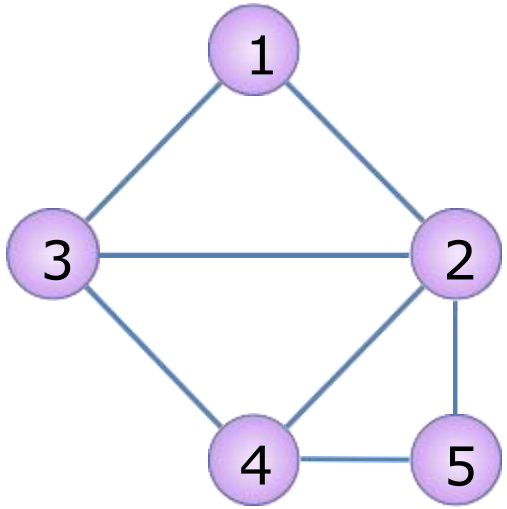
G9

0	1	2	3	4
A	B	C	D	E

	0	1	2	3	4
0	0	0	1	1	1
1	0	0	0	1	0
2	1	0	0	0	0
3	1	1	0	0	0
4	1	0	0	0	0

G9的邻接矩阵

无向图邻接矩阵表示法



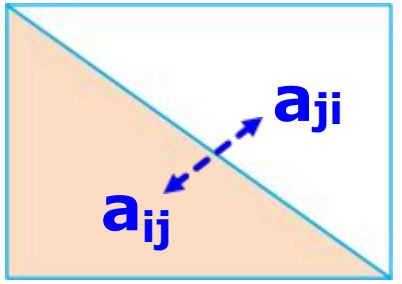
无向图G1

	1	2	3	4	5
1	0	1	1	0	0
2	1	0	1	1	1
3	1	1	0	1	0
4	0	1	1	0	1
5	0	1	0	1	0

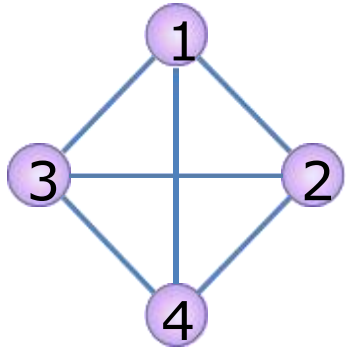
无向图G1的邻接矩阵

- 无向图的邻接矩阵是对称的

$$\begin{pmatrix} 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 \end{pmatrix} \Rightarrow$$

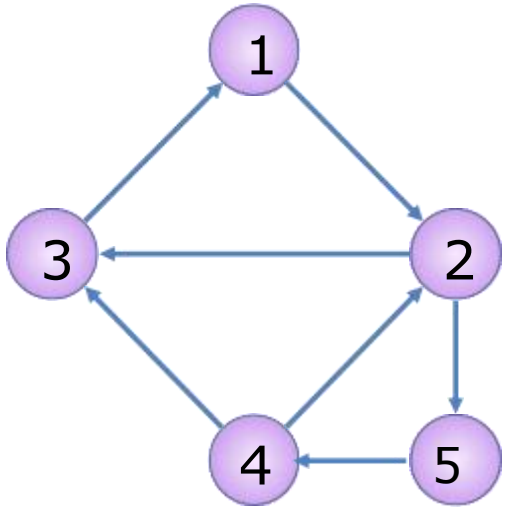


- 顶点 i 的度 = 第 i 行（列）中 1 的个数
- 完全图的邻接矩阵，对角元素为0，其余为1



$$\Rightarrow \begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix}$$

有向图邻接矩阵表示法



有向图G6

	1	2	3	4	5
1	0	1	0	0	0
2	0	0	1	0	1
3	1	0	0	0	0
4	0	1	1	0	0
5	0	0	0	1	0

有向图G6的邻接矩阵

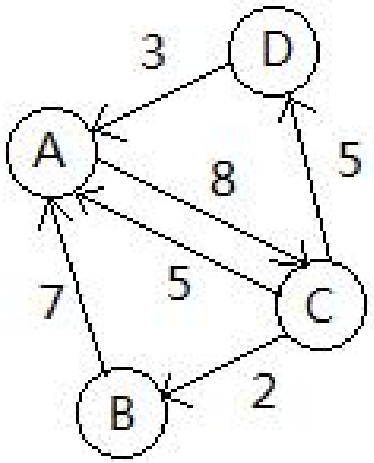
- 在有向图中的邻接矩阵中，
  - ✓ 第 i 行，表示以结点 v_i 为尾的边（出度边）
  - ✓ 第 j 列，表示以结点 v_j 为头的边（入度边）
- 有向图的邻接矩阵可能是不对称的
- 顶点的出度 = 第 i 行元素个数之和
- 顶点的入度 = 第 j 列元素个数之和
- 顶点的度数 = 第 i 行 和 第 i列 元素个数之和

加权邻接矩阵

- 当图中边带有权值时，可以用加权邻接矩阵表示加权有向图或无向图。
- 如果加权图中，存在一条自顶点vi到vj 的有向边或无向边，那么在二维矩阵（如A）中，a[i][j] = 权值，否则 a[i][j] = ∞。

$$A[u][v]=\begin{cases} w & ,\ (u,v),<u,v>\in E \\ 0 & ,\ u=v \\ \infty & ,\ (u,v),<u,v>\notin E \end{cases}$$

- 按照简单图的定义，主对角线上元素a[i][i] = 0或者∞，即顶点到自身没有边相连。



G10

0	1	2	3
A	B	C	D

	0	1	2	3
0	0	∞	8	∞
1	7	0	∞	∞
2	5	2	0	5
3	3	∞	∞	0

G10的邻接矩阵

邻接矩阵和加权邻接矩阵的优缺点

- 优点
  - ✓ 简单、直观、好理解
  - ✓ 方便检查任意一对结点间是否存在边（A[i][j] = 0|1），O(1)的时间复杂度
  - ✓ 方便查找任意顶点的所有“邻接点”（有边直接相连）
  - ✓ 方便计算任一顶点的度（所有出发的边数为出度，所有指向该结点的边数为入度）
- 缺点
  - ✓ 不便于增加和删除顶点
  - ✓ 浪费时间：不便于统计边的数量，需要遍历矩阵所有元素，时间复杂度高 $O(n^2)$
  - ✓ 浪费空间：存储稀疏图存在大量无效元素（当边的总数远远小于n^2，也需n^2个内存单元来存储边的信息，）

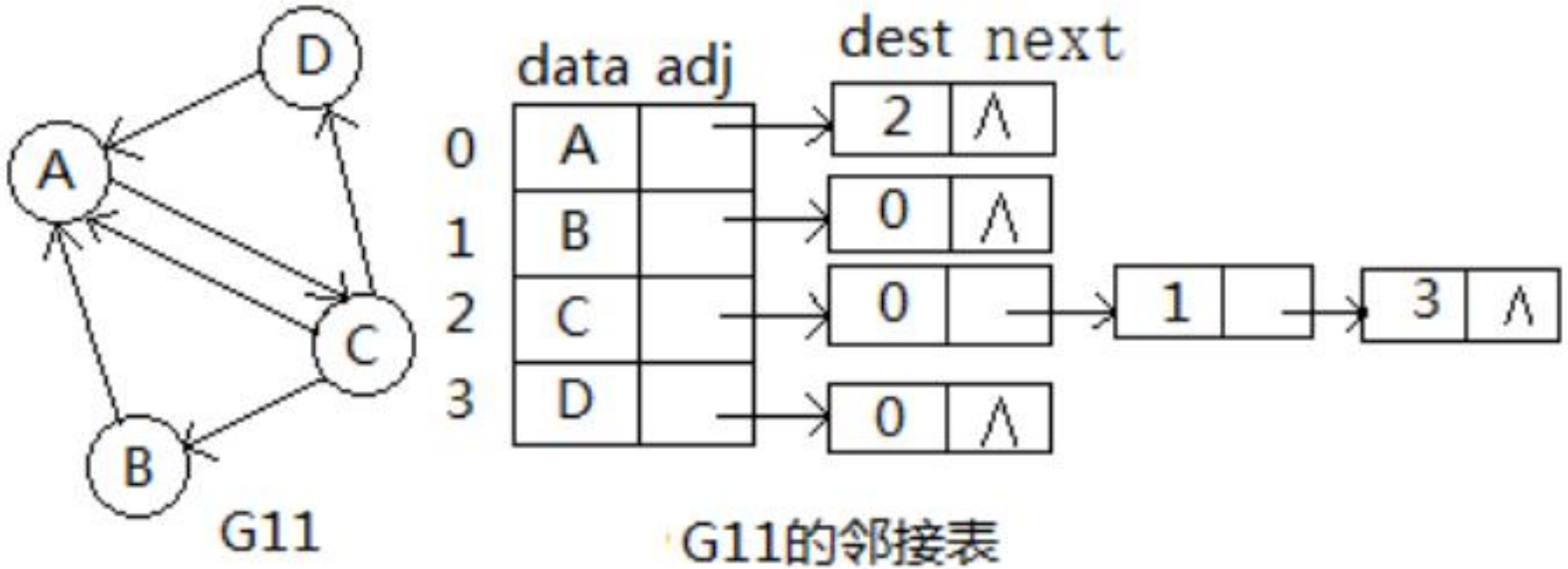
邻接矩阵的适应情况和特殊图的存储处理

- 如果图是稠密图（边数非常多）：
  - ✓ 对有向图，采用邻接矩阵是合适的。
  - ✓ 对无向图，因关于主对角线对称，可只存储其上三角矩阵或下三角矩阵。
- 如果图是稀疏图（边数很少),且非零元素的分布没有规律:
  - ✓ 通常的做法是只存储其中的非零元素和非零元素所在的位置，每个非零元素a[i][j]用一个三元组来表示：（i,j, a[i][j]）.
  - ✓ 将此三元组按照一定的次序排列，如先按照行序再按照列序排列。
  - ✓ 三元组可以放在顺序表或者链表中。

邻接表

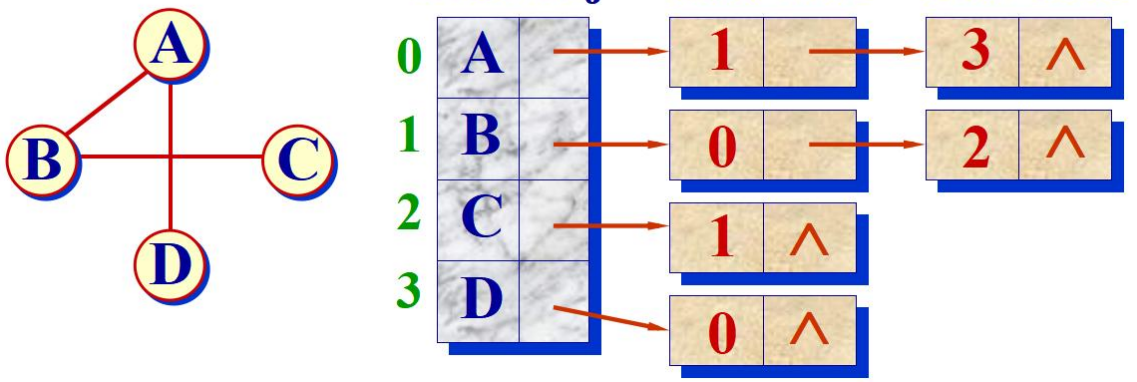
- 顶点依然用一个一维数组来存储，而边的存储是将由同一个顶点出发的所有边组成一条单链表。
- 存储顶点的一维数组称**顶点表**，存储边信息的单链表称**边表**。一个图由顶点表和边表共同表示。
- 顶点表不仅保存各个顶点的信息，还保存由该顶点射出的边形成的单链表中首结点的地址（首指针），这种方法称**邻接表**表示法。

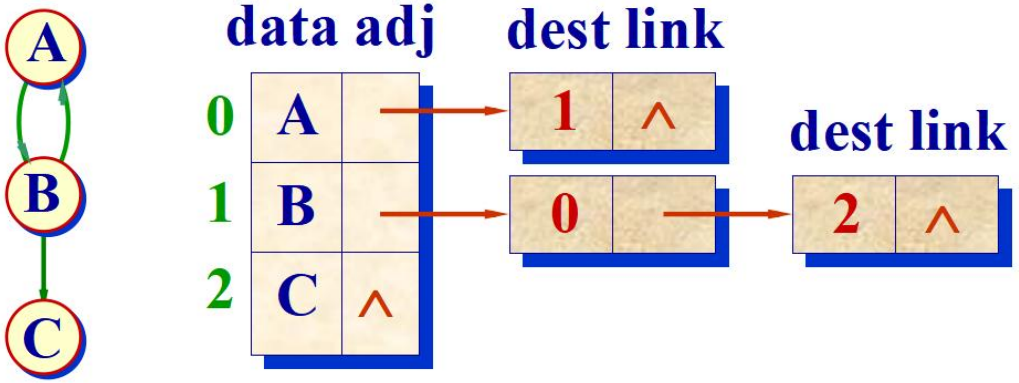
邻接表



邻接表存储特点

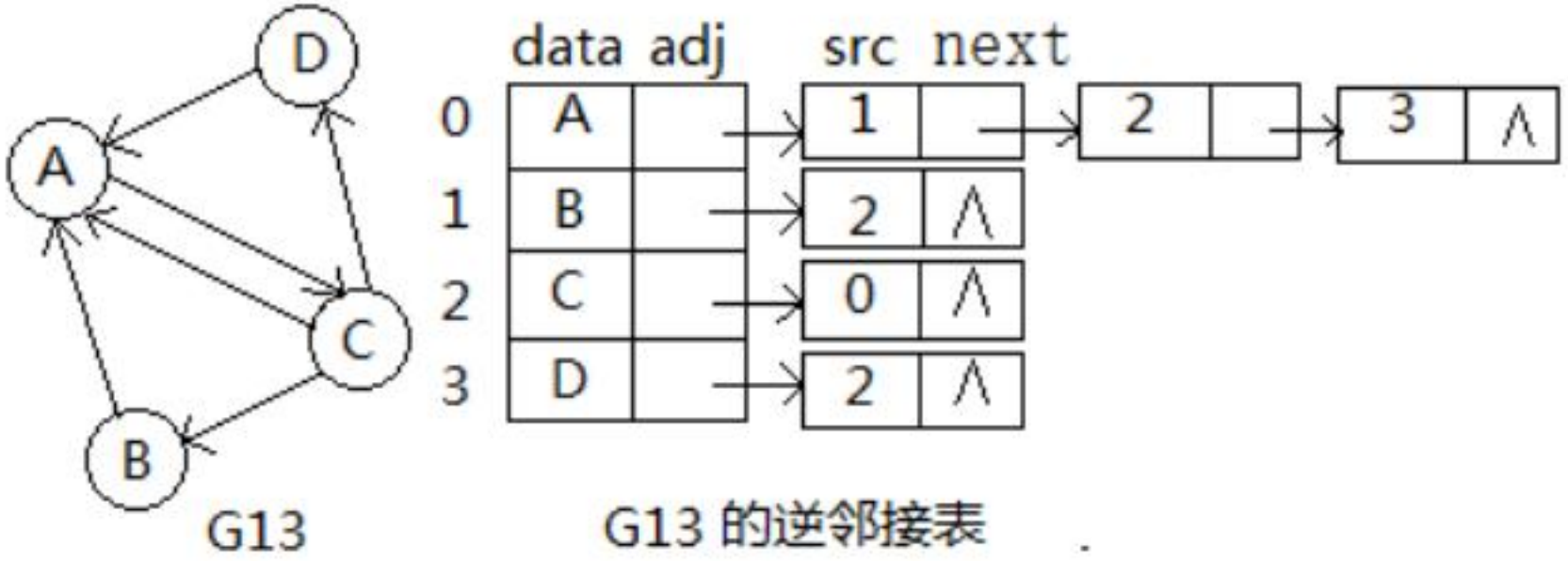
- 仅存储有边的信息，不存储无边信息，在图比较稀疏的情况下，空间的利用率大大提高
- 无向图，同一条边存储了两次
- 计算某个顶点v的出度（有向图）或者度（无向图），只需遍历该顶点v指向的边表，即利于计算出度
- 计算某个顶点v的入度（有向图），需要遍历所有顶点v指向的边表，即不利于计算入度





邻接表存储

逆邻接表：有向图的逆邻接表中，顶点表保存该顶点的射入边形成的单链表的首结点地址，有利于计算顶点的入度。



G13　　G13 的逆邻接表

邻接表的优缺点

- 优点
  - ✓ 方便查找任一顶点的所有“邻接点”：遍历某一结点的边链表
  - ✓ 便于统计边的数量：按照头结点表 -> 边链表顺序扫描所有结点，有向图时间复杂度O(V+e)，无向图的时间复杂度为O(V+2e)
  - ✓ 空间效率高：非常适合于稀疏图，有向图的空间复杂度都是S(V+e)，无向图的空间复杂度为S(V+2e)。
  - ✓ 便于统计无向图各顶点的度

- 缺点
  - ✓ 不便于统计有向图各个顶点的度：只能计算出度；计算入度需要构造“逆邻接表”（保存指向自己的边）
  - ✓ 不方便判断顶点间是否存在边：需要遍历多个边链表，最坏情况O(n)

另外一种邻接表存储

邻接表中，顶点表用了一维数组，图初始化时需要预估数组规模。

一种改进：

顶点表也用一个单链表表示。此时，dest用顶点结点地址而不用下标。

邻接多重表*

邻接表中，无向图时每条边都用了两个边结点，即同一条边被存储了两次。

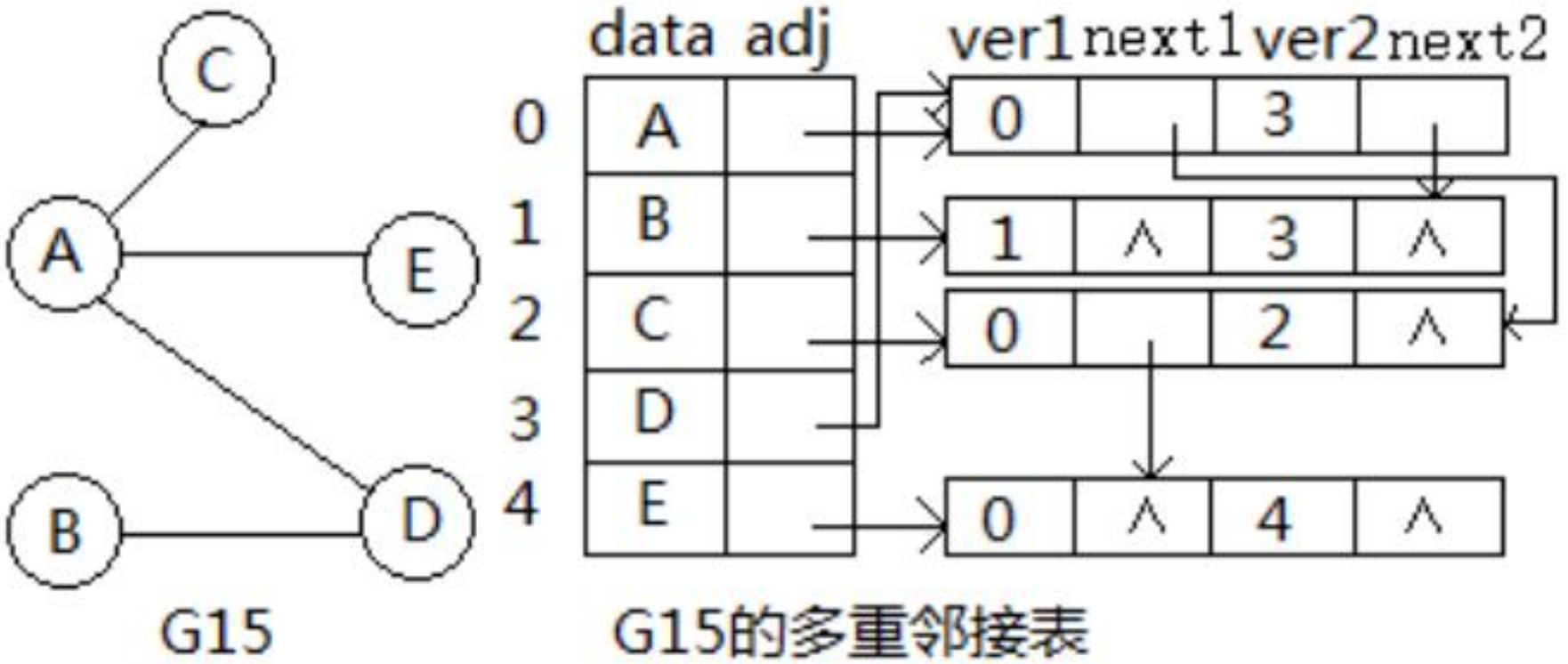
1）空间浪费，

2）在某些应用中，如遍历所有边时因重复而不方便

邻接多重表：

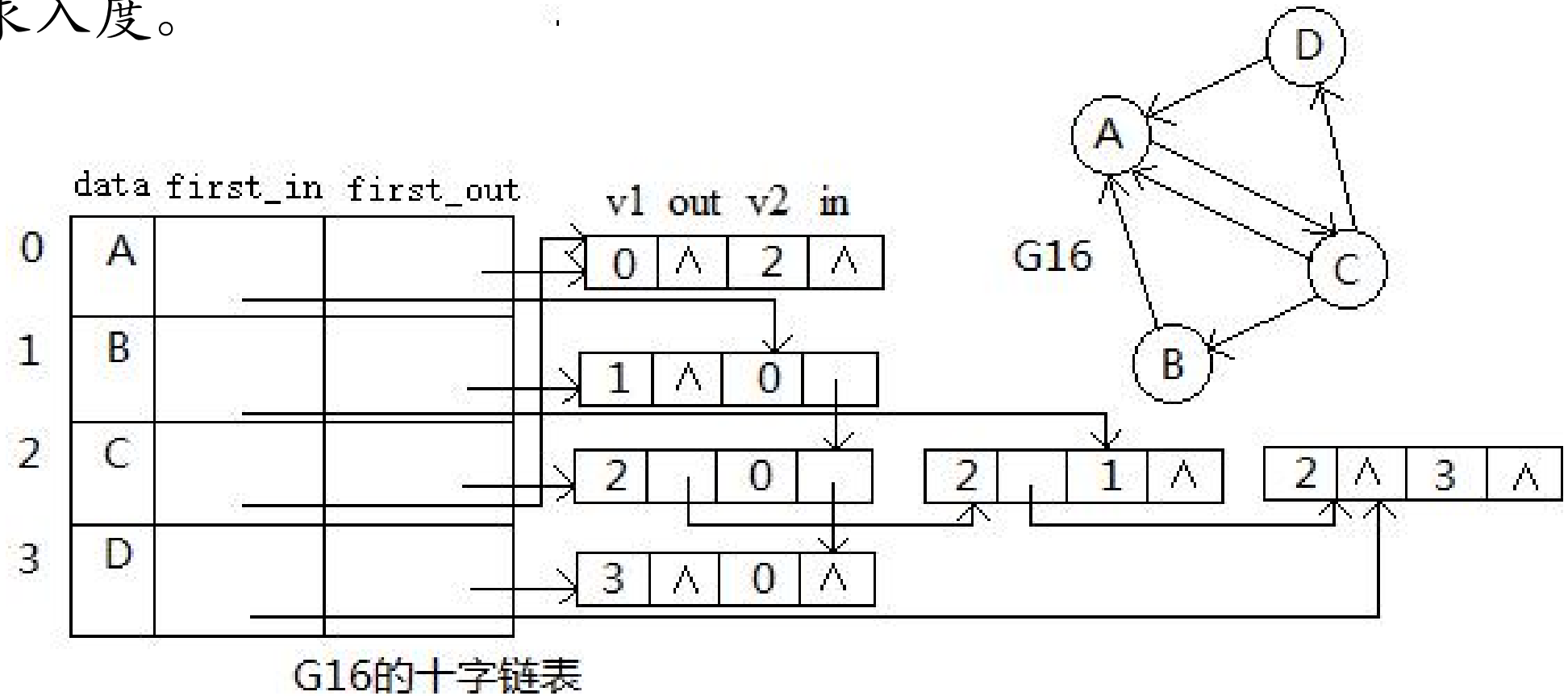
1. 每条边仅使用一个结点来表示，即只存储一次，但这个边结点同时要在它邻接的两个顶点的边表中被链接。
2. 为了方便两个边表同时链接，每个边结点不再像邻接表中那样只存储边的一个顶点，而是存储两个顶点。

邻接多重表*


C
A
E
B
D
G15
data adj
ver1 next1 ver2 next2
0
1
2
3
4
A
B
C
D
E
0
3
1
^
3
^
0
2
^
0
^
4
^
G15的多重邻接表

十字链表«

十字链表将有向图的邻接表和逆邻接表结合在了一起，既利于求出度又利于求入度。



G16的十字链表

图的基本操作实现

图的操作所需要花费的时间通常既和顶点的个数n有关，又和边的条数m有关，因此时间复杂度会包含n和m两个变量。

➢ **邻接矩阵表示的图**

假设已知条件为：

- ✓ 图中实际顶点个数 *n_verts*、图中实际边的条数*m_edges*、
- ✓ 图中顶点可能的最大数量*kMaxVertex*、保存顶点数据的一维数组*ver_list*、
- ✓ 保存邻接矩阵内容的二维数组*edge_matrix*、
- ✓ 无边时权重的赋值*no_edge_value*（一般图为0，网为无穷大kMaxNum）、
- ✓ 有向或无向图标志*directed*(有向图为真，无向图为假)。

图用邻接矩阵表示时部分基本操作算法描述

算法7-1: 获取图的顶点个数 NumberOfVex(*graph*)

输入： 图*graph*

输出： 图的顶点个数

1.**return** *graph.n_verts*

时间复杂度 O（1）

算法7-2: 判断边是否存在 ExistEdge(graph, u, v)

输入： 图graph、两个顶点u和v

输出： u到v有边返回 true，否则返回 false

```
1.  if u <graph.n_verts且 v <graph.n_verts then
2.  |  if  u≠v 且 graph.edge_matrix[u][v]≠graph.no_edge_value then
3.  |  |  return true
4.  |  end
5.  end
6.  return false
```

时间复杂度 O（1）

图用邻接矩阵表示时部分基本操作算法描述

算法7-4: 向图中插入边 InsertEdge(*graph, u,v,weight*)

输入：图*graph*，边的两个端点*u*和*v*，边的权重*weight*

输出：插入了边(*u,v*)或<*u,v*>的图

```
1.  if u≠ v 且 ExistEdge(graph, u, v) = false then
2.  |  graph.edge_matrix[u][v]←weight
3.  |  graph.m_edges←m_edges+1
4.  |  if graph.directed = false then  //如果是无向图，对主对角线对称的元素赋值
5.  |  |  graph.edge_matrix[v][u] ←weight
6.  |  end
7.  end
```

时间复杂度 O（1）

图用邻接矩阵表示时部分基本操作算法描述

算法7-6: 从图中删除顶点及所有邻接于该顶点的边 RemoveVex(*graph*,*v*)

输入： 图*graph*、顶点*v*

输出： 删除了顶点*v*及所有邻接于顶点*v*的边的图*graph*

1. if *v*<0 或*v* ≥ graph.n_verts then
2. | 待删除的顶点不存在，退出
3. end
4. *graph.ver_list*[v]← *graph.ver_list[graph.n_verts-1]* //用最后一个顶点信息覆盖*v*
5. *count* ← 0 //*count*计数由顶点*v*射出的边的条数

1-5： 时间复杂度 O（1）

图用邻接矩阵表示时部分基本操作算法描述

```
6.  for u= 0 to graph.n_verts-1 do
7.  |    if ExistEdge(graph, v, u) = true then
8.  |  |  count← count+1
9.  |  end
10. end
11. if graph.directed=true then //有向图还要计数射入顶点v的边的条数
12. |  for u= 0 to graph.n_verts-1 do
13. |  |  if ExistEdge(graph, u, v) = true then
14. |  |  |  count← count+1
15. |  |  end
16. |  end
17. end
```

6-10： 时间复杂度 O（n）

11-17： 时间复杂度 O（n）

图用邻接矩阵表示时部分基本操作算法描述

```
18. for u= 0 to graph.n_verts-1 do //将矩阵最后一行移入第v行
19. |     graph.edge_matrix[v][u]← graph.edge_matrix[graph.n_verts-1][u]
20. end
21. for u= 0 to graph.n_verts-1 do //将矩阵最后一列移入第v列
22. |     graph.edge_matrix[u][v] ← graph.edge_matrix[u][graph.n_verts-1]
23. end
24. graph.m_edges← graph.m_edges-count //更新边的条数
25. graph.n_verts← graph.n_verts-1//更新顶点个数
```

18-20： 时间复杂度 O（n）

21-23： 时间复杂度 O（n）

24-25： 时间复杂度 O（1）

加法原理，总时间复杂度 O（n）

图用邻接表表示时部分基本操作算法描述

算法7-7: 返回图中顶点的第一个邻接顶点 FirstAdjVex(*graph*,*v*)

输入： 图*graph*、顶点*v*

输出： 图*graph*中顶点*v*的第一个邻接顶点，若*v*无邻接顶点返回NIL。

```
1.  if v<graph.n_verts then
2.  |  return graph.ver_list[v].adj
3.  end
```

时间复杂度 O（1）

图用邻接表表示时部分基本操作算法描述

算法7-8: 判断边是否存在 ExistEdge(*graph, u, v*)

输入：图graph、两个顶点u和v

输出：u到v有边返回 true，否则返回 false

```
1.  p ← FirstAdjVex(graph,u)
2.  while p≠NIL 且 p.dest≠v do
3.  |  p ← p.next
4.  end
5.  if p≠NIL then
6.  |  return true
7.  else
8.  |  return false
9.  end
```

时间复杂度 O（m）

图用邻接表表示时部分基本操作算法描述

算法7-9: 向图中插入边 InsertEdge(*graph, u,v,weight*)

输入： 图*graph*，边的两个端点*u*和*v*，边的权重*weight*

输出： 插入了边(*u,v*)或<*u,v*>的图

```
1.  if ExistEdge(graph, u, v)=false then
2.  |  p← new EdgeNode
3.  |  p.dest←v
4.  |  p.weight← weight
5.  |  p.next←graph.ver_list[u].adj
6.  |  graph.ver_list[u].adj←p
7.  |  graph.m_edges←graph.m_edges+1
```

图用邻接表表示时部分基本操作算法描述

```
8.   |  if graph.directed=false then  //如果是无向图，还要将u插入v的边表中
9.   |  |  p ← new EdgeNode
10.  |  |  p.dest ←u
11.  |  |  p.weight ← weight
12.  |  |  p.next ← graph.ver_list[v].adj
13.  |  |  graph.ver_list[v].adj ←p
14.  |  end
15. end
```

时间复杂度 O（1）

图用邻接表表示时部分基本操作算法描述

算法7-10: 从图中删除顶点及所有邻接于该顶点的边 RemoveVex(*graph*,*v*)

输入： 图*graph*、顶点*v*

输出： 删除了顶点*v*及所有邻接于顶点*v*的边的图*graph*

```
1.  if v<0 或v≥ graph.n_verts-1  then
2.  |  待删除的顶点不存在，退出
3.  end
4.  count←0 //count计数与顶点v邻接的边的条数
5.  p ←graph.ver_list[v].adj //删除由顶点v射出的边
6.  while p≠ NIL  do
7.  |  next_p ← p.next
8.  |  delete p
9.  |  count ←count+1
10. |  p ← next_p
11. end
```

1-11：时间复杂度 O（m）

图用邻接表表示时部分基本操作算法描述

算法7-10: 从图中删除顶点及所有邻接于该顶点的边 RemoveVex(*graph,v*)

```
12. for u←0 to graph.n_verts-1 do //删除射入顶点v的边
13. |     p ←graph.ver_list[u].adj
14. |     if p≠NIL then    //非空链表
15. |  |     if p.dest=v then //首结点为射入顶点v的边
16. |  |  |     graph.ver_list[u].adj ←p.next
17. |  |  |     delete p
18. |  |  |     count  count+1
19. |  |     else //非首结点
20. |  |  |  while p.next≠NIL  且 p.next.dest≠v do //找到射入顶点v的边
21. |  |  |  |  p ← p.next
22. |  |  |  end
```

图用邻接表表示时部分基本操作算法描述

```
23. |   |   |   if p.next≠NIL then //找到<u,v>这条边，删除
24. |   |   |   |   next_p ← p.next
25. |   |   |   |   p.next ← next_p.next
26. |   |   |   |   delete next_p
27. |   |   |   |   count  count+1
28. |   |   |   end
29. |   |   end
30. |   end
31. end
32. last_v ← graph.n_verts - 1 //最后一个顶点的编号
```

12-32：内外循环相关，换个角度分析，算法针对每个顶点，检测了其邻接的每一条边。故时间复杂度 O（n+m）

图用邻接表表示时部分基本操作算法描述

```
33. for u←0 to last_v-1 do //将原来射入最后一个顶点的边都更新编号为v
34. |   p ←graph.ver_list[u].adj
35. |   if p≠NIL then    //非空链表
36. |   |   while p ≠NIL  且 p.dest≠last_v do //找到射入顶点v的边
37. |   |   |   p ← p.next
38. |   |   end
39. |   |   if p ≠ NIL then //将原来射入最后一个顶点的边都更新编号为v
40. |   |   |   p.dest ← v
41. |   |   end
42. |   end
43. end
```

33-43：和12-32同理，时间复杂度 O（n+m）

图用邻接表表示时部分基本操作算法描述

```
44. graph.ver_list[v] ← graph.ver_list[last_v] //顶点表中最后一个顶点移到位置v
45. if graph.directed=false then //无向图实际删除的边数要减半
46. |   count ←count/2
47. end
48. graph.m_edges ←graph.m_edges  – count //更新边数
49. graph.n_verts ←graph.n_verts-1  //更新顶点个数
```

44-49：时间复杂度 O（1）

加法原理：总时间复杂度 O（n+m）

提纲

- 7.1 问题引入及求解
- 7.2 图的定义与结构
- 7.3 图的存储实现
- 7.4 图的遍历
- 7.5 图的连通性
- 7.6 图的应用
- 7.7 拓展延伸*
- 7.8 应用场景

图的遍历

按照某种方式逐个访问图中的所有顶点，且每个顶点只被访问一次。

1. 最简单的方式是沿着顶点表循环访问一遍，由此达到了遍历的目标。

   这种方式，完全没有借用边的信息。

2. 两种借助边信息实现遍历的算法：深度优先遍历和广度优先遍历。

   基于这两种遍历可以解决图中更多的涉及到边的问题，如图的连通性问题

遍历图和遍历二叉树的不同

- 图中的顶点地位相同，没有特殊的顶点；二叉树结构中有一个特殊的根结点。
- 图中一个顶点可以和多个其它顶点邻接，可看作有多个直接前驱结点和多个后继结点，并可能存在回路。二叉树中每个结点的直接前驱结点只有一个，直接后继结点最多有两个，且不存在回路。
- 无向图中，邻接于一条边的两个顶点，甚至可以视作互为后继。

为避免通过不同前驱多次到达同一顶点，造成重复访问已经访问过的顶点，在图的遍历过程中，通常对已经访问过的顶点加特殊标记（即已访问标志）。

深度优先遍历

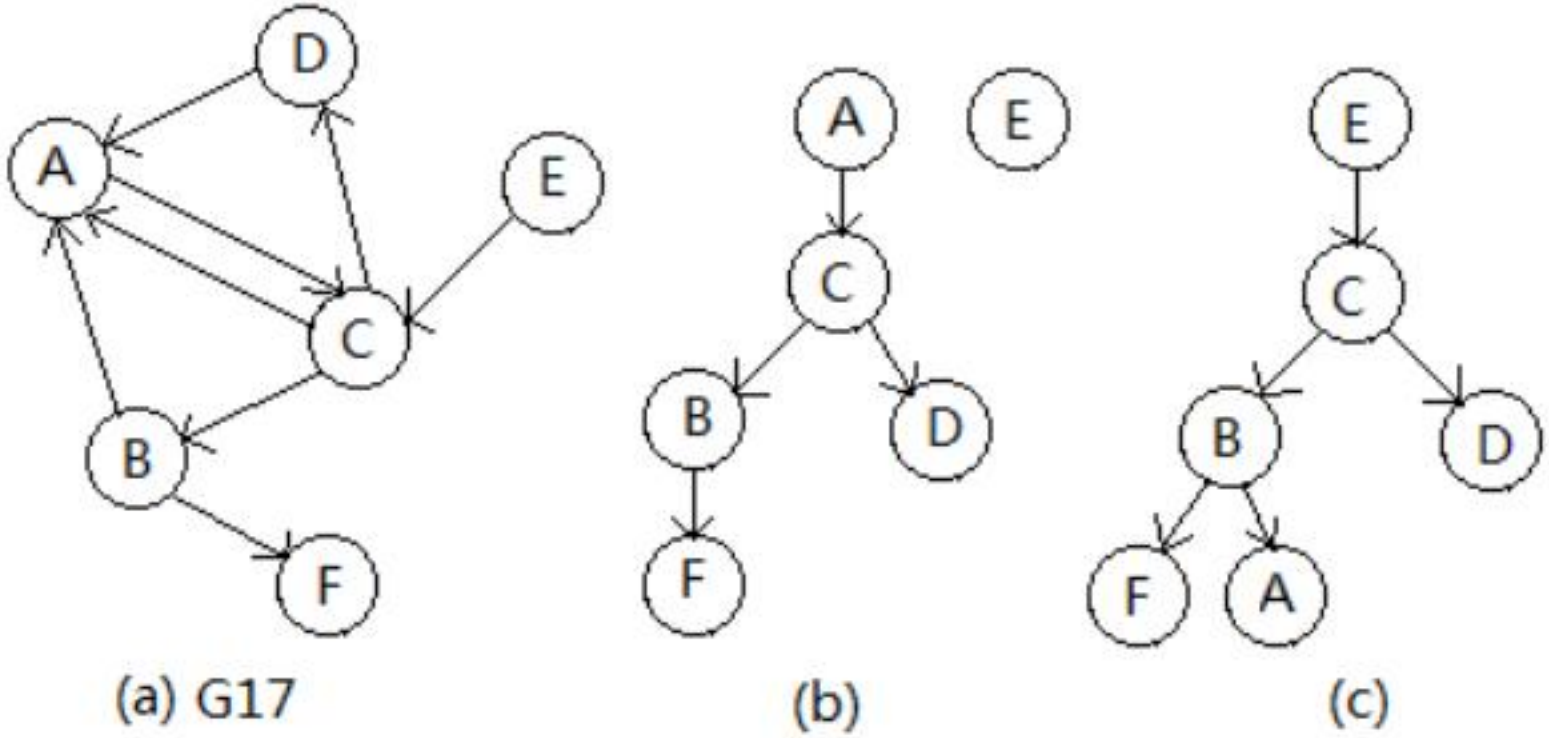
DFS（Depth First Search）

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. 依次从该顶点的未被访问过的第1个、第2个、第3个…… 邻接顶点出发，依次进行深度优先遍历，即转向1。
3. 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。

思考：什么条件下会从3再转向1？

深度优先遍历示例



- 深度优先遍历结果是不唯一的。
- 它是一个典型的递归过程：随着未访问顶点的逐步变少，它是用了一个对规模小的图的遍历问题去解决对规模大的图的遍历问题。

深度优先遍历算法

算法7-11: 按深度优先遍历图中结点　DFS(*graph*)

输入： 图*graph*

输出： 图*graph*的深度优先遍历序列

```
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. |   |   DFS(graph, v, visited)
7. |   end
8. end
```

1-3：时间复杂度 O（n）

4-8：for循环O(n), 循环体依赖于第6行DFS时间复杂度

深度优先遍历

算法7-12: 从指定顶点开始深度优先遍历 DFS(*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*出发的深度优先访问序列

```
1.  visited[v]←true
2.  visit(graph, v)
3.  p ← graph.ver_list[v].adj
4.  while p≠ NIL do
5.  | if visited[p.dest]=false then
6.  | | DFS(graph, p.dest, visited)
7.  | end
8.  | p ←p.next
9.  end
```

1-9：每次DFS调用访问1个顶点和若干条边。合计访问到每个顶点和每条边一次，时间复杂度 O（n+m）。

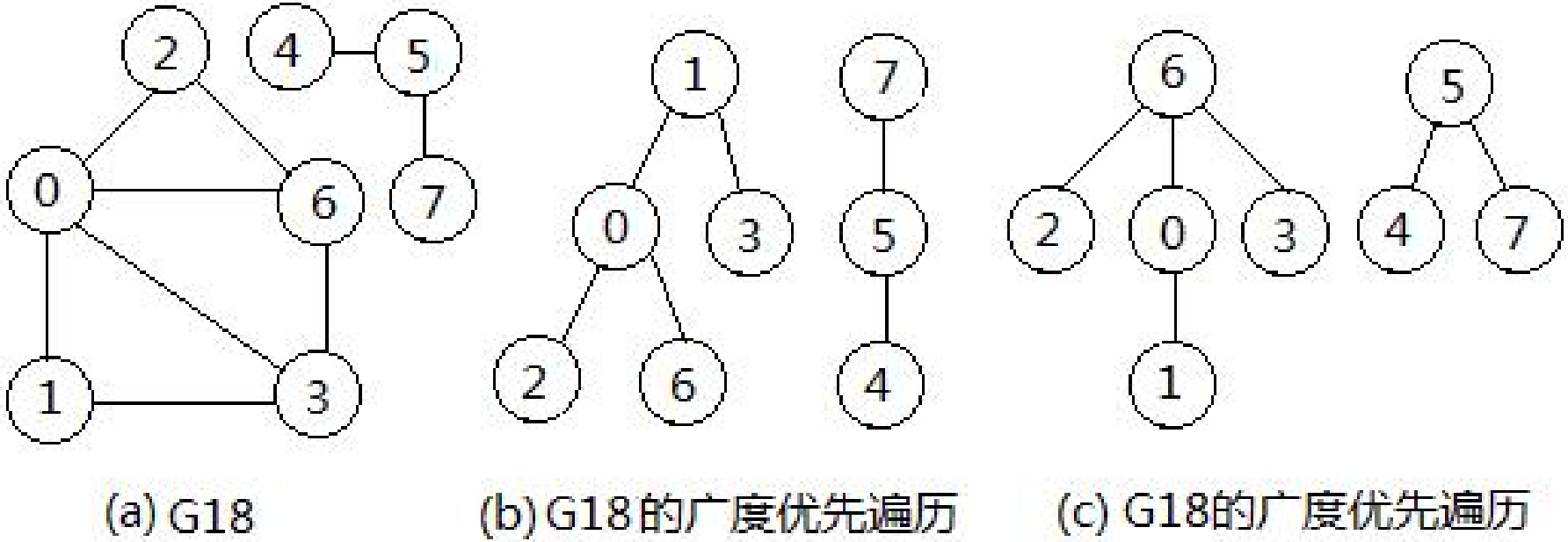
广度优先遍历

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 *v1* 、*v2* 、*v3*…… *vk*进行访问且加已访问标志。
3. 依次对顶点 *v1* 、*v2*、*v3*…… *vk*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



(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。

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. |   |   BFS(graph, v, visited)
7. |   end
8. end
```

1-3：时间复杂度 O（n）

4-8：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

```
1.  InitQueue(queue)
2.  EnQueue(queue, v)
3.  while IsEmpty(queue)=false do
4.  | u← DeQueue(queue)
5.  | if visited[u]=false then
6.  | | visited[u] ←true
7.  | | visit(graph, u)
8.  | | p ← graph.ver_list[u].adj
```

广度优先遍历算法

```
9.  | | while p≠NIL do
10. | | | if visited[p.dest]=false then
11. | | | | EnQueue(queue, p.dest)
12. | | | end
13. | | | p ←p.next
14. | | end
15. | end
16. end
```

1-16：每次BFS调用访问1个顶点和若干条边。合计访问到每个顶点和每条边一次，时间复杂度 O（n+m）。

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。

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。

提纲

- 7.1 问题引入及求解
- 7.2 图的定义与结构
- 7.3 图的存储实现
- 7.4 图的遍历
- 7.5 图的连通性
- 7.6 图的应用
- 7.7 拓展延伸*
- 7.8 应用场景

无向图的连通性

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
5.  for v ← 0 to graph.n_verts-1 do
6.  |  if visited[v]=false then
7.  |  |  count ← count +1
8.  |  |  BFS(graph, v, visited) |
9.  |  end
10. end
```

无向图的连通性

```
11. if count=1 then
12. |  ret ← true
13. else
14. |  print  count
15. |  ret ← false
16. end
17. 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。

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

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

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*)
6. InitQueue(*level_queue*)
7. EnQueue(*ver_queue, v*)
8. EnQueue(*level_queue,* 0)

六度空间理论的验证算法

```
9.  while IsEmpty(ver_queue)=false do
10. | cur_ver ←DeQueue(ver_queue)
11. | cur_level ←DeQueue(level_queue)
12. | if cur_level≤6 then
13. | | if visited[cur_ver]=false then //未访问过该结点则对它加访问标志
14. | | | visited[cur_ver] ←true
15. | | | count ←count +1
16. | | | p ←graph.ver_list[cur_ver].adj //向cur_ver的下一层搜索
17. | | | while p≠ NIL do
18. | | | | if visited[p.dest]=false then
19. | | | | | EnQueue(ver_queue, p.dest)
20. | | | | | EnQueue(level_queue, cur_level+1)
21. | | | | end
22. | | | | p ← p.next
23. | | | end
```

六度空间理论的验证算法

```
24. |  |  end
25. |  else  //已完成6层搜索，算法结束
26. |  |  break
27. |  end
28. end
29. return count/graph.n_vers
```

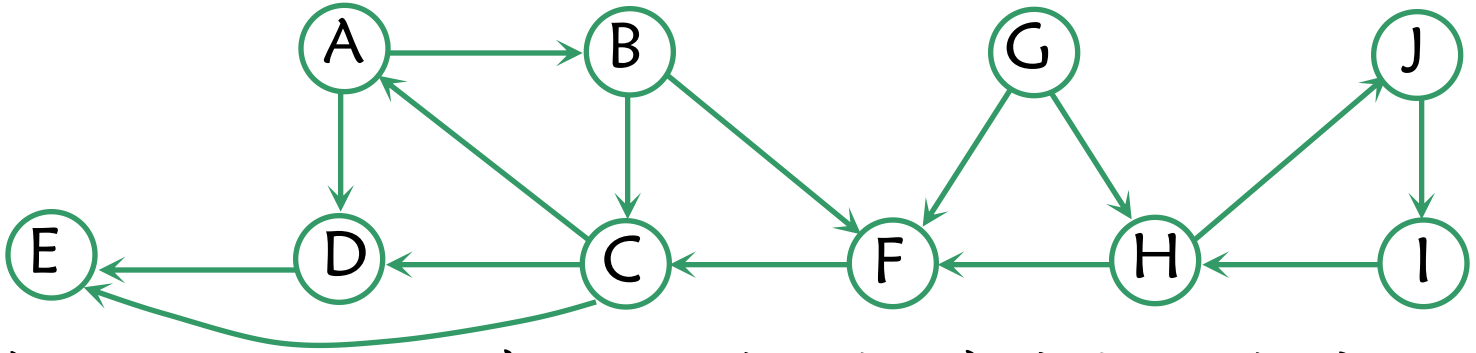
无向连通图的BFS 算法，时间复杂度 O（n+m）

有向图的连通性

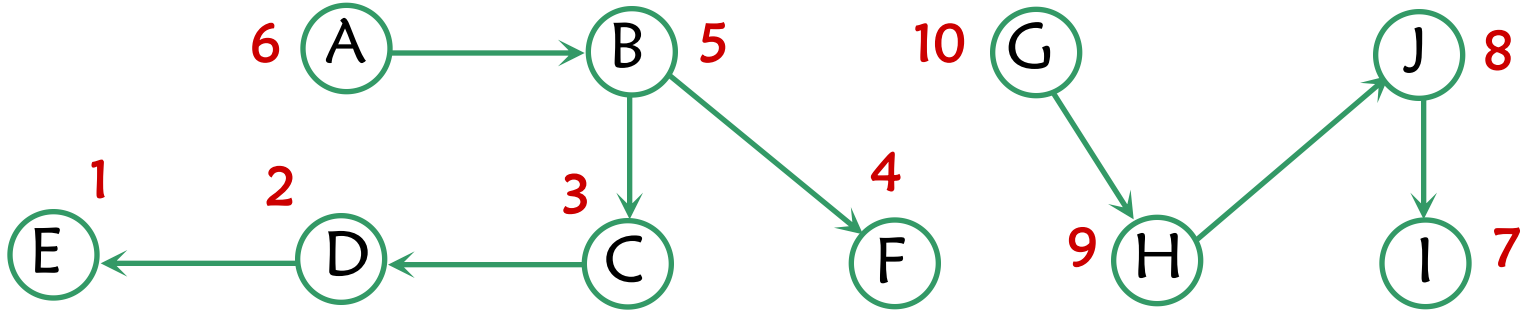
- 有向图的强连通分量问题解决起来比较复杂。
- 对一个强连通分量来说，要求每一对顶点间相互可达。
- 以上的深度、广度优先遍历都只是计算了单向路径。

寻找强连通分量的Kosaraju算法

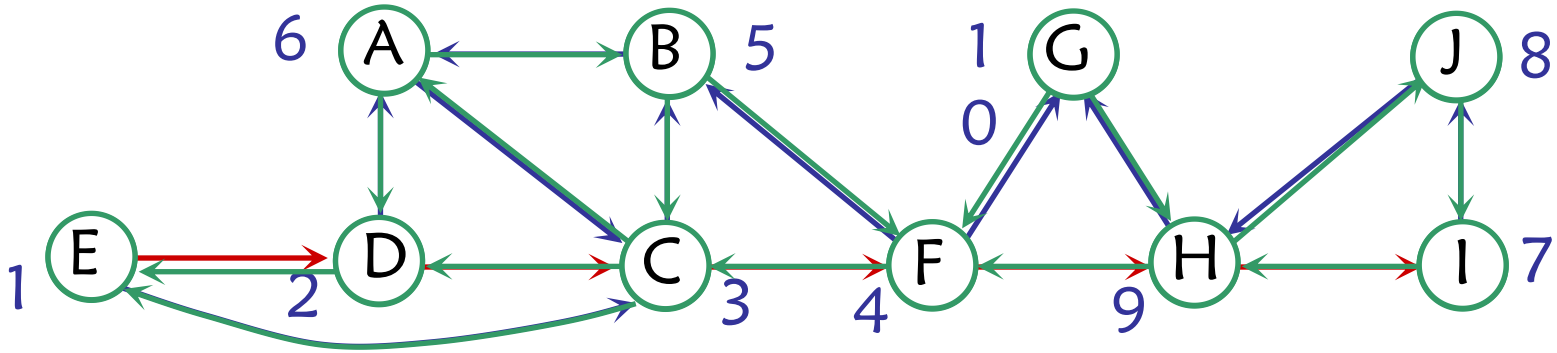
- 以下图为例，说明算法的思路。



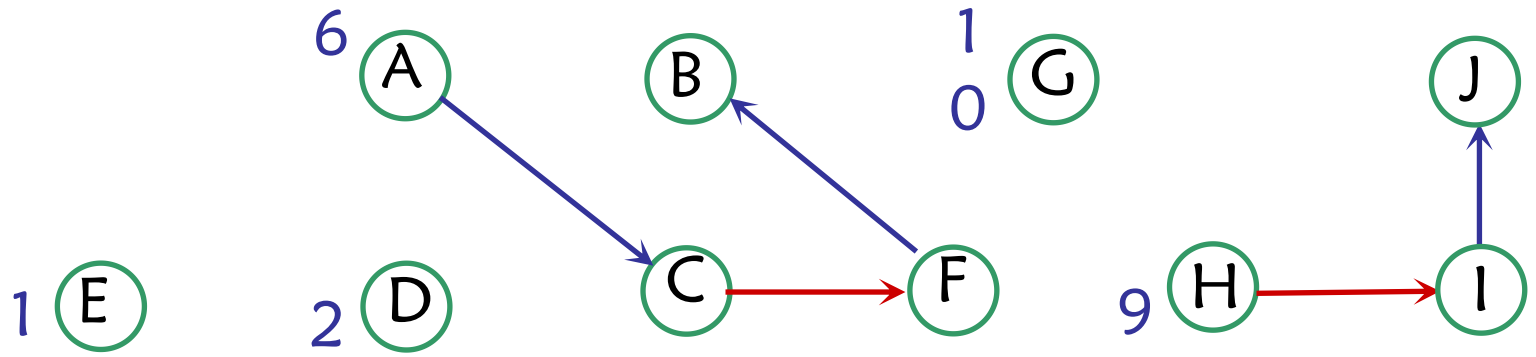
1. 首先对图进行一次DFS，在回退时记忆对结点回溯的顺序：
   EDCFBAIJHG



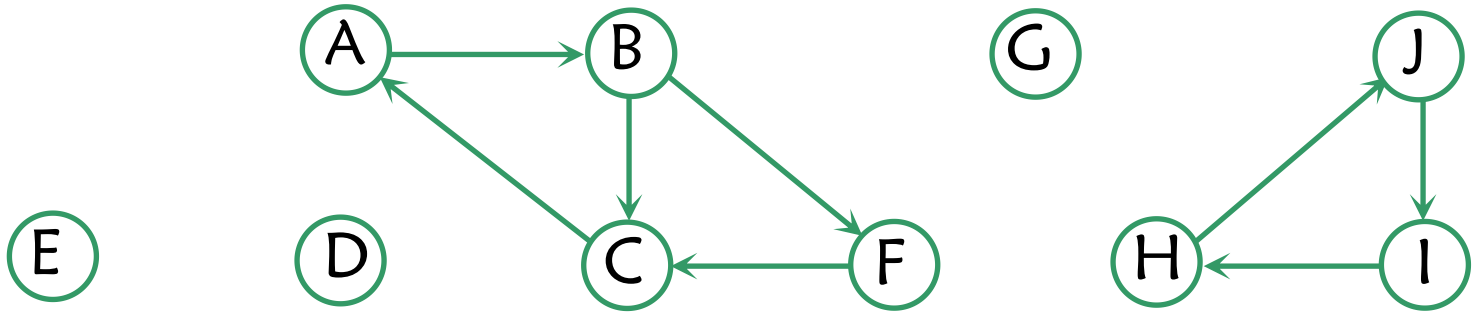
2. 把图的所有的有向边逆转：



3. 再对得到的图沿回溯顺序 EDCFBAIJHG 从编号最高的 G 开始，再进行一次 DFS，所得到的深度优先森林（树）即为强连通分量的划分。



- 对应到原图，可得到 5 个强连通分量：

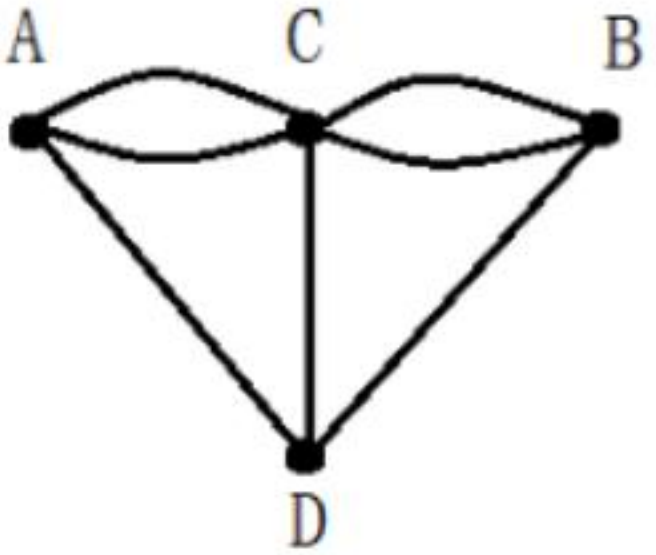


提纲

- 7.1 问题引入及求解
- 7.2 图的定义与结构
- 7.3 图的存储实现
- 7.4 图的遍历
- 7.5 图的连通性
- 7.6 图的应用
- 7.7 拓展延伸*
- 7.8 应用场景

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。

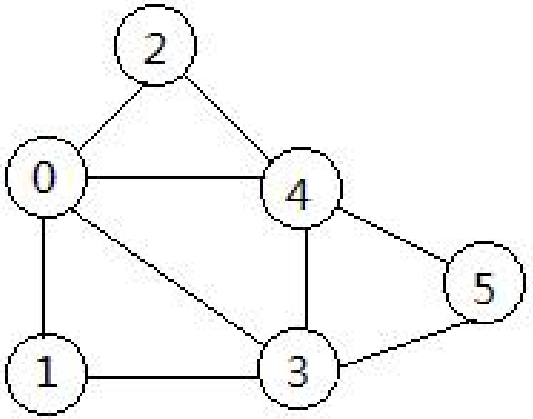
欧拉定理

- 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）。

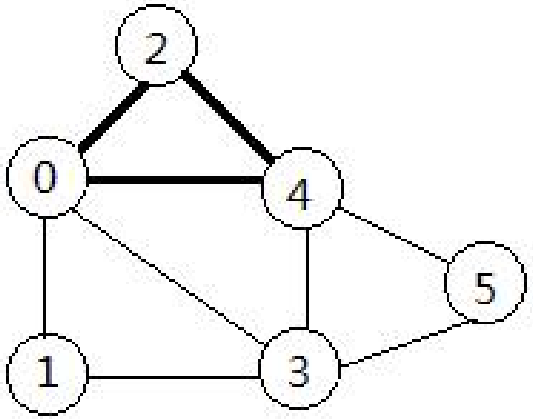
欧拉回路求解方法

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，最后直到回到顶点v且v没有尚未被访问的边，此时便得到了一个回路，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



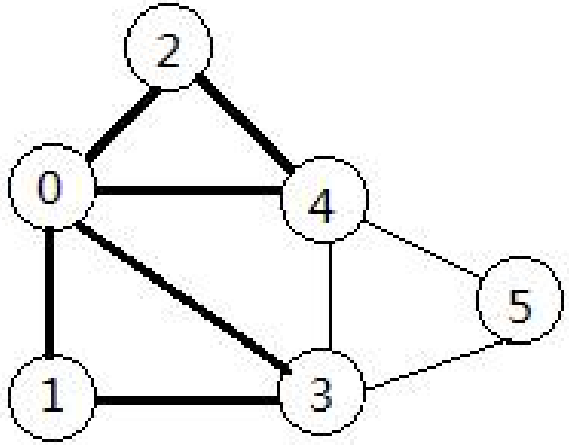
(a)G20



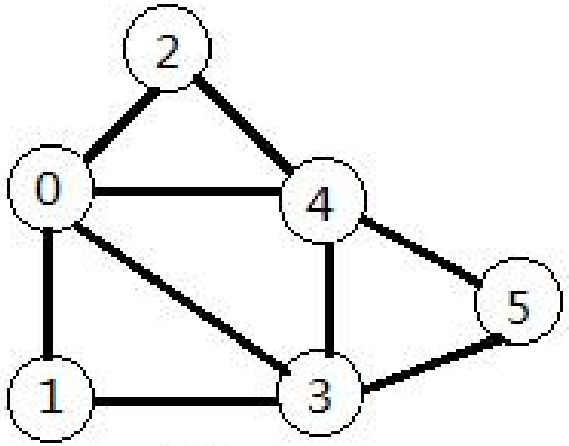
(b)

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

算法7-17: 从给定点出发获得一条回路 GetCircuit(*graph, start*)

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*
4. *circuit.last* ←*new_node*
5. *p* ←*graph.ver_list*[*start*].*adj*
6. *head* ←*start*

欧拉回路求解算法

```
7.  while p≠ NIL 且 p.dest ≠ start do
8.  |  tail ← p.dest
9.  |  RmoveEdge(graph,head, tail)
10. |  RmoveEdge(graph,tail, head)
11. |  new_node ←new EulerNode(tail, NIL)
12. |  circuit.last.next ← new_node
13. |  circuit.last ← circuit.last.next
14. |  p ←graph.ver_list[tail].adj
15. |  head ← tail
16. end
17. return circuit
```

时间复杂度 O（n+m）

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

```
1.  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
2.  |  p ←graph.ver_list[v].adj
3.  | degree[v] ← 0
4.  | while p≠ NIL do
5.  | | degree[v] ←degree[v]+1
6.  | | p ←p.next
7.  | end
8.  | if degree[v]%2=1 then
9.  | | return NIL //存在度为奇数的顶点，该无向连通图无欧拉回路
10. | end
11. end
```

1-11：访问所有顶点和边表，时间复杂度 O（n+m）

欧拉回路求解算法

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

13：时间复杂度 O（n+m）

14-23：访问所有顶点和边表，时间复杂度 O（n+m）

时间复杂度 O（n+m）

提纲

- 7.1 问题引入及求解
- 7.2 图的定义与结构
- 7.3 图的存储实现
- 7.4 图的遍历
- 7.5 图的连通性
- 7.6 图的应用
- 7.7 拓展延伸*
- 7.8 应用场景

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)。

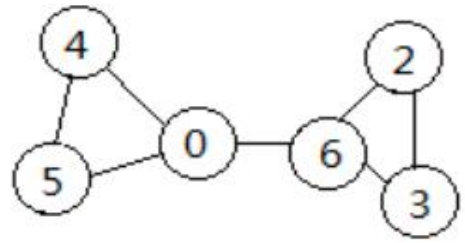


图 G18-1(d)

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**。

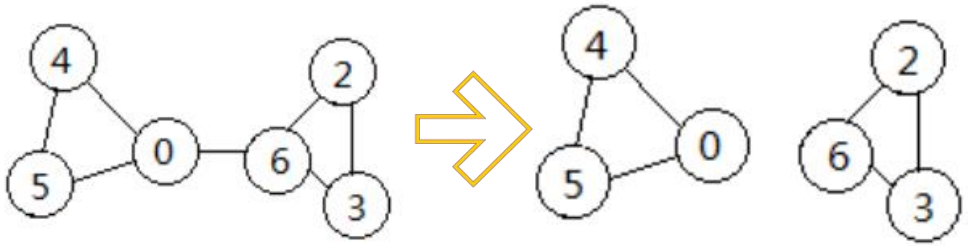


图 G18-1(d)　　图 G18-2(d)

双连通分量

点双连通图：若一个无向连通图中去掉任意一个顶点都不会改变图的连通性，即不存在割点，该图称作点双连通图。

点双连通分量：一个无向图中的每一个极大点双连通子图称该无向图的点双连通分量。

- 双连通分量包含了边双连通分量和点双连通分量两种类型，
- 双连通分量更深入地反映了无向连通图的连通程度。

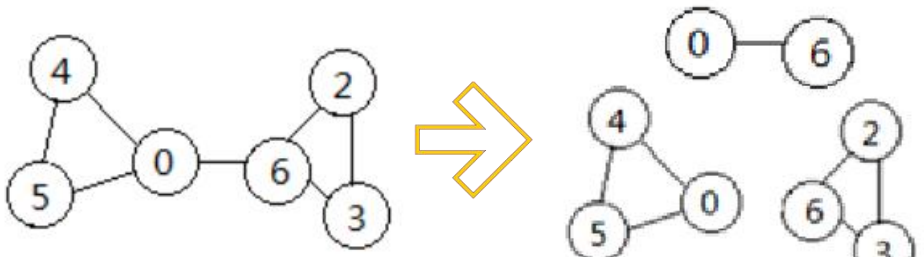


图 G18-1(d)　　图 G18-3(d)

边双连通分量的性质

- 任意一个边双连通分量中的一对顶点，它们之间至少有两条边不重复的路径。
- 从原图中去掉所有割边，剩下的连通分量为边双连通分量。
- 如果增加一条边，它能连接两个边双连通分量，这条边必是桥。
- 不同的边双连通分量之间没有公共点和边。
- 图中的割边不属于任何一个边双连通分量。
- 任意一个点和边最多属于一个边双连通分量。

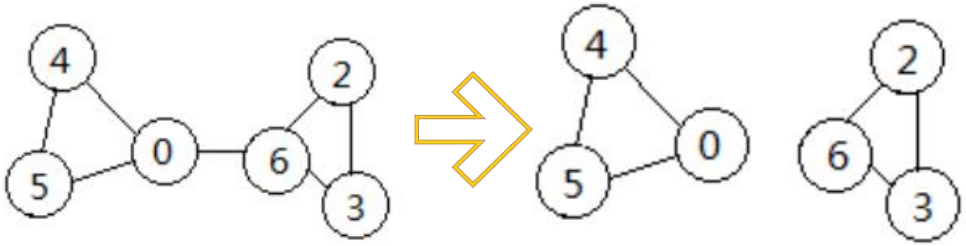


图 G18-1(d)　　图 G18-2(d)

点双连通分量的性质

- 在一个无向连通图中，任意两点间有两条或两条以上的点不重复路径。
- 点双连通分量内部的任意两条边都在同一个简单环中。
- 任意一条边，最多属于一个点连通分量。
- 不同的点双连通分量，有且只有一个公共点，且这个点一定是原图的割点。
- 任意割点都是至少两个不同点双连通分量的公共点。
- 离散的顶点，和两点一边的连通分量也属于点双连通分量，因为这些连通分量没有割点
- 除了两点一线情况，其余的点双连通分量一定是边双连通分量。

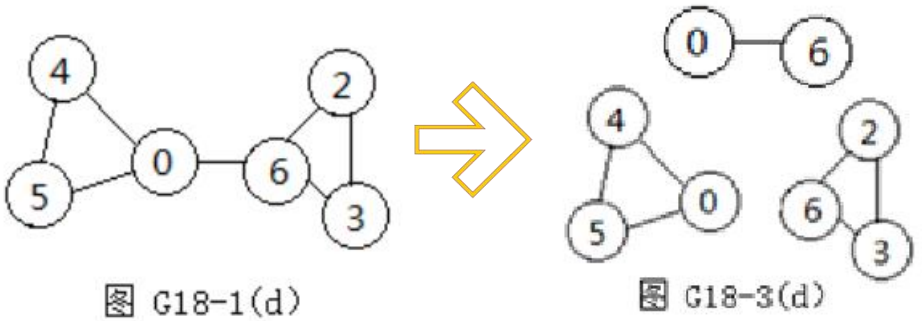


图 G18-1(d)　　图 G18-3(d)

无向连通图的割点和割边

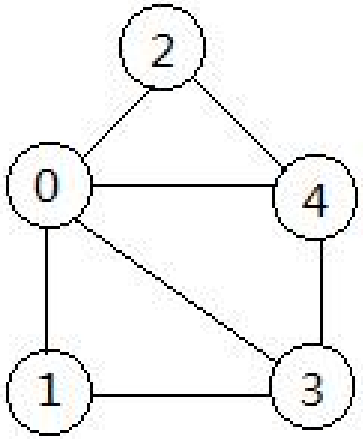


图 G18-1(a)

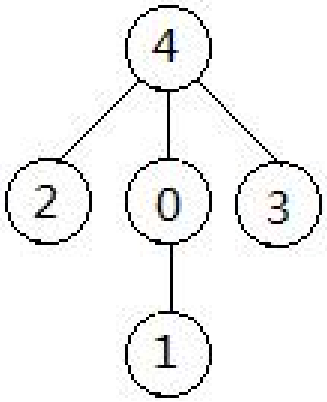


图 G18-1(b)

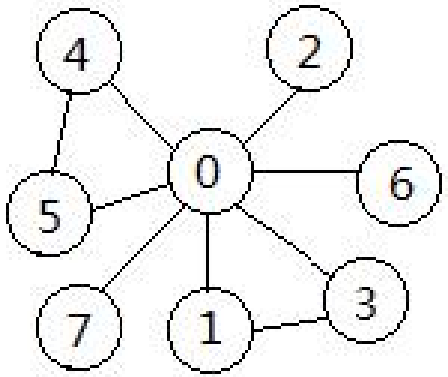


图 G18-1(c)

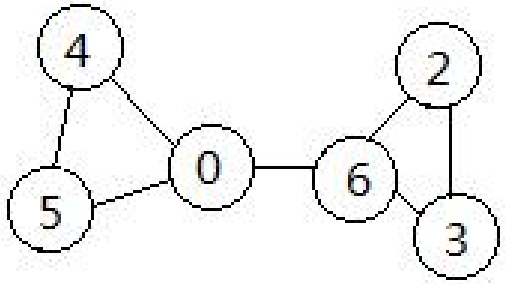


图 G18-1(d)

(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)

无向连通图的边双连通分量

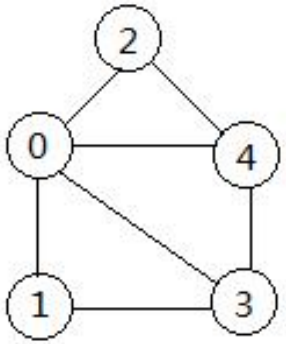


图 G18-1(a)

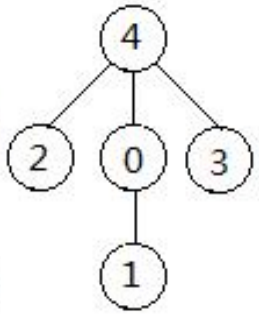


图 G18-1(b)

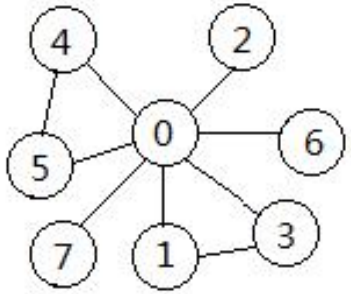


图 G18-1(c)

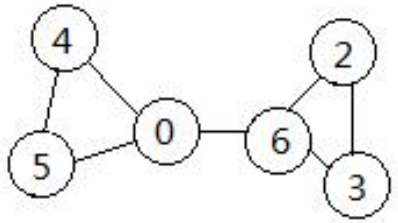


图 G18-1(d)

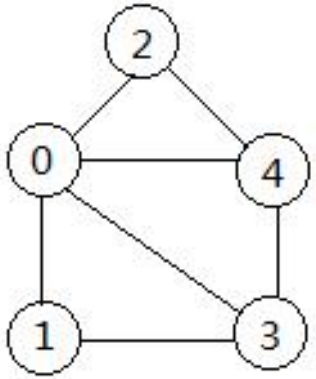


图 G18-2(a)

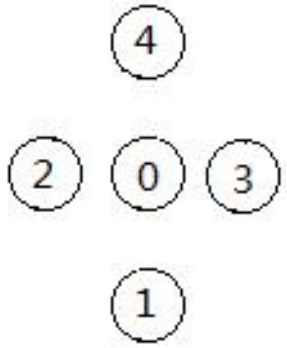


图 G18-2(b)

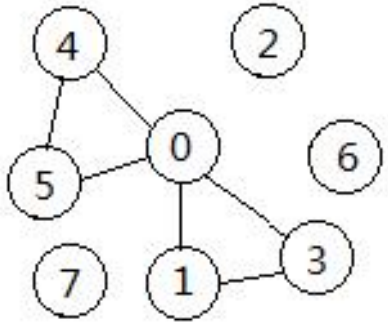


图 G18-2(c)

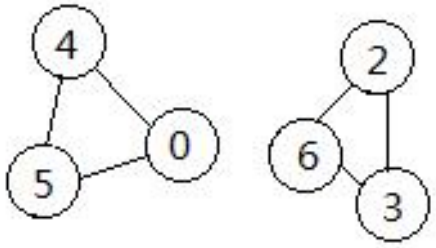


图 G18-2(d)

无向连通图的点双连通分量

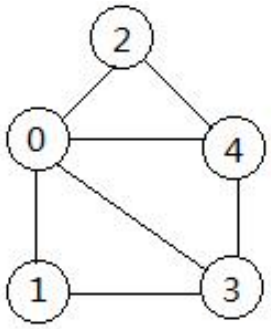


图 G18-1(a)

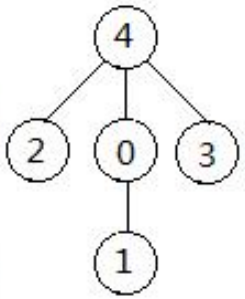


图 G18-1(b)

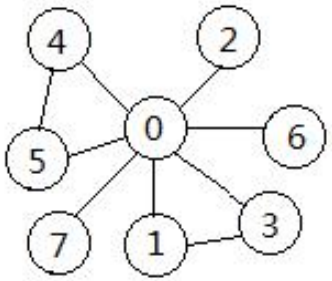


图 G18-1(c)

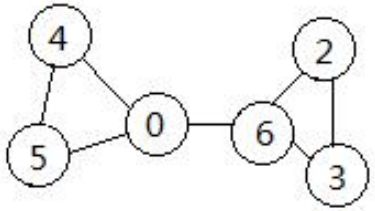


图 G18-1(d)

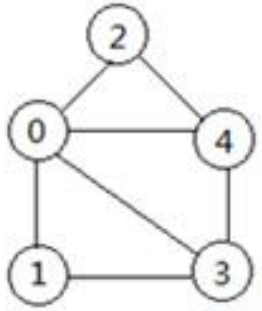


图 G18-3(a)

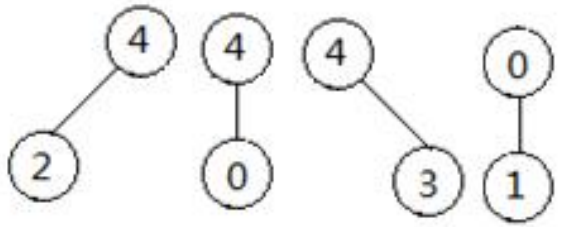
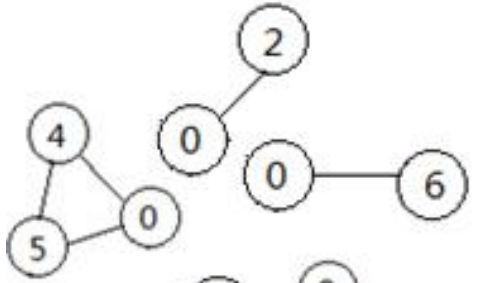


图 G18-3(b)



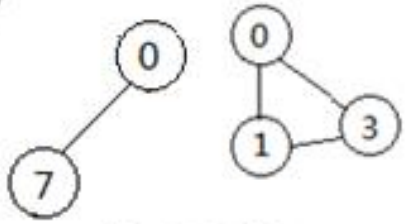


图 G18-3(c)

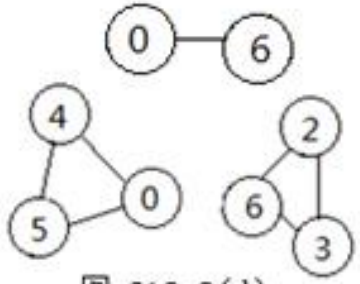


图 G18-3(d)

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。

Tarjan算法求无向连通图割点算法思路

- 对一个无向连通图进行深度优先遍历能获得该图的一个深度优先搜索生成树。
- 在深度优先遍历过程中，使用两个整型数组*dfn*和*low*。

  *dfn*数组记录每个顶点的访问顺序(也称时间戳)。

  *low*数组记录了该顶点及其子结点除了通过其父结点所能达到的具有最小时间戳*dfn*的祖先。

Tarjan算法求无向连通图割点算法思路

- *dfn*和*low*计算方法：

  用一个全局变量*count*，记录当前访问时间戳，初值为1。

  对每个结点*u*，初始时令其*low*[*u*]=*dfn*[*u*]。

  利用Tarjan算法更新每个结点的*low*值

- 每个结点获得了其*dfn*和*low*值后，可根据两种情况判断一个顶点是否是割点：

  1）如果顶点*u*是生成树的根，且*u*的孩子结点个数大于等于2，则顶点*u*是割点。

  2）如果顶点*u*不是生成树的根，对生成树中的边<*u*，*v*>，若满足*low*[*v*]≥*dfn*[*u*]，则顶点*u*即为割点。

Tarjan算法求无向连通图割点算法思路

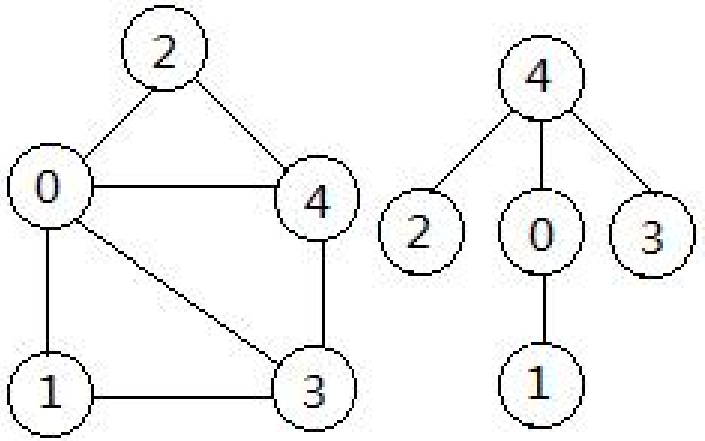
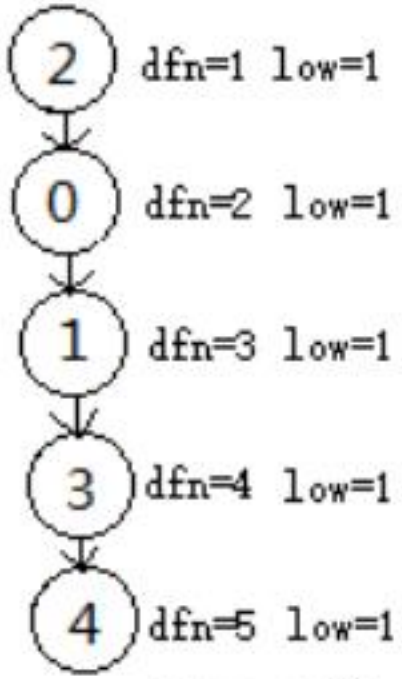
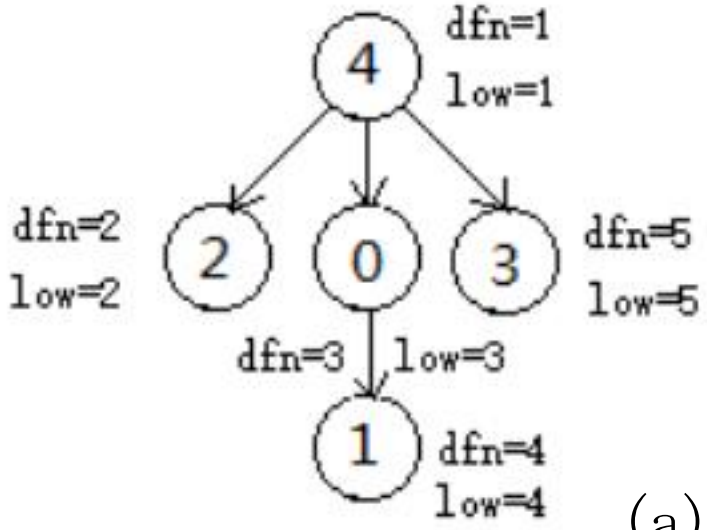


图 G18-1(a)

图 G18-1(b)



图G18-4(a)



图G18-4(b)

(a)割点：无
(b)割点：4、0
(c)割点：0
(d)割点：0、6

Tarjan算法求无向连通图割点算法思路

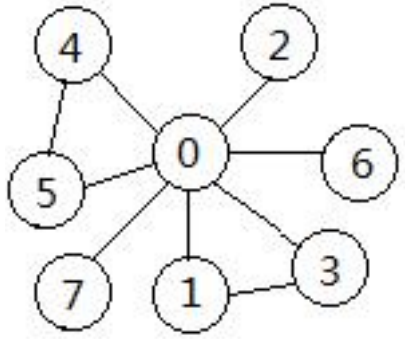


图 G18-1(c)

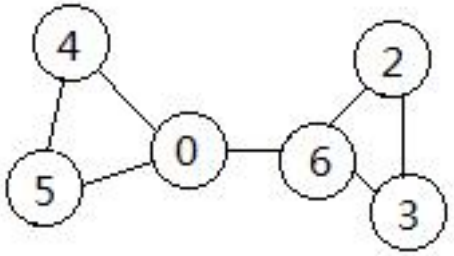
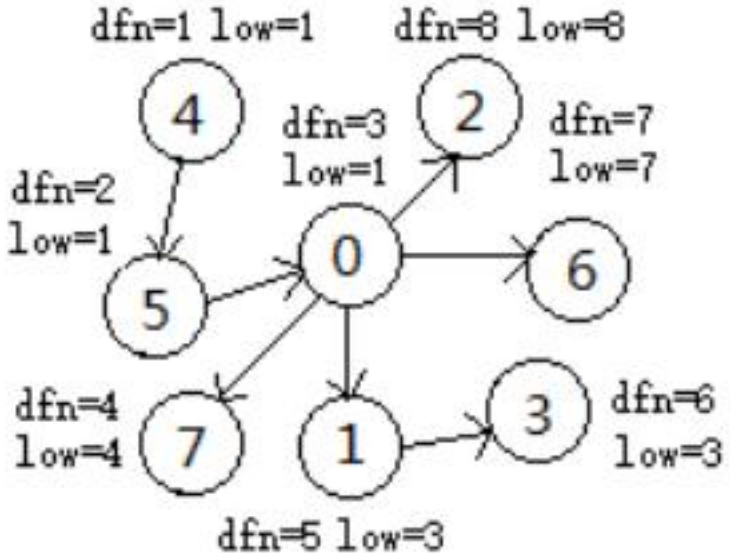
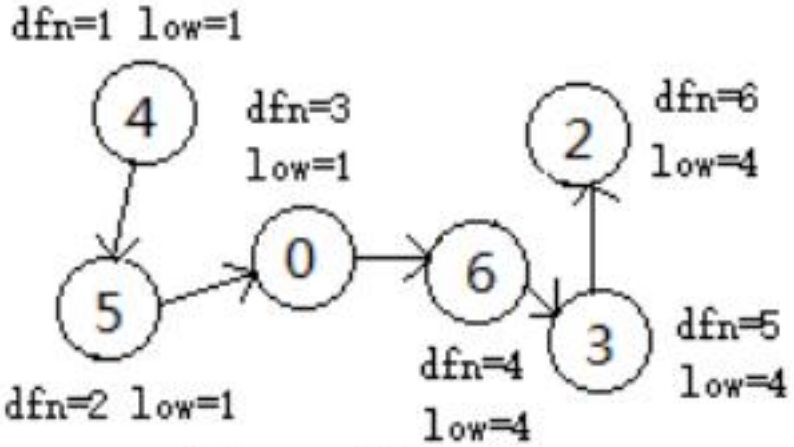


图 G18-1(d)



图G18-4(c)



图G18-4(d)

(a)割点：无
(b)割点：4、0
(c)割点：0
(d)割点：0、6

Tarjan算法求无向连通图割点算法

算法7-19： 利用深度优先遍历计算*dfn*和*low*的值 DfnAndLow(*graph,v,parent*)
输入: 图*graph*，起始顶点*v*，DFS中*v*的父结点*parent*
输出： *dfn*和*low*数组的值
全局变量： 数组*dfn, low, parents, visited*，整型变量*count*的初值为1，
parent的初值为-1

1. *visited*[*v*]←**true**
2. *dfn*[*v*] ←*count*
3. *low*[*v*] ←*count*
4. *parents*[*v*] ← *parent*
5. *count* ←*count*+*1*
6. *p* ←*graph.ver_list[v].adj* //沿v向下搜索

Tarjan算法求无向连通图割点算法

```
7.  while p≠ NIL do
8.  |  if visited[p.dest]=false then
9.  |  |  DfnAndLow(graph,p.dest,v)
10. |  |  low[v] ← Min(low[v], low[p.dest])
11. |  else
12. |  |  if p.dest ≠ parent then
13. |  |  |  low[v] ← Min(low[v], dfn[p.dest])
14. |  |  end
15. |  end
16. |  p ← p.next
17. end
```

访问到每个顶点一次，每条边一次，
时间复杂度 O（n+m）

Tarjan算法求无向连通图割点算法

算法7-20: 求割点的Tarjan算法 ArticulationPoint (*graph, start*)
输入: 图*graph*，起始顶点*start*
输出： 图*graph*的割点
全局变量： 数组*dfn, low, parents, visited*，整型变量*count*的初值为1

1. **for** *v*← 0 **to** *graph.n_verts*-1 **do**
2. | *visited*[*v*] ← **false** //初始化各顶点的访问标志为未访问
3. **end**
4. DfnAndLow(*graph, start,* -1)
5. *n_child* ← 0//对根结点的孩子结点计数

1-3：时间复杂度 O（n）

4-5：时间复杂度 O（n+m）

Tarjan算法求无向连通图割点算法

```
6.  for ← 0 to graph.n_verts-1 do
7.  |  if parents[v]≠ -1 then //若v不是根结点
8.  |  |  if parents[v]=start then
9.  |  |  |  n_child ← n_child + 1
10. |  |  else
11. |  |  |  if low[v] ≥ dfn[parents[v]] then
12. |  |  |  |  print graph.ver_list[parents[v]].data //打印割点
13. |  |  |  end
14. |  |  end
15. |  end
16. end
17. if n_child≥2 then //判断根结点是否割点
18. |  print graph.ver_list[start].data  //打印割点
19. end
```

6-19：时间复杂度 O（n）

算法总的时间复杂度 O（n+m）

Tarjan算法求无向连通图割边算法

算法7-21: 求割边的Tarjan算法 ArticulationEdge(*graph, start*)
输入: 图*graph*，起始顶点*start*
输出： 图*graph*的割边
全局变量： 数组*dfn, low, parents, visited*，整型变量*count*的初值为1。

```
1.  for v← 0 to graph.n_verts-1 do
2.  |  visited[v] ← false  //初始化各顶点的访问标志为未访问
3.  end
4.  DfnAndLow(graph, start, -1)
```

1-4：时间复杂度 O（n+m）

Tarjan算法求无向连通图割边算法

```
5.  for v ← 0 to graph.n_verts-1 do
6.  |  if parents[v] ≠ -1 then  //若v不是根结点
7.  |  |  if low[v] ≥ dfn[parents[v]] then
8.  |  |  |  print (graph.ver_list[parents[v]].data, graph.ver_list[v].data) //打印割边
9.  |  |  end
10. |  end
11. end
```

5-11：时间复杂度 O（n）

算法总的时间复杂度 O（n+m）

Tarjan算法求无向连通图是否双连通图算法思路

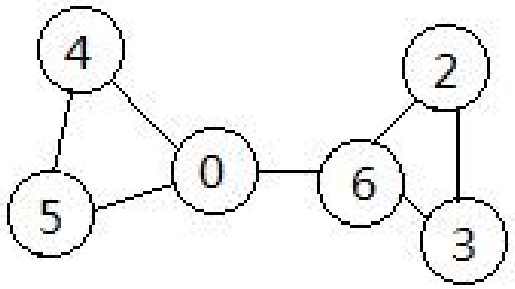
1. 基于Tarjan算法判断出割边，同时对割边计数，如果割边数量为0，则该无向连通图为边双连通图。
2. 基于Tarjan算法判断出割点，同时对割点计数，如果割点数量为0，则该无向连通图为点双连通图。

Tarjan算法求无向连通图边双连通分量算法思路

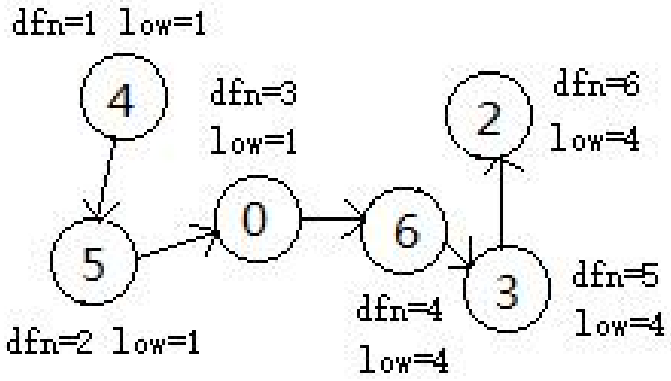
可用两种方法：

1. 基于Tarjan算法判断出割边，去除所有割边，便得到一个无向非连通图。利用深度或者广度优先遍历可方便地求得该图的所有连通分量，每个连通分量便是原来的无向连通图的边双连通分量。
2. 凡是*low*值相同的点和原图中这些点之间的边都在同一个边双连通分量中，深度优先搜索树中有几种*low*的值就有几个边双连通分量。

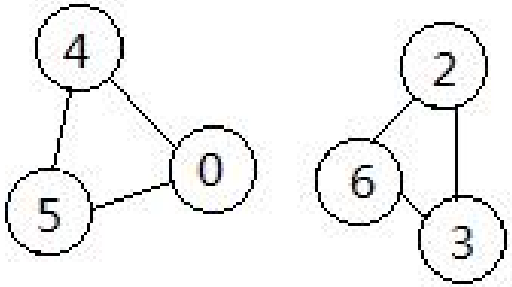
Tarjan算法求无向连通图边双连通分量算法思路



(a) 无向连通图



(b) Tarjan求dfn和low



(c) 边双连通分量

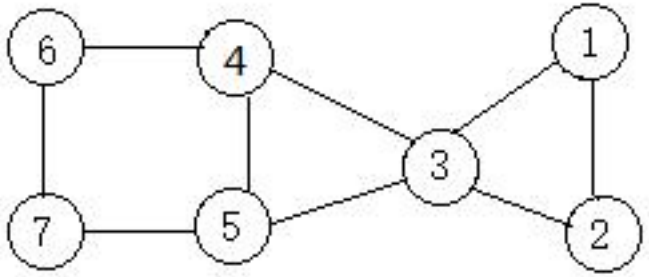
Tarjan算法求无向连通图边点双连通分量算法思路

- 图中的非割点：根据其low值，凡是low值相同的必在同一个点双连通分量中，非割点有几种low值就有几个点双连通分量。
- 图中的割点：

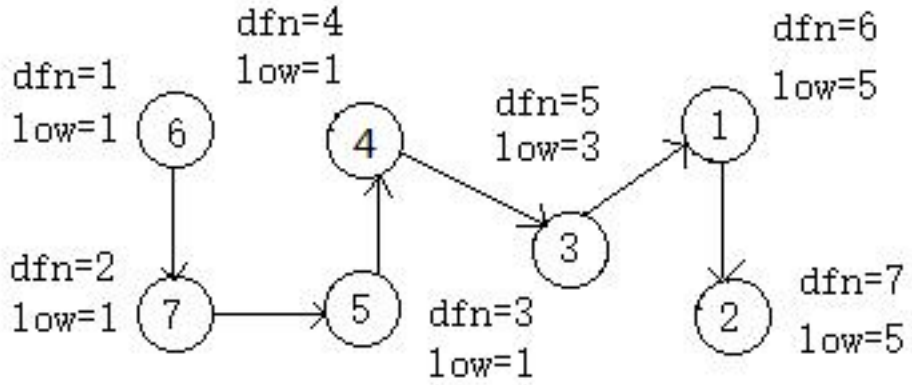
1. 其dfn值和哪个点双连通分量的low值相同，它也属于哪个点双连通分量。
2. 其low值和哪个点的dfn值相同，也和它在同一个点双连通分量中。

故可以看出，一个割点至少属于2个点双连通分量。

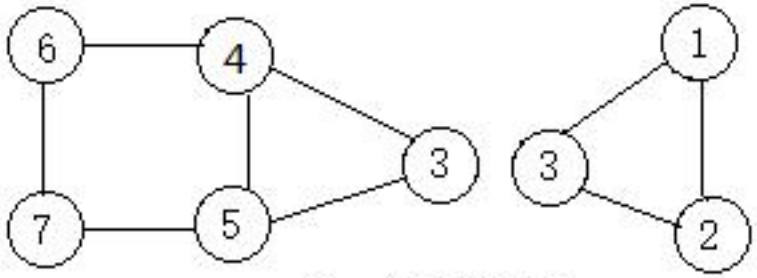
Tarjan算法求无向连通图边点连通分量实例



(a) 无向连通图



(b) 按照Tarjan算法求得dfn和low



(c) 点连通分量

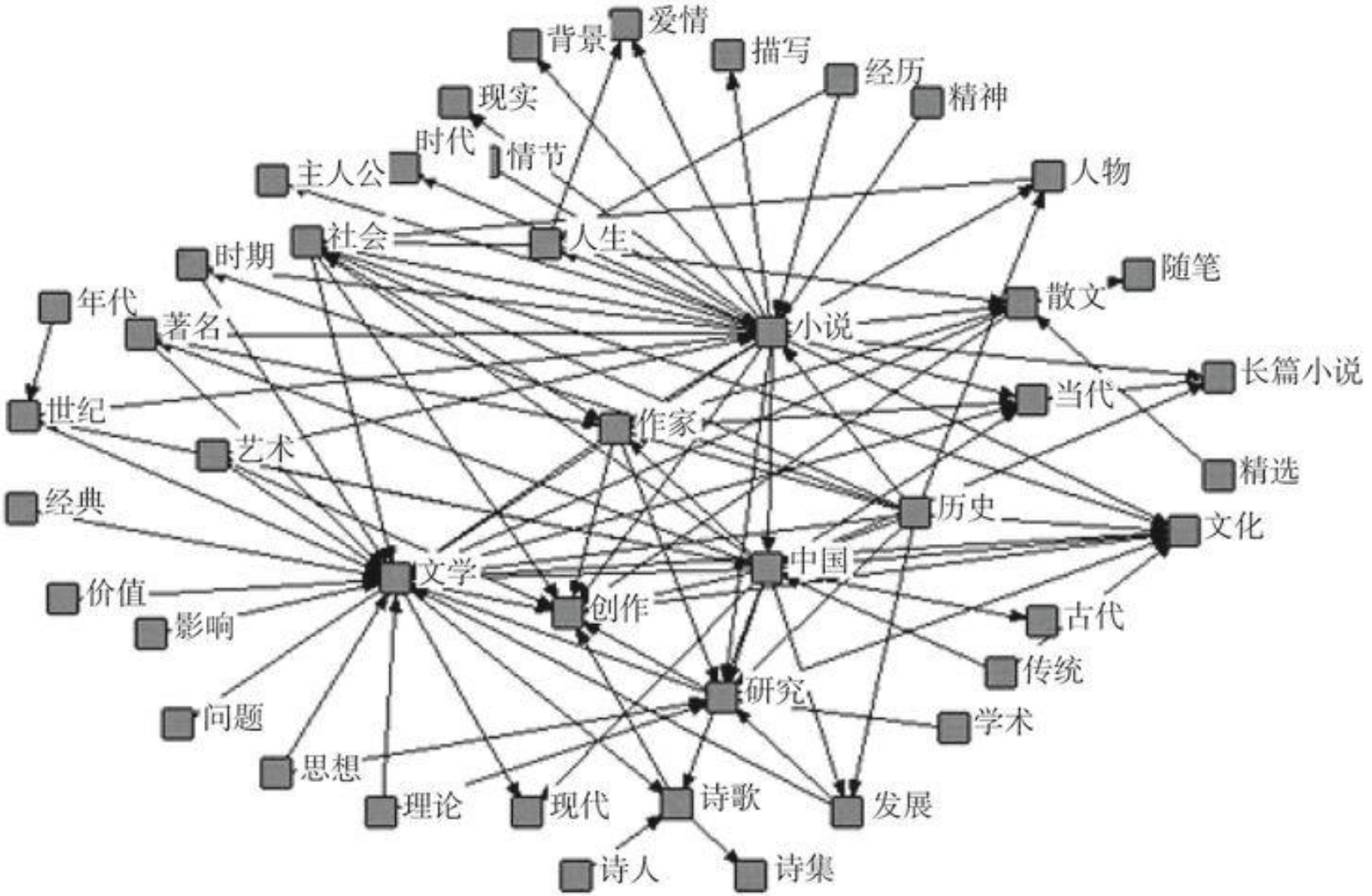
提纲

- 7.1　问题引入及求解
- 7.2　图的定义与结构
- 7.3　图的存储实现
- 7.4　图的遍历
- 7.5　图的连通性
- 7.6　图的应用
- 7.7　拓展延伸*
- 7.8　应用场景

语义网络

1. 语义网络（Semantic Network）是由Quillian于上世纪60年代提出的知识表达模式，它用相互连接的结点和边来表示知识。
2. 结点表示对象、概念，边表示结点之间的关系，边上附加的信息可体现出两个结点间的语义关联程度。
3. 典型应用：WordNet、HotNet

语义网络


爱情
背景
描写
经历
精神
现实
时代
情节
主人公
人物
社会
人生
时期
随笔
年代
散文
著名
小说
长篇小说
当代
世纪
作家
艺术
精选
经典
历史
文化
中国
文学
价值
创作
古代
影响
传统
研究
问题
学术
思想
理论
现代
诗歌
发展
诗人
诗集

小结

- 图是一种很常见的数据结构，有着广泛的用途。
- 本章介绍了邻接矩阵和邻接表，这是两种最常用的存储方法，并给出了这两种表示方式下的基本操作实现。
- 图的一个重要的操作是遍历所有的结点，图的遍历比其他数据结构的遍历都复杂，本章介绍了两种遍历方法：深度优先搜索和广度优先搜索，并给出了它们在邻接表的存储方式下的实现。
- 图的很多应用都是基于遍历实现的，本章还介绍了基于遍历检测图的连通性、寻找无向图的欧拉回路、寻找有向图的强连通分量等方面的内容。

小结

	邻接矩阵	邻接表	十字链表	邻接多重表
空间复杂度	无向图：可用对称矩阵压缩 $O(n(n+1)/2)$；有向图：$O(n^2)$	无向图：$O(n+2e)$ 有向图：$O(n+e)$	$O(n+e)$	$O(n+e)$
求顶点的度(找邻边)	无向图：查找序号对应的行$O(n)$ 有向图：入度查找行，出度查找列$O(n)$	无向图：遍历边链表$O(n)$ 有向图：入度遍历边链表$O(n)$，出度遍历整个邻接表$O(n+e)$	很方便	很方便
删除边或顶点	删除边很容易， 删除顶点需要移动大量元素	都不方便	很方便	很方便
求边的数量	遍历整个矩阵$O(n^2)$	按顶点表遍历所有边链表（有向图：$O(n+e)$，无向图：$(n+2e)$)	-	-
判定边是否存在	查找元素A[i][j] $O(1)$	查找边链表 $O(n)$	-	-
适用于	稠密图	稀疏图或非稠密图	只能是有向图	只能是无向图
表示方式	唯一	不唯一	不唯一	不唯一

小结

- 遍历算法经过的边可能是一棵生成树，也可能是一个生成森林
- 深度优先搜索算法借助于栈结构（递归）来实现，适用于只求一个解的问题
- 广度优先搜索算法借助于队列结构实现，适用于求距离起点最近的解
- 两种遍历算法的过程实质都是通过边来找邻接点的过程
- 两种遍历算法的不同之处仅仅在于对顶点的访问顺序不同
- 使用邻接矩阵存储时，时间复杂度为$O(n^2)$
- 使用邻接表存储时，时间复杂度为：有向图$O(n+e)$/无向图$O(n+2e)$

图高频必刷题(LeetCode)

分类	题号	题目	考察点（高频原因）	难度
图的存储与实现	997	Find the Town Judge	有向图入度出度统计、邻接表存储	Easy
图的遍历（核心）	463	Island Perimeter	图的遍历、几何属性计算	Easy
	994	Rotting Oranges	图的广度优先搜索	Medium
	797	All Paths From Source to Target	有向图 DFS 回溯遍历、路径记录	Medium
	841	Keys and Rooms	有向图 DFS/BFS 遍历、可达性判断	Medium
图的连通性	1791	Find Center of Star Graph	无向图的节点度数特性	Easy
	200	Number of Islands	图的遍历、无向图连通分量计数、边界条件处理	Medium
	547	Number of Provinces	无向图连通性判断、DFS/BFS 统计连通分量	Medium
	1020	Number of Enclaves	DFS/BFS 遍历、连通性筛选	Medium
	827	Making A Large Island	图的遍历、连通分量合并	Hard
欧拉回路	332	Reconstruct Itinerary	欧拉路径求解、Hierholzer 算法	Hard
Tarjan 算法	1382	Balance a Binary Search Tree	无向图割点判断、树结构转化	Medium