计算机领域本科教育教学改革试点
工作计划（“101计划”）研究成果

数据结构

授课教师：

湖南大学 信息科学与工程学院

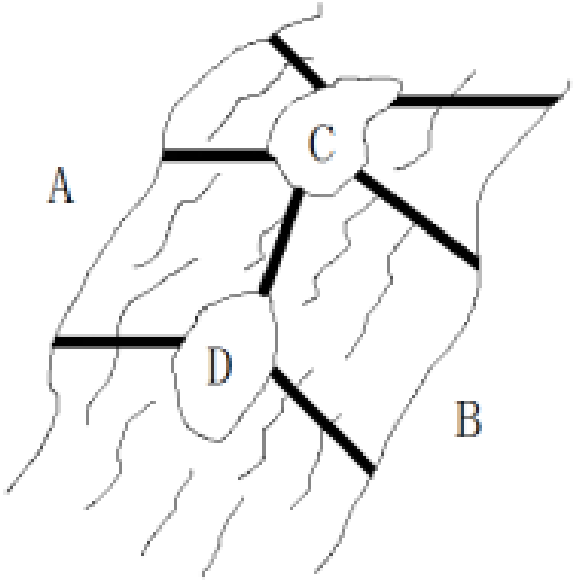
第 7 章

图

提纲

问题引入：哥尼斯堡七桥问题

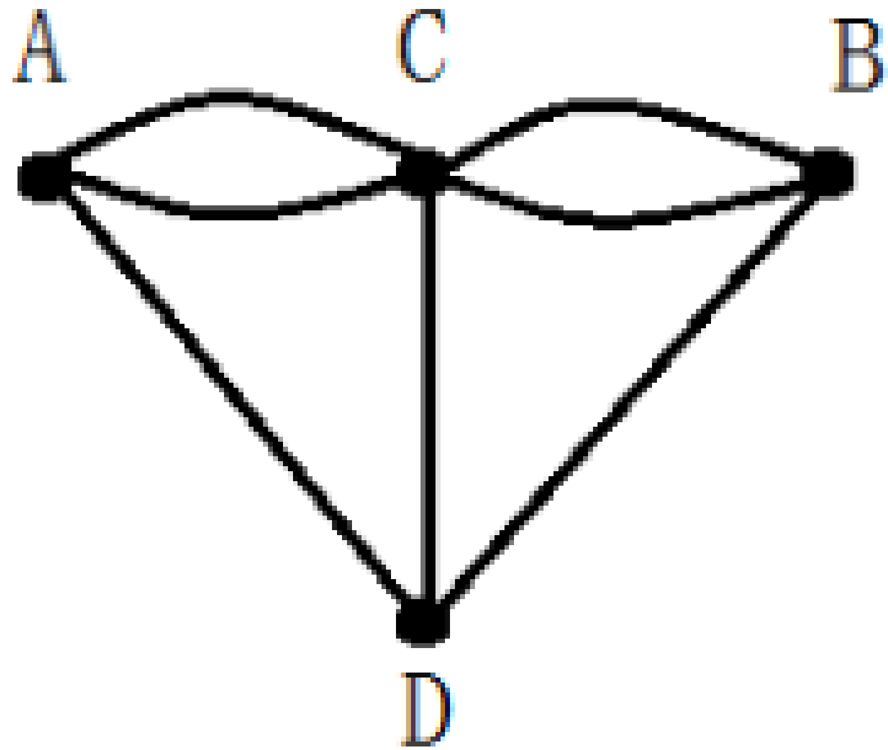
- **问题**：18世纪的哥尼斯堡，一条河流穿城而过，城市除被一分为二外，还包含了河中的两个小岛，河上有七座桥把这些陆地和岛屿联系了起来，可否从一个陆地或岛屿出发，一次经过全部的七座桥且每座桥只走一遍，最后还能回到出发点?
- **问题分析**：2个问题，如何判断是否有解？如果有解，如何找到解?



问题求解：哥尼斯堡七桥问题

问题抽象：抽象地表达和描述-陆地或者岛屿为元素（顶点），桥为元素间关系（边）。涉及到的数学工具-图

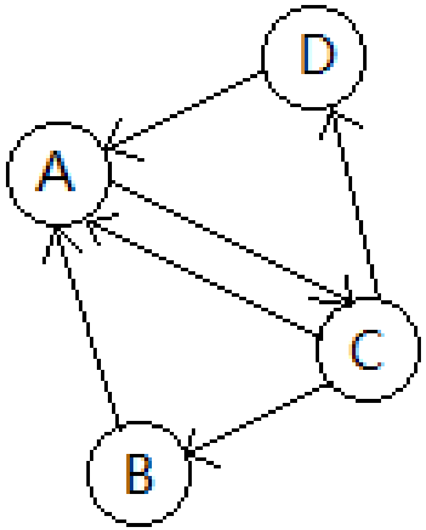
问题转化为：是否存在从任意一个顶点出发，经过每条边一次且仅一次，最后回到该顶点的路径（一笔画问题）。



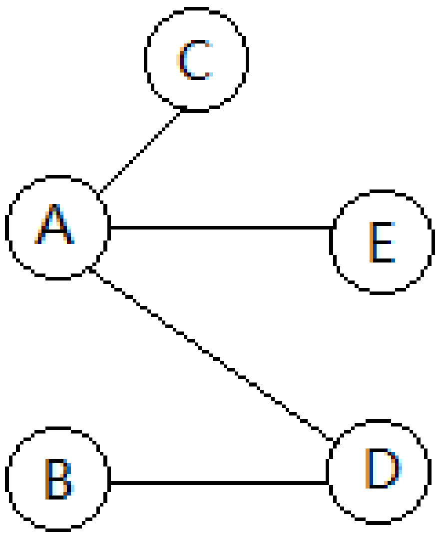
图的定义

- **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



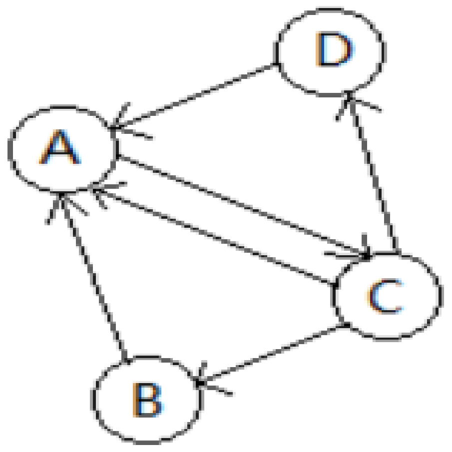
G2

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。

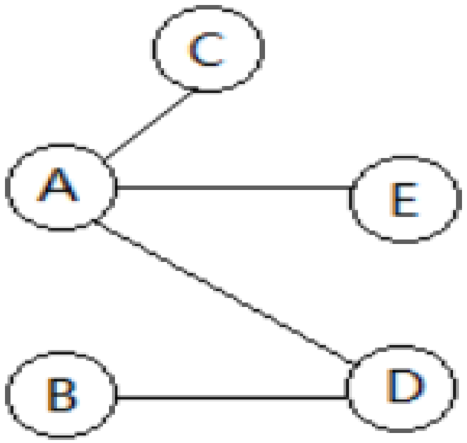


G1

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。



G2

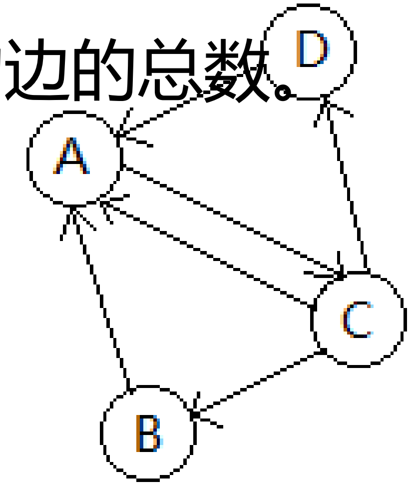
图的术语

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。

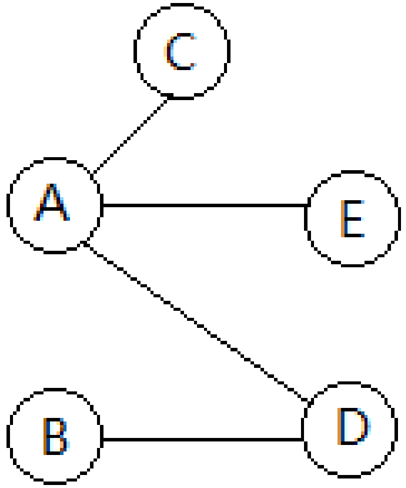
出度：有向图中一个顶点的**出度**是指由该顶点射出的有向边的条数。

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

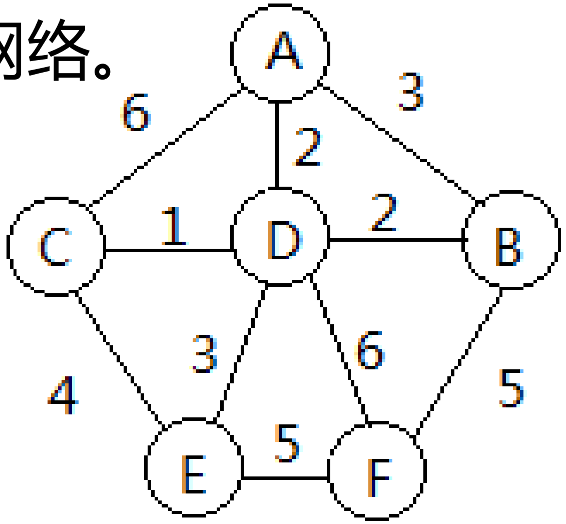
无向完全图：当无向图中边的条数达到最大，为n（n - 1）/ 2时的图。

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。

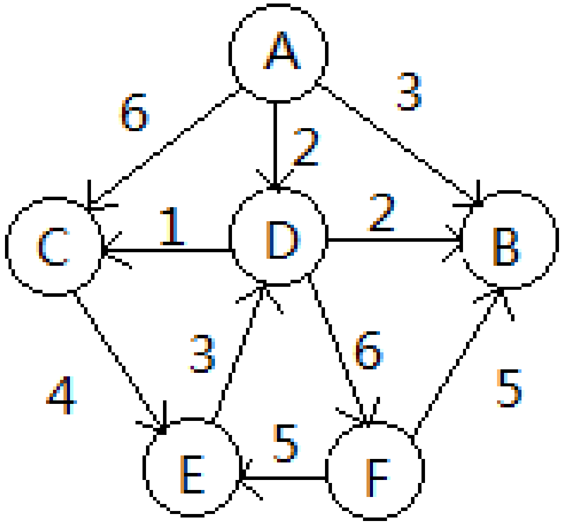
加权有向图：边上带有权重的有向图。

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。



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；

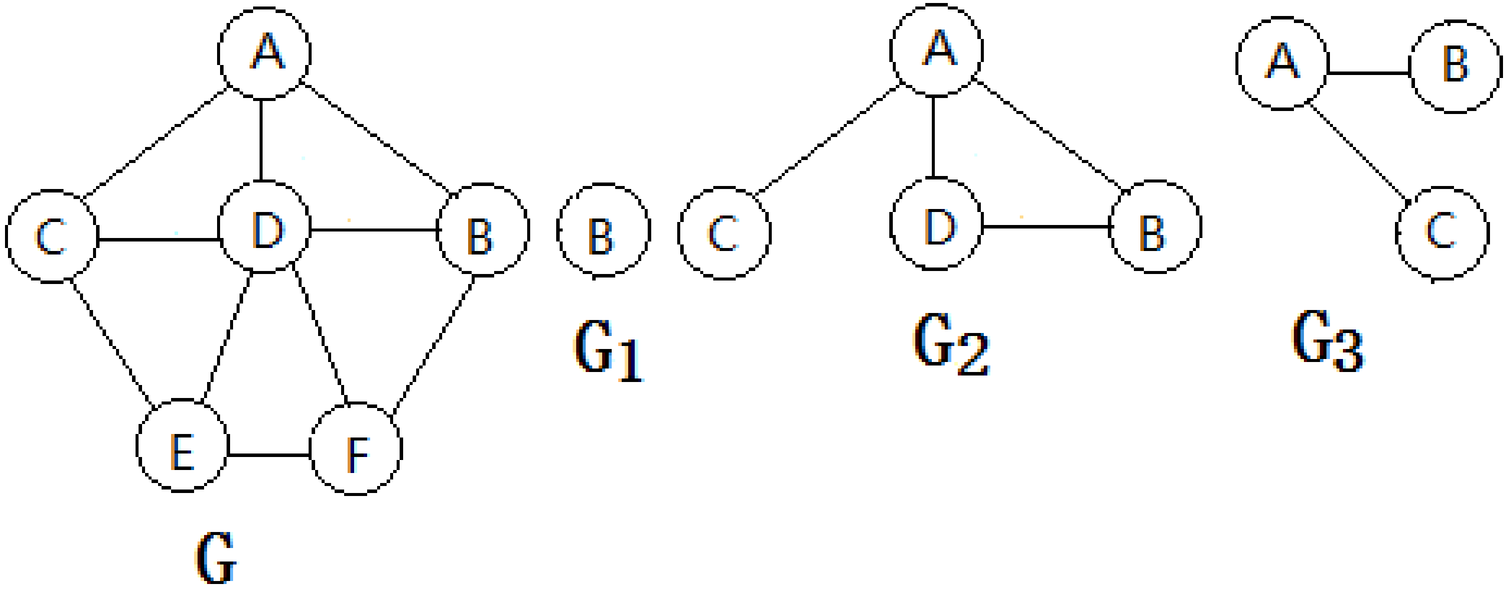
- 如果边上有权重，路径长度也可以用路径上所有边的权重之和来表示。

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。

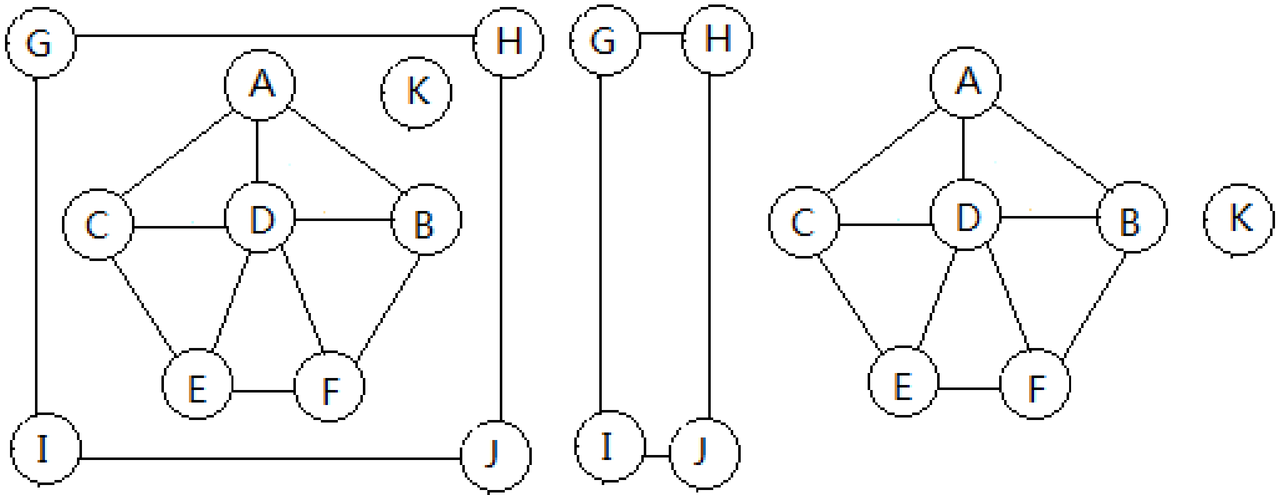
图的术语

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。

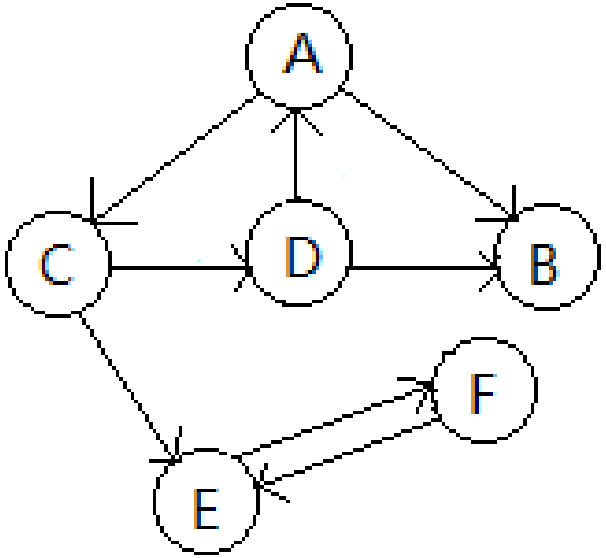


(a) 无向图G5　(b)　(c)　(d)

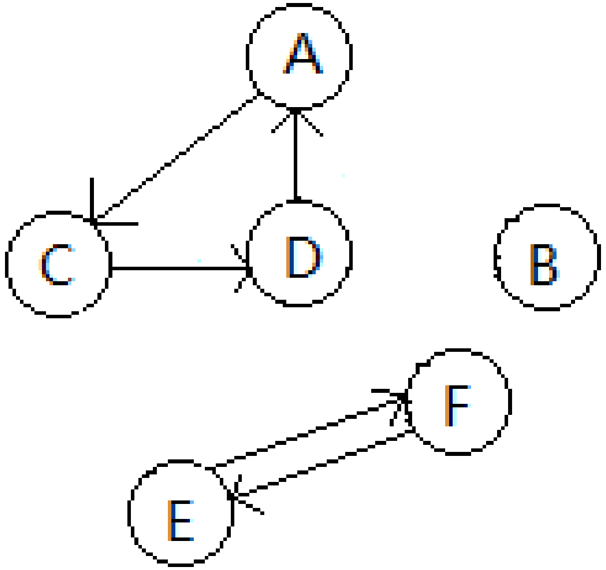
图的术语

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

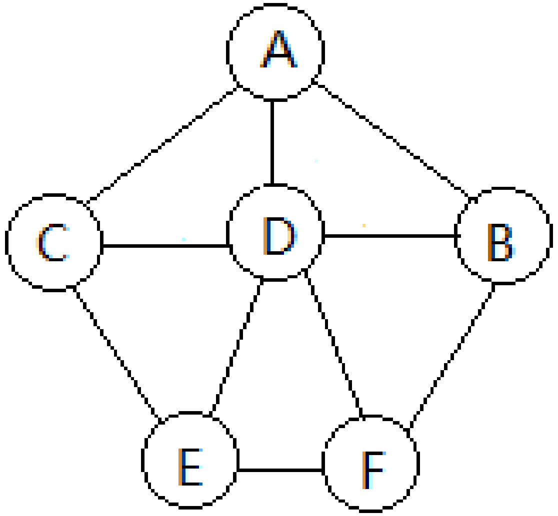


(b) G6的三个强连通分量

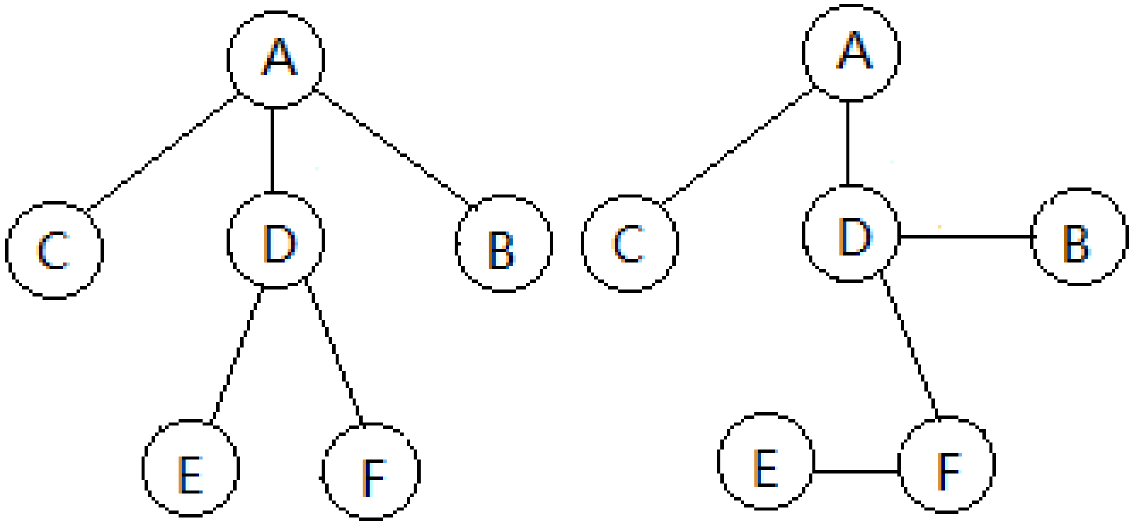
图的术语

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：**
  - $\{v_i | v_i \in \text{ElemSet}, i=1,2,3,\ldots\ldots n, n > 0\}$ 或 Φ；ElemSet为顶点集合。
- **数据关系：**
  - $\{<v_i, v_j>$或$(v_i, v_j) | v_i$、$v_j \in$ElemSet, 且$P(v_i, v_j)$, $i, j=1,2,3,\ldots\ldots n\}$,
  - 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。

图结构的A D T

- **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

图结构的ADT

- 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

}

图的存储

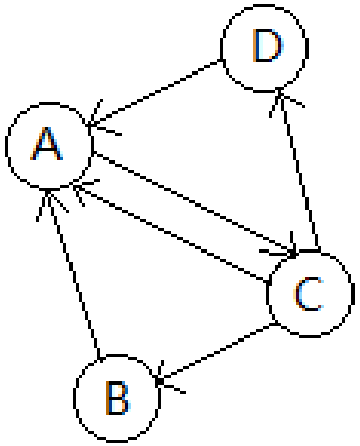
- 图的存储既要考虑到顶点的存储又要考虑到边的存储。
- 如果按照线性结构和树结构的存储思路，找到一个类似的、既能同时存储顶点又能存储表示顶点间关系的边的结构就非常困难。不妨换个思路，将顶点和边的存储独立开来，顶点归顶点存、边归边存。
- 顶点用一维数组存，边用二维矩阵存 ---**邻接矩阵存储法**。

  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。
- 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。
- 无向图的邻接矩阵是以主对角线为轴对称的。

邻接矩阵

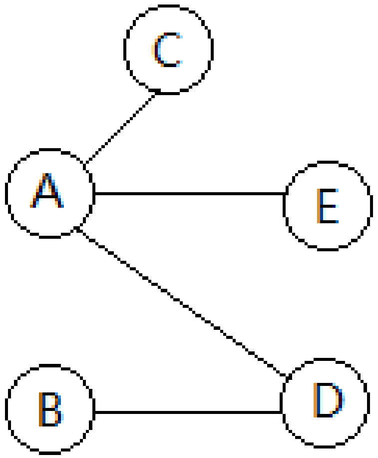


G8

0	1	2	3
A	B	C	D

$$\begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\ \left(\begin{array}{cccc} 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{array} \right) \end{array}$$

G8的邻接矩阵



G9

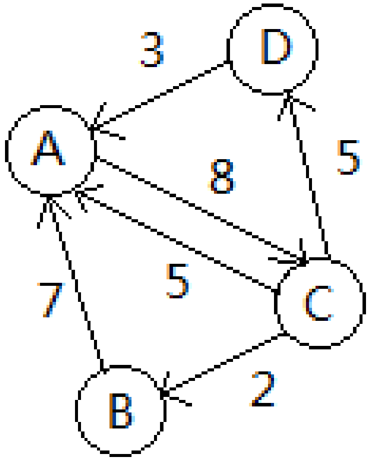
0	1	2	3	4
A	B	C	D	E

$$\begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{ccccc} 0 & 1 & 2 & 3 & 4 \end{array} \\ \left(\begin{array}{ccccc} 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \end{array} \right) \end{array}$$

G9的邻接矩阵

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] = ∞。
- 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

$$
\begin{array}{c c}
 & \begin{matrix} 0 & 1 & 2 & 3 \end{matrix} \\
\begin{matrix} 0 \\ 1 \\ 2 \\ 3 \end{matrix} & \begin{pmatrix} 0 & \infty & 8 & \infty \\ 7 & 0 & \infty & \infty \\ 5 & 2 & 0 & 5 \\ 3 & \infty & \infty & 0 \end{pmatrix}
\end{array}
$$

G10的邻接矩阵

邻接矩阵和加权邻接矩阵的优缺点

- 判断任意二个顶点v_i和v_j之间是否存在一条边非常容易，直接看a[i][j]，O(1)的时间复杂度。
- 在用邻接矩阵表示无向图和有向图时，可以很容易地得到顶点的度或者出度、入度。
- 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，空间消耗太大。

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）：

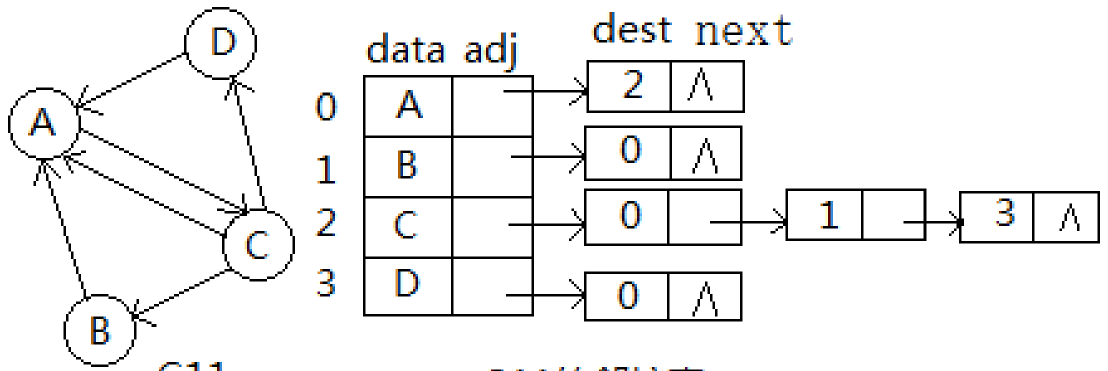
  对有向图，采用邻接矩阵是合适的。

  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。

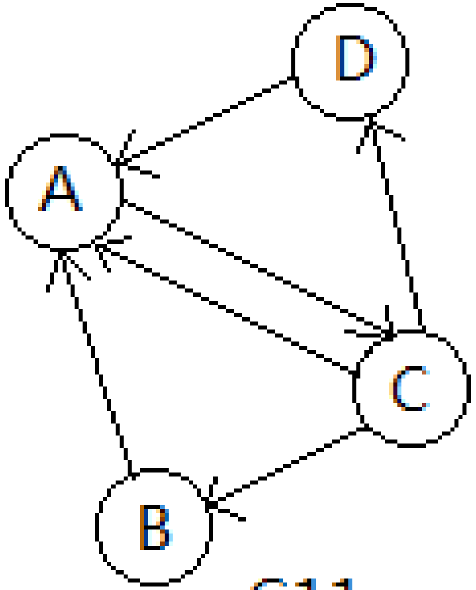
邻接表

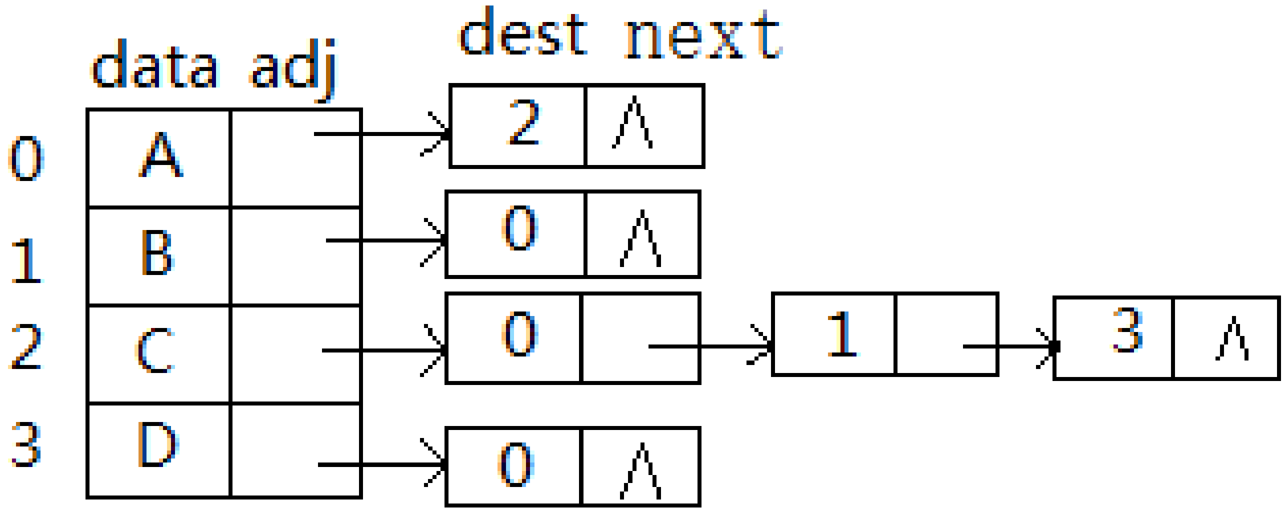
- 顶点依然用一个一维数组来存储，而边的存储是将由同一个顶点出发的所有边组成一条单链表。
- 存储顶点的一维数组称**顶点表**，存储边信息的单链表称**边表**。一个图由顶点表和边表共同表示。
- 顶点表不仅保存各个顶点的信息，还保存由该顶点射出的边形成的单链表中首结点的地址（首指针），这种方法称**邻接表**表示法。



G11的邻接表

邻接表





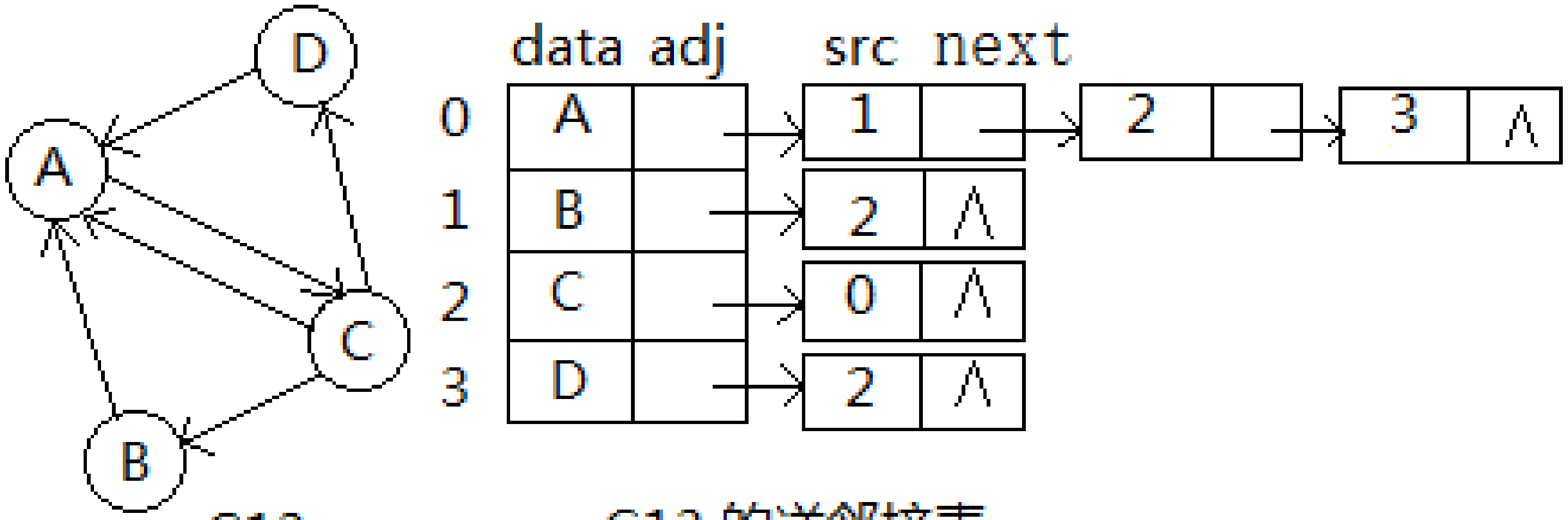
G11的邻接表

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

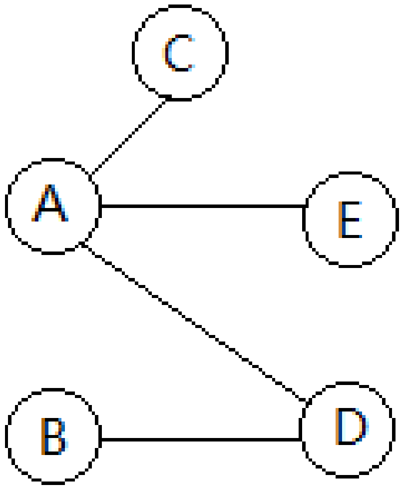
G13 的逆邻接表

另外一种邻接表存储

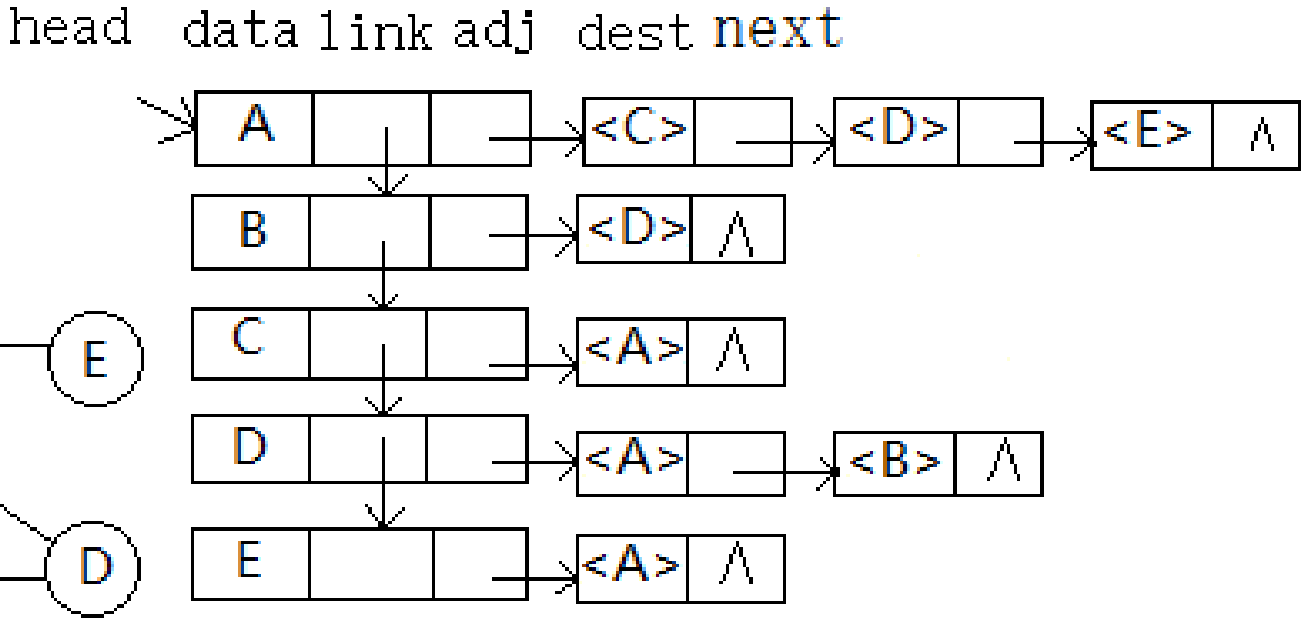
邻接表中，顶点表用了一维数组，图初始化时需要预估数组规模。

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。



G14



G14的邻接表

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。

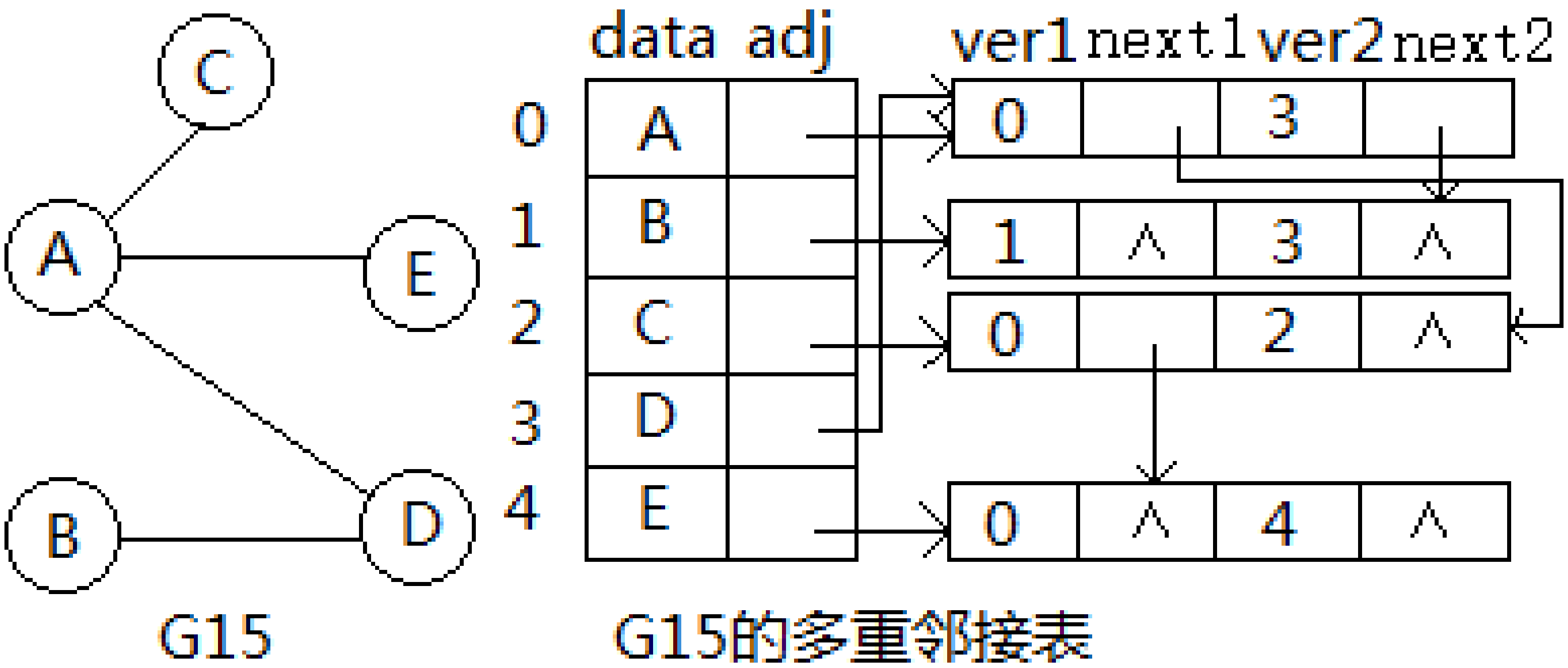
1）空间浪费，

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。

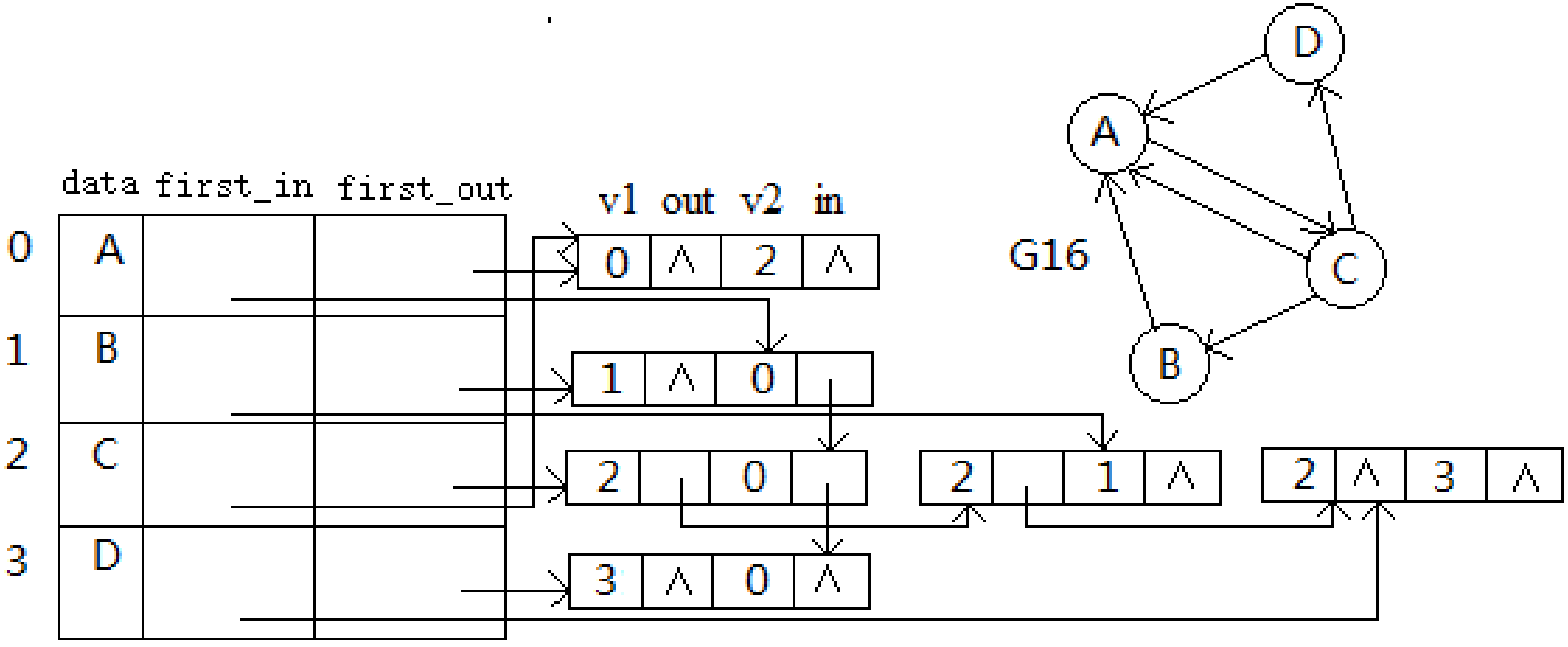
邻接多重表*



边的next1指针指向链接ver1的下一个边，next2指针指向链接ver2的下一个边

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，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。所需要花费的时间通常既和顶点的个数n有关

- **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** $u \neq v$ 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（min(n,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

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

插到邻接表的第一个位置

图用邻接表表示时部分基本操作算法描述

```
| if graph.directed=false then  //如果是无向图，还要将u插入v的边表中
| | p ← new EdgeNode
| | p.dest ←u
| | p.weight ← weight
| | p.next← graph.ver_list[v].adj
| | graph.ver_list[v].adj ←p
| end
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（min(m,n))

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*)

```
for u←0 to graph.n_verts-1 do //删除射入顶点v的边
|       p ←graph.ver_list[u].adj
|       if p≠NIL then    //非空链表
|   |       if p.dest=v then //首结点为射入顶点v的边
|   |   |       graph.ver_list[u].adj ←p.next
|   |   |       delete p
|   |   |       count  count+1
|   |       else //非首结点
|   |   |   while p.next≠NIL  且 p.next.dest≠v do //找到射入顶点v的边
|   |   |   |  p ← p.next
|   |   |   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，后续会将
                           //顶点表中最后一个顶点移到位置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）

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

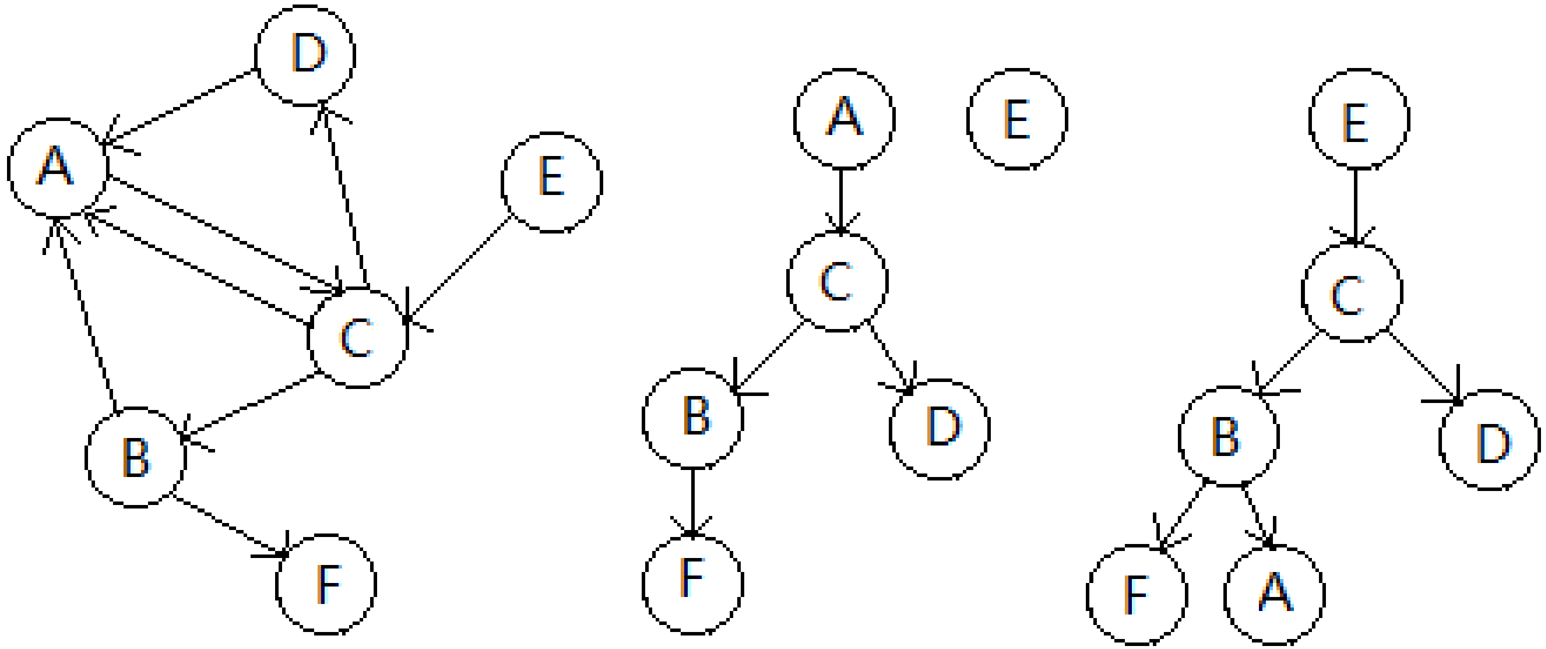
DFS（Depth First Search）

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



(a) G17　　(b)　　(c)

- 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** //可能不连通，从单节点出发可能不能遍历所有结点
5. | **if** visited[v]=**false then**
6. | | DFS(graph, v, visited) //该函数见下页
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    //访问该顶点
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) //递归
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）。

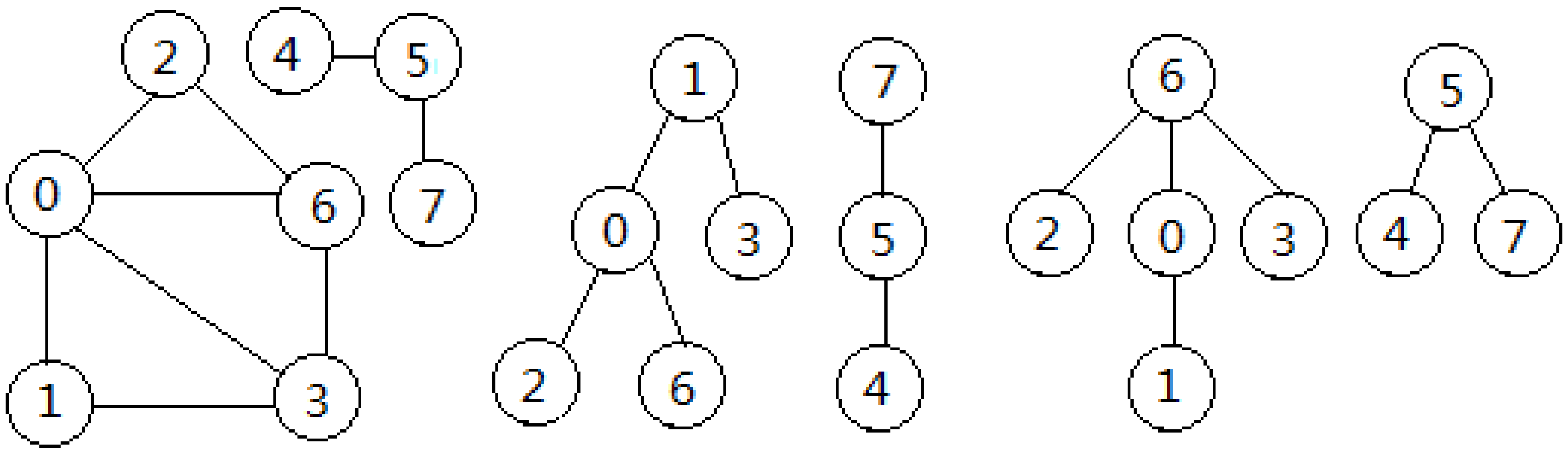
广度优先遍历

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) //见下页，每个联通分量调用一次这个函数
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 //访问该结点
7.  | | visit(graph, u)
8.  | | p ← graph.ver_list[u].adj //将所有未访问邻居放入队列
```

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

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。

1. 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。

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。

```
for v← 0 to graph.n_verts-1 do //初始化各顶点的访问标志为未访问
     visited[v]←false
end
count ←0
for v ← 0 to graph.n_verts-1 do
 | if visited[v]=false then
 | |  count ← count +1  //每个联通分量都调用一次BFS函数
 | |  BFS(graph, v, visited) |
 | end
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

```
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
```

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（课后自学）

- 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（课后自学）

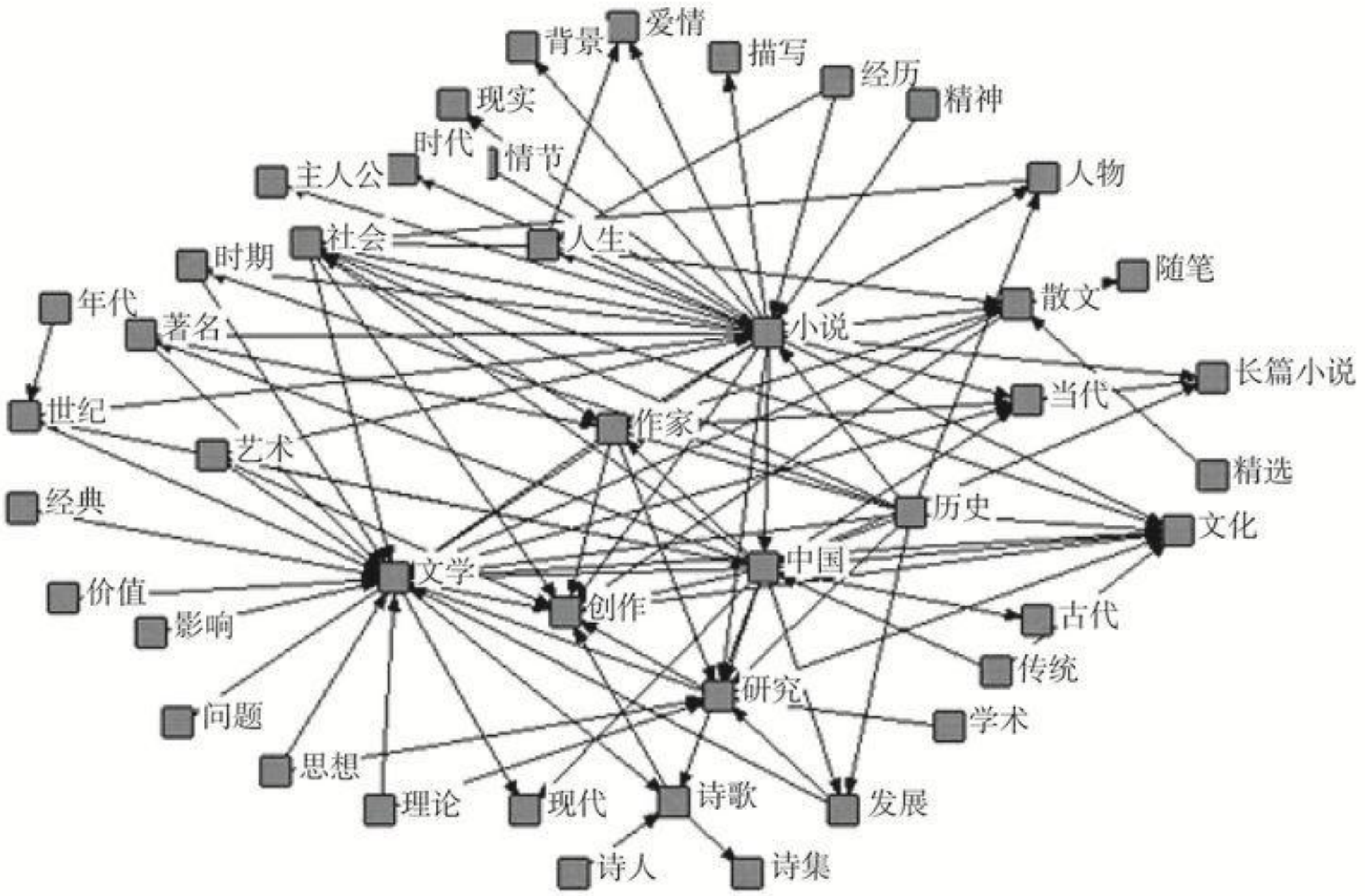
依然可利用有向图的深度优先遍历DFS，通过以下算法获得：

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。

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



背景
爱情
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
散文
年代
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

7.9 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。

谢谢观看