

圖



学习目标

掌握图的定义

理解图的抽象数据类型定义

熟练掌握图的邻接矩阵存储及实现

掌握图的邻接表存储及实现

掌握图的遍历

掌握图的一些常用算法（最小生成树、最短距离、拓扑排序、关键路径）



第六章 图

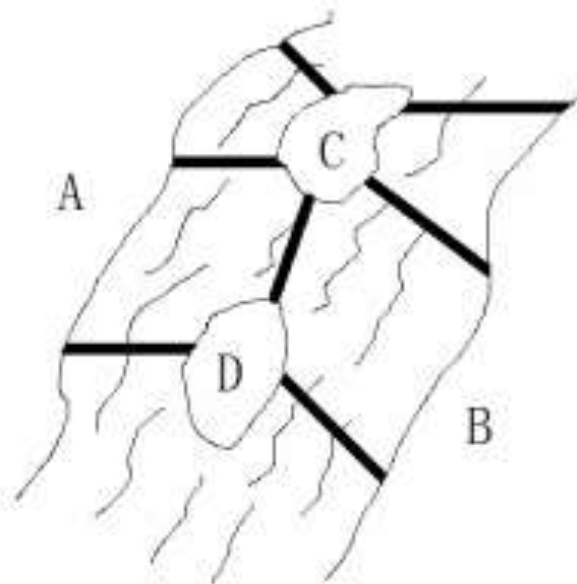
6-1 图的提出



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？

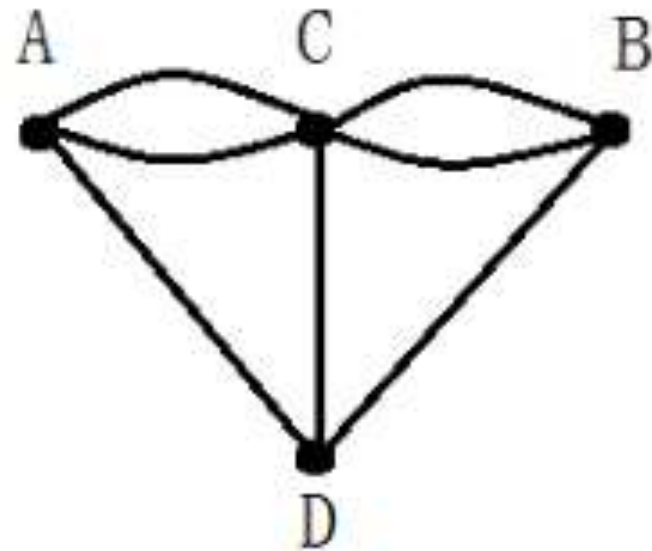




问题求解：哥尼斯堡七桥问题

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）。

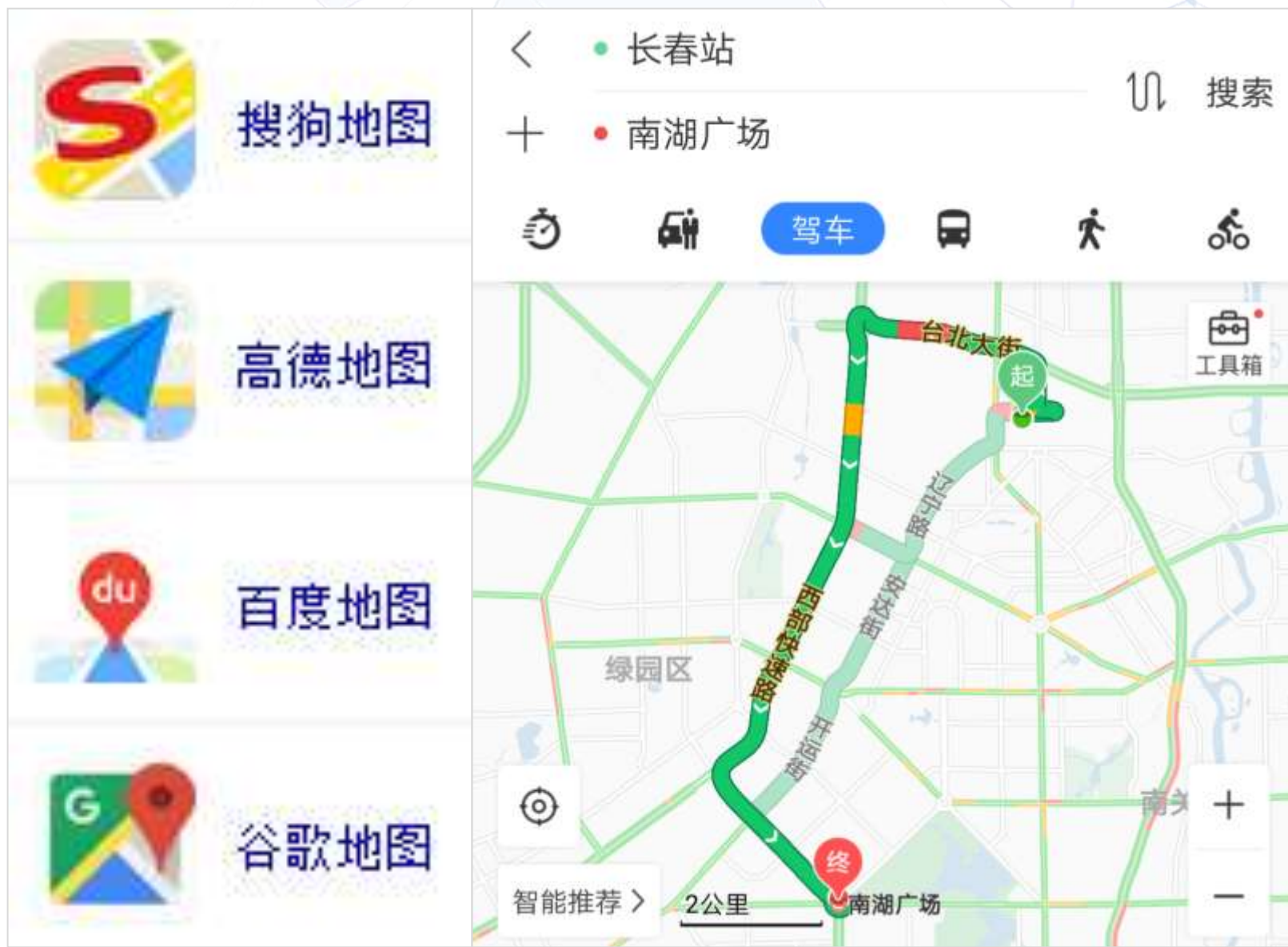


社交网络

【社交网络】即社交网络服务（Social Network Service，SNS），也称为社会性网络服务或社会化网络服务



路径规划



关于图结构



什么是图？在逻辑上有什么特点？有哪些基本术语？



如何存储图结构？



在不同的存储结构上，如何实现图的基本操作？



最小生成树



最短路径



拓扑排序



关键路径

图的应用



第六章 图

6-2 图的逻辑结构

讲什么？



图的定义



图的基本术语



图的抽象数据类型定义



图的遍历——深度优先



图的遍历——广度优先

图的定义

数据元素



图(Graph): 由顶点(vertex)的 (有限) 集合和顶点之间边(edge)的集合组成, 通常表示为:

$$G = (V, E)$$



顶点的集合



顶点之间边的集合

101教材中, V 必须为非空集合, $V \neq \emptyset$ 但 $E = \emptyset$ 的图为零图

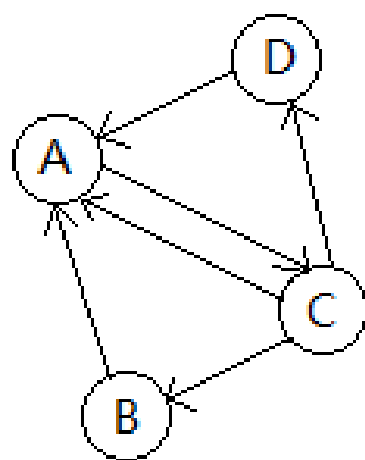


空表、空栈、空队列、空串、空二叉树、空树、零图

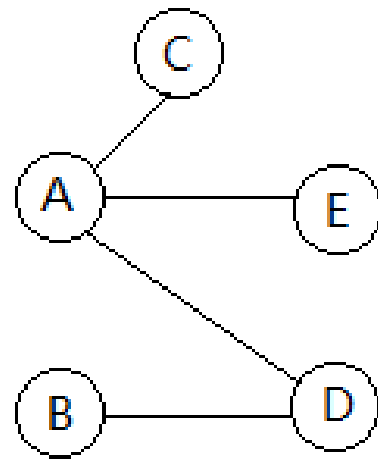
G1的构成

顶点集合 $V = \{A, B, C, D\}$

边的集合 $E = \{ \langle B, A \rangle, \langle A, C \rangle, \langle C, A \rangle, \langle C, D \rangle, \langle D, A \rangle, \langle C, B \rangle \}$



G1



G2

尖括号表示有方向

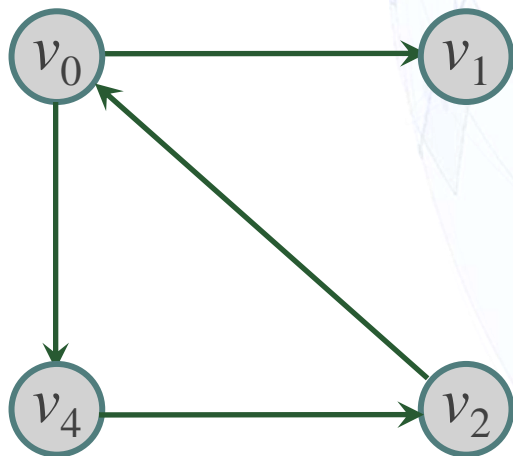
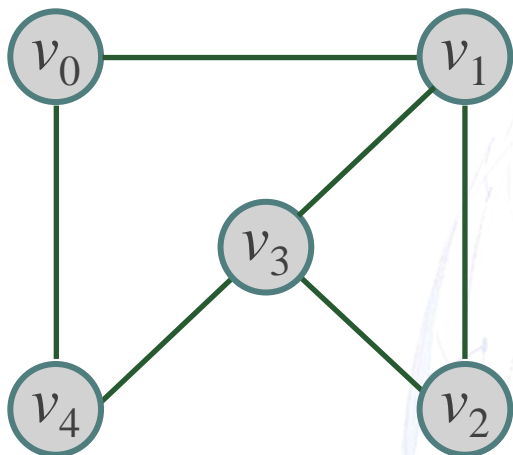
G2的构成

顶点集合 $V = \{A, B, C, D, E\}$

边集合 $E = \{ (A, C), (A, E), (D, B), (D, A) \}$

圆括号表示没有方向

图的分类



📌 无向边：表示为 (v_i, v_j) ，顶点 v_i 和 v_j 之间的边没有方向

📌 无向图：图中任意两个顶点之间的边都是无向边

图（边是否有方向）

无向图 (undirected graph)

有向图 (directed graph)

📌 有向边（弧）：表示为 $\langle v_i, v_j \rangle$ ，从 v_i 到 v_j 的边有方向， v_i 称为弧尾， v_j 称为弧头

📌 有向图：图中任意两个顶点之间的边都是有向边

思考

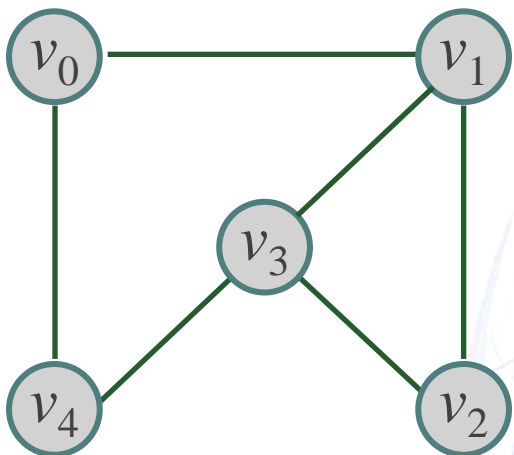


如果图中同时出现无向边和有向边, 如何处理?



将混合图转成有向图

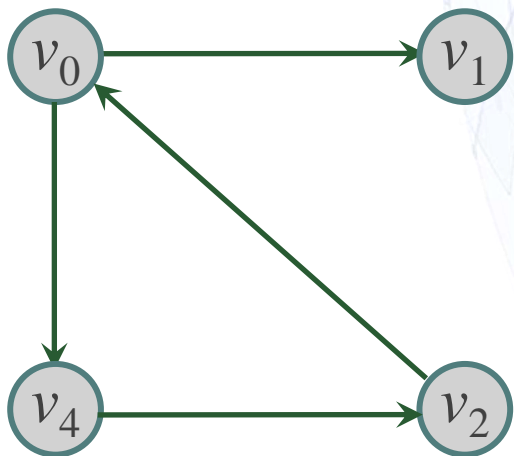
逻辑关系



邻接(adjacent)、依附(adhere): **无向图**中, 对于任意两个顶点 v_i 和顶点 v_j , 若存在边 (v_i, v_j) , 则称顶点 v_i 和顶点 v_j 互为**邻接点**, 同时称边 (v_i, v_j) 依附于顶点 v_i 和顶点 v_j

v_0 的邻接点: v_1, v_4

v_2 的邻接点: v_1, v_3

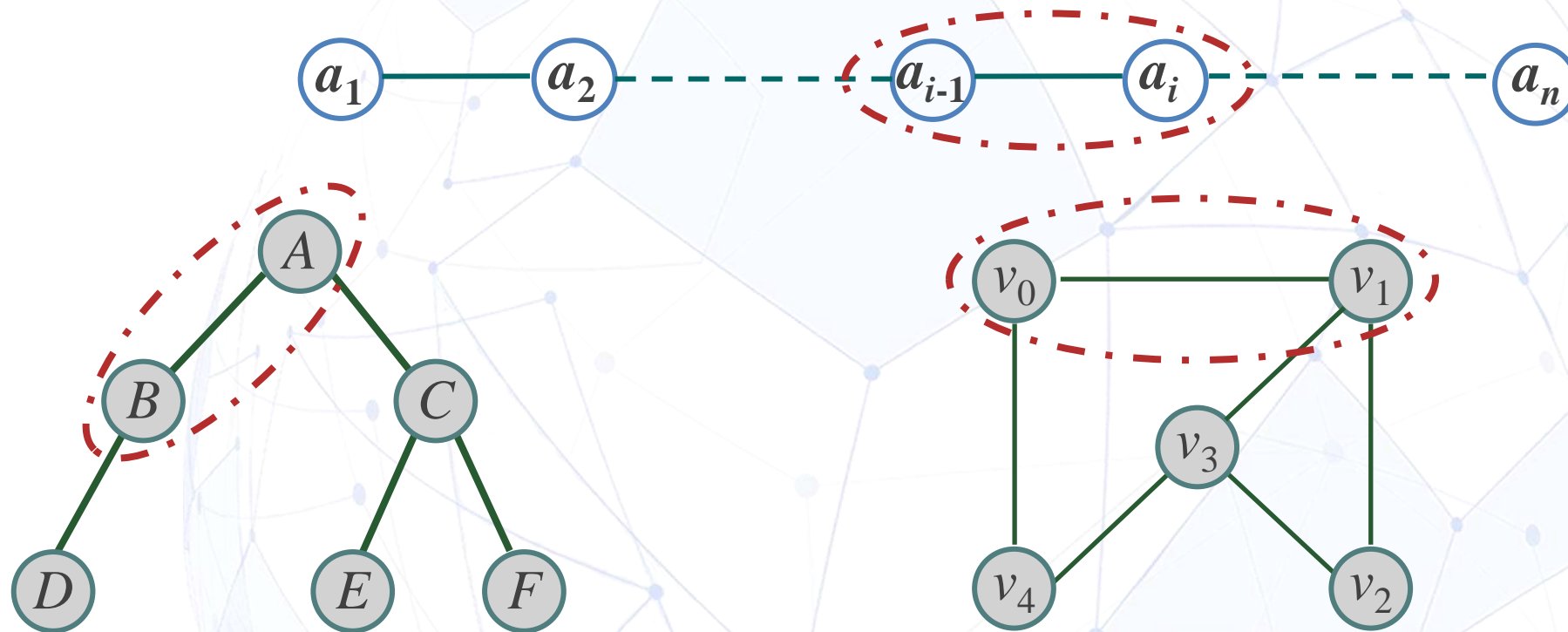


邻接、依附: **有向图**中, 对于任意两个顶点 v_i 和顶点 v_j , 若存在弧 $\langle v_i, v_j \rangle$, 则称顶点 v_i **邻接到** v_j , 顶点 v_j **邻接自** v_i , 同时称弧 $\langle v_i, v_j \rangle$ 依附于顶点 v_i 和顶点 v_j

v_0 的邻接点: v_1, v_4

v_2 的邻接点: v_0

逻辑关系



线性结构中，数据元素之间具有线性关系，逻辑关系表现为前驱-后继；

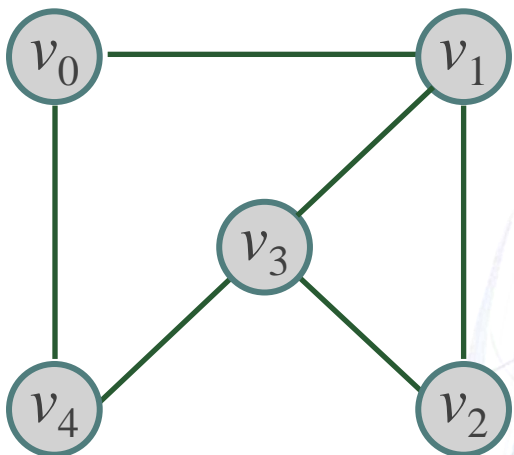


树结构中，结点之间具有层次关系，逻辑关系表现为双亲-孩子



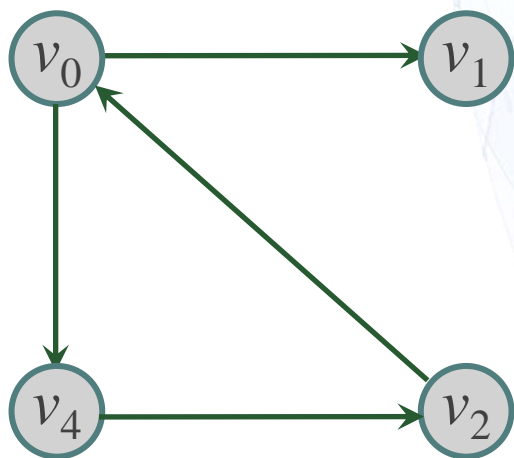
图结构中，任意两个顶点之间都可能有关系，逻辑关系表现为邻接

度、入度和出度



顶点的度(degree): 在**无向图**中, 顶点 v 的度是指依附于该顶点的边数, 通常记为 $TD(v)$

$$TD(v_0) = 2, \quad TD(v_3) = 3$$



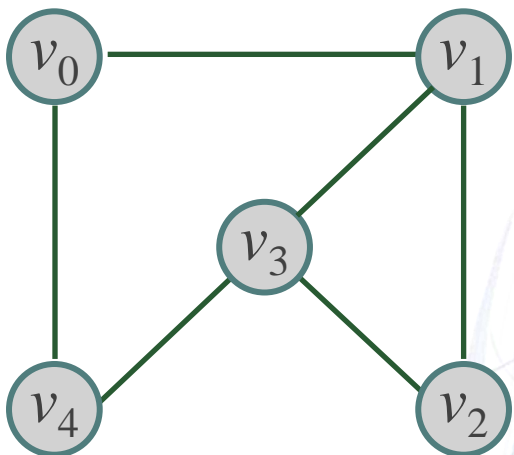
顶点的入度(in-degree): 在**有向图**中, 顶点 v 的入度是指以该顶点终止的弧的数目, 记为 $ID(v)$;



顶点的出度(out-degree): 在**有向图**中, 顶点 v 的出度是指以该顶点出发的弧的数目, 记为 $OD(v)$;

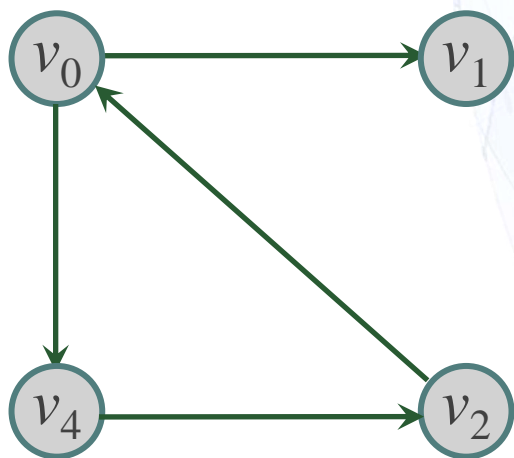
$$ID(v_0) = 1, \quad OD(v_0) = 2,$$

度、入度和出度



在具有 n 个顶点、 e 条边的**无向图**中，各顶点的度之和与边数之和有什么关系？

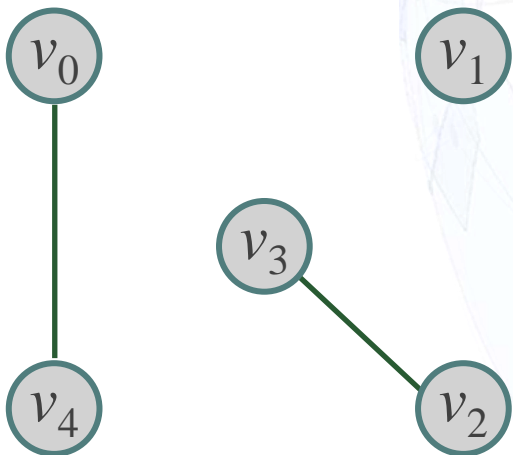
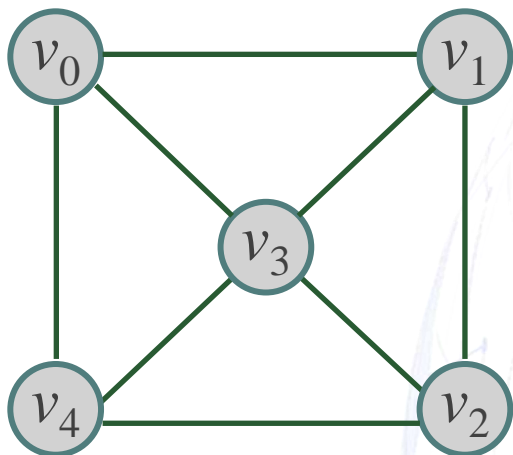
$$\sum_{i=0}^{n-1} TD(v_i) = 2e$$



在具有 n 个顶点、 e 条边的**有向图**中，各顶点的入度之和与各顶点的出度之和有什么关系？与边数之和有什么关系？

$$\sum_{i=0}^{n-1} ID(v_i) = \sum_{i=0}^{n-1} OD(v_i) = e$$

图的分类



稠密图：边数很多的图

图（边数的多寡）

稠密图(dense graph)

稀疏图(sparse graph)



稀疏图：边数很少的图

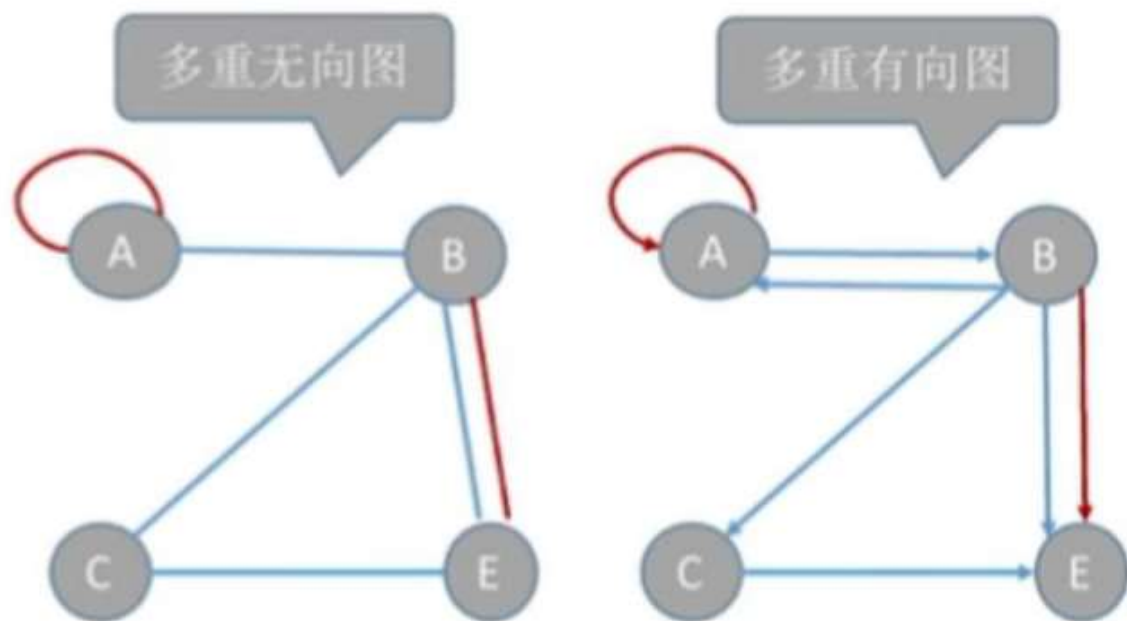
图的分类



简单图：若一个图中不包含同一条边的多个副本，也不包含自连边即 (v_i, v_i) 或 $\langle v_i, v_i \rangle$ ，则称这样的图是简单图

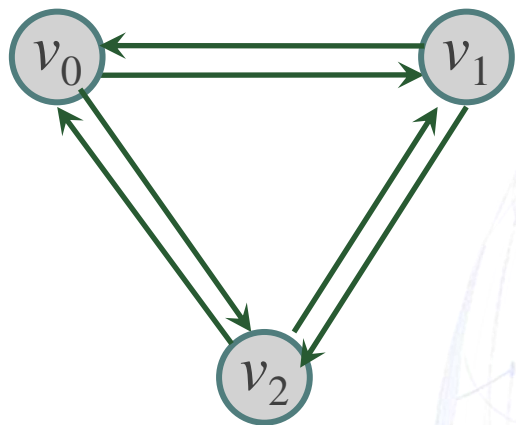


多重图：不符合简单图定义的图称为多重图



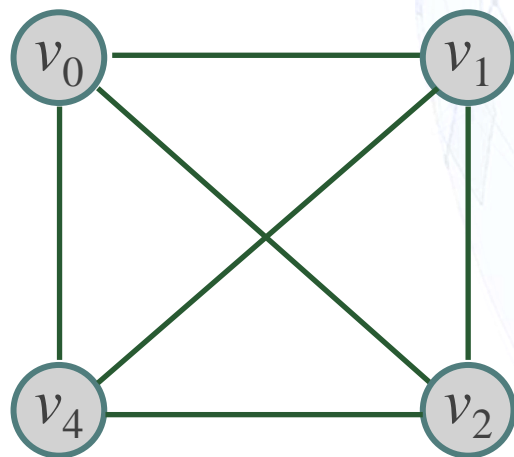
注意：数据结构只讨论简单图

完全图 (complete graph)



📌 有向完全图：有向图中，任意两个顶点之间都存在方向相反的两条弧

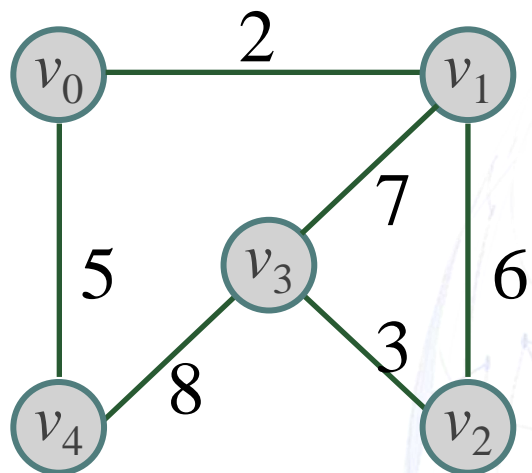
🕒 n 个顶点的有向完全图有多少条弧？ $\Rightarrow n \times (n-1)$



📌 无向完全图：无向图中，任意两个顶点之间都存在边

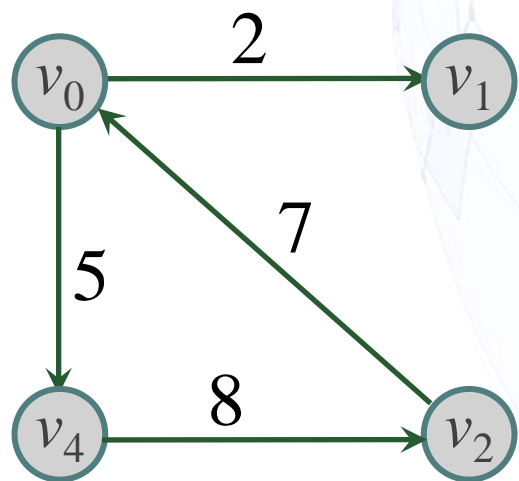
🕒 n 个顶点的无向完全图有多少条边？ $\Rightarrow n \times (n-1)/2$

图的分类



📌 权：对边赋予的有意义的数值量

📌 带权图（**网图 network graph**）：
边上带权的图（包括无向和有向图）



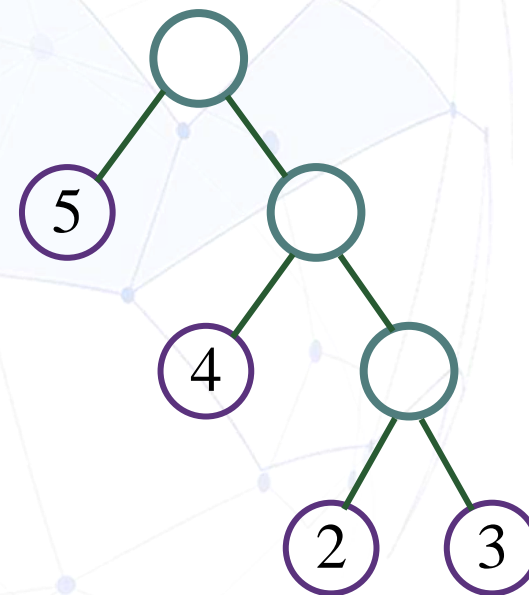
📌 树结构中，权通常赋予在结点

图（边上是否带权）

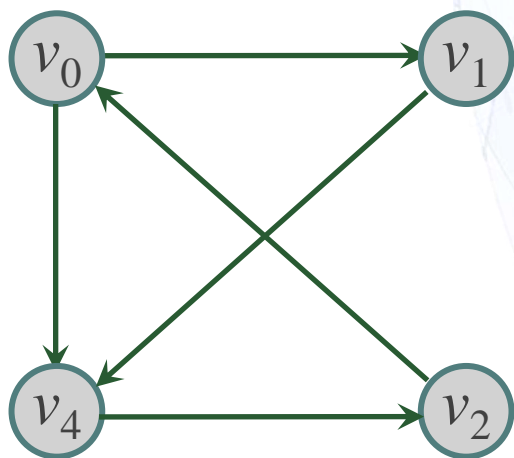
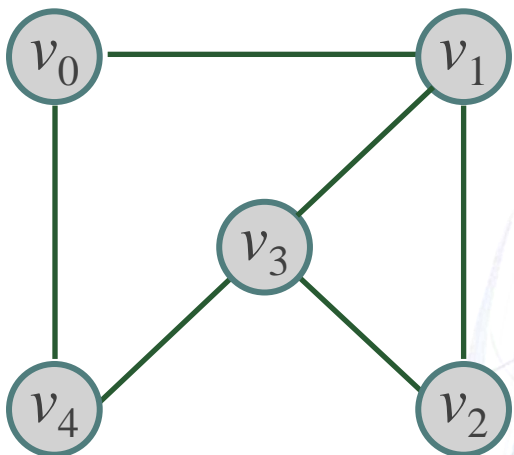
非带权图

带权图

(weighted graph)



路径、回路



路径(path): 在**无向图**中, 顶点 v_p 和顶点 v_q **之间**的路径是一个**顶点序列** $(v_p=v_{i0}, v_{i1}, \dots, v_{im}=v_q)$, 其中 $(v_{ij-1}, v_{ij}) \in E (1 \leq j \leq m)$

顶点 v_0 和顶点 v_4 之间的路径:

$v_0 v_4$ 、 $v_0 v_1 v_3 v_4$ 、 $v_0 v_1 v_2 v_3 v_4$ 、



回路 (环,circuit) : 第一个顶点和最后一个顶点相同的路径

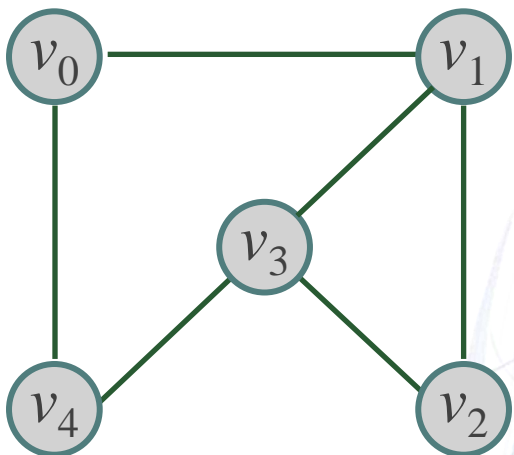


路径: 在**有向图**中, **从**顶点 v_p **到**顶点 v_q 的路径是一个**顶点序列** $(v_p=v_{i0}, v_{i1}, \dots, v_{im}=v_q)$, 其中 $\langle v_{ij-1}, v_{ij} \rangle \in E (1 \leq j \leq m)$

从顶点 v_0 到顶点 v_4 的路径:

$v_0 v_4$ 、 $v_0 v_1 v_4$

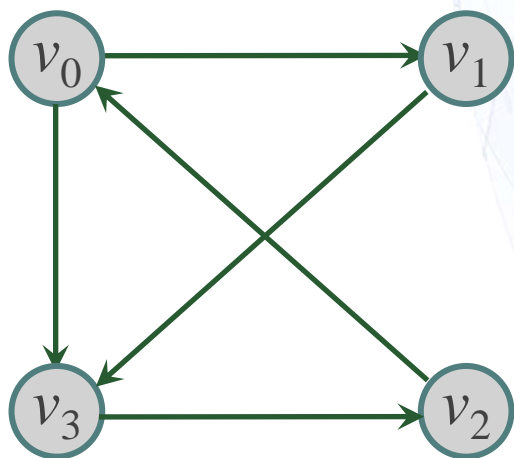
路径、回路



简单路径：序列中顶点不重复出现的路径

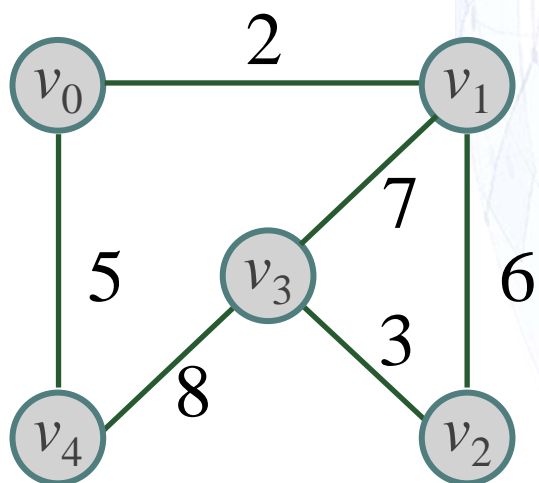
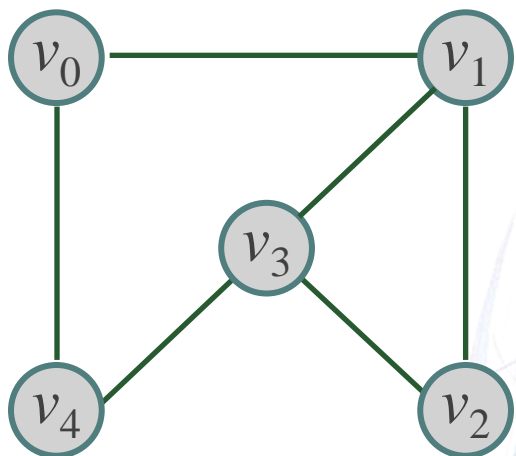


简单回路（简单环）：除了第一个顶点和最后一个顶点外，其余顶点不重复出现的回路。



不致混淆的情况下，路径和回路都是简单的

路径、回路



路径长度：非带权图——路径上边的个数

路径长度：带权图——路径上边的权值之和

顶点 v_0 和顶点 v_4 之间的路径长度：

$v_0 v_4$: 1

$v_0 v_4$: 5

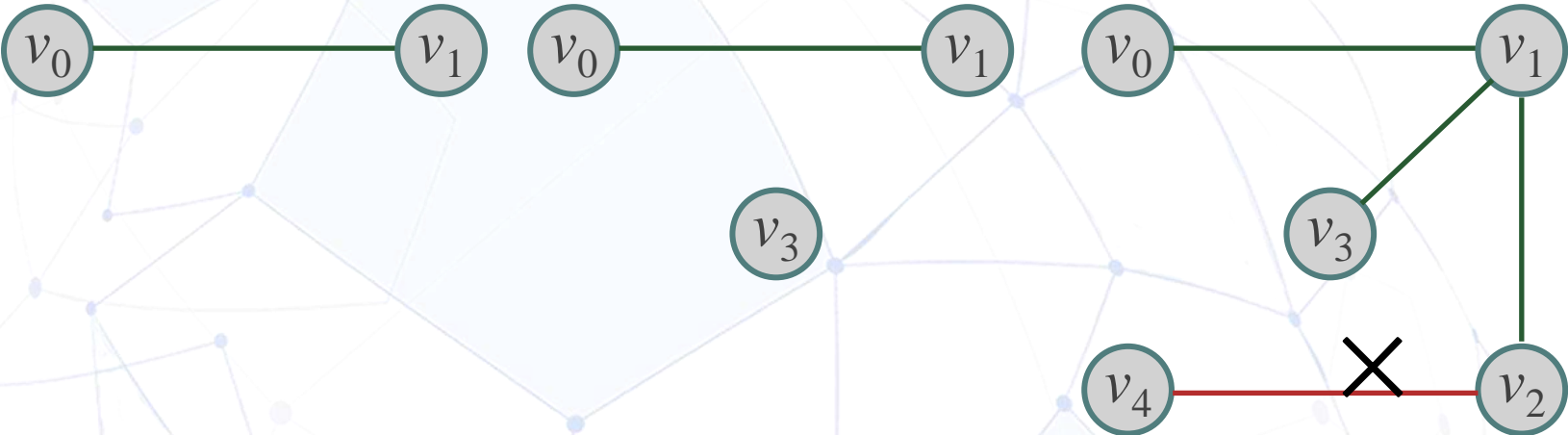
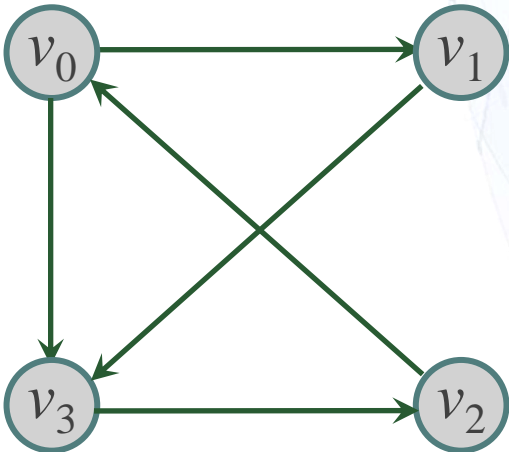
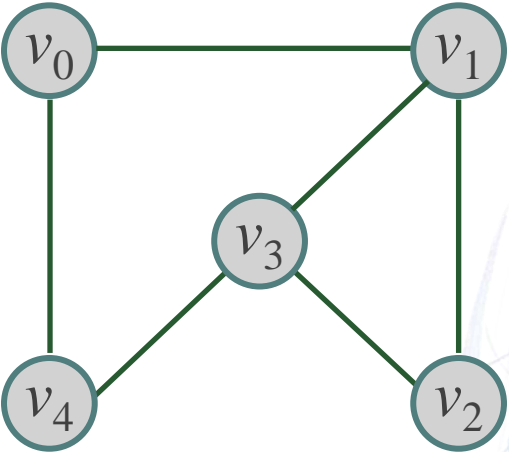
$v_0 v_1 v_3 v_4$: 3

$v_0 v_1 v_3 v_4$: 17

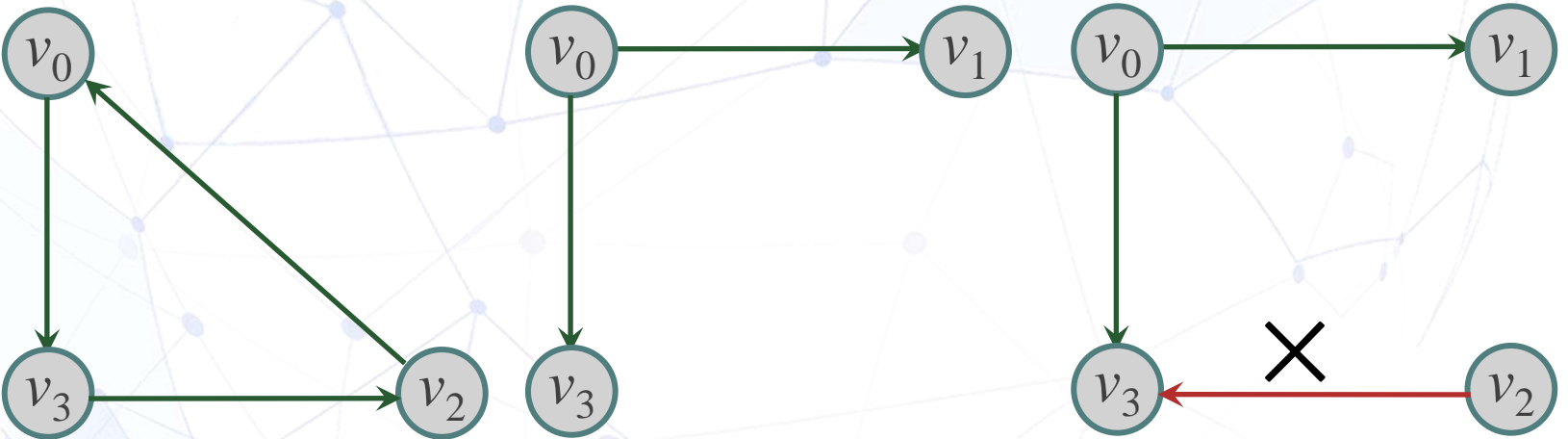
$v_0 v_1 v_2 v_3 v_4$: 4

$v_0 v_1 v_2 v_3 v_4$: 19

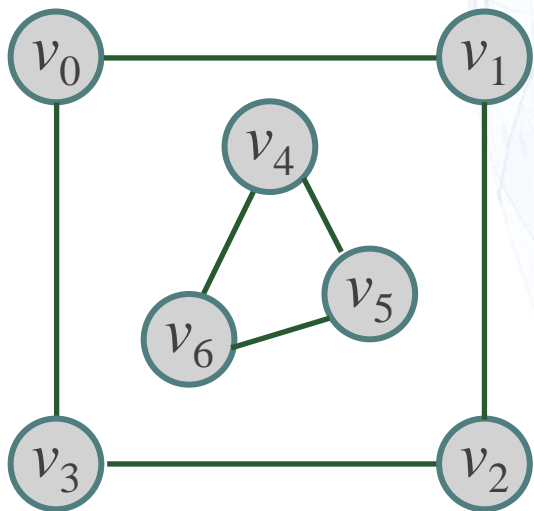
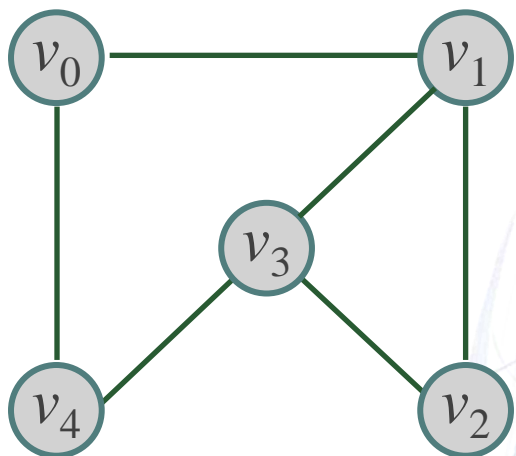
子图



子图：若图 $G=(V, E)$, $G'=(V', E')$, 如果 $V' \subseteq V$ 且 $E' \subseteq E$, 则称图 G' 是 G 的子图



连通图



连通顶点: 在**无向图**中, 如果顶点 v_i 和顶点 $v_j (i \neq j)$ 之间有**路径**, 则称顶点 v_i 和 v_j 是连通的



连通图: 在**无向图**中, 如果**任意**两个顶点都是连通的, 则称该无向图是连通图



对于非连通图, 实际处理中会有什么问题?

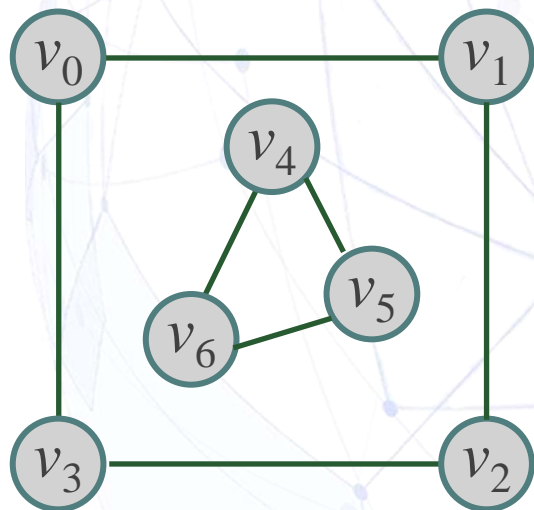


从某顶点出发进行**遍历**, 有一些顶点访问不到

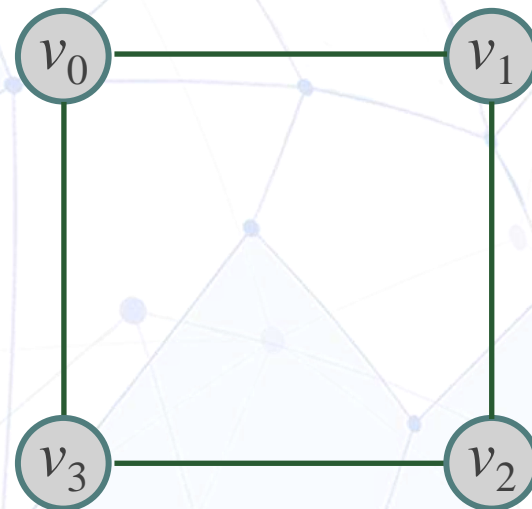
连通分量

📌 连通分量：非连通图的极大连通子图

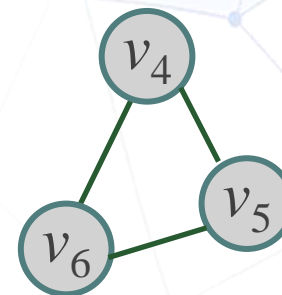
含有最多的顶点数
依附于这些顶点的所有边



连通分量1



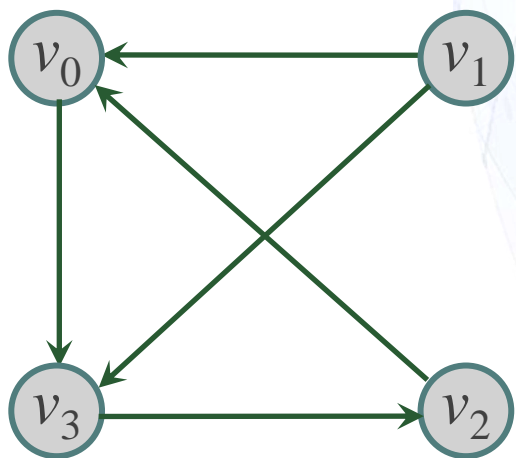
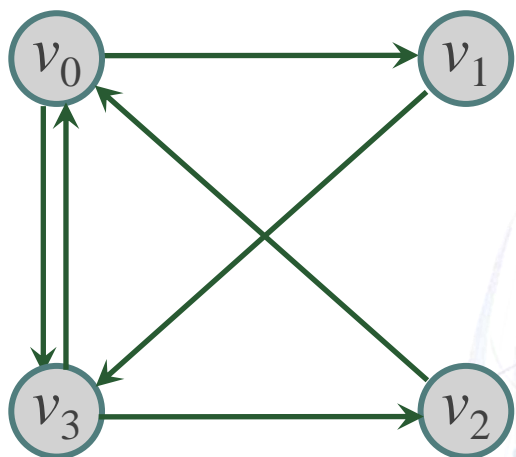
连通分量2



连通分量是对**无向图**的一种划分

任何连通图的连通分量只有一个，即是其自身，非连通的无向图有多个连通分量。

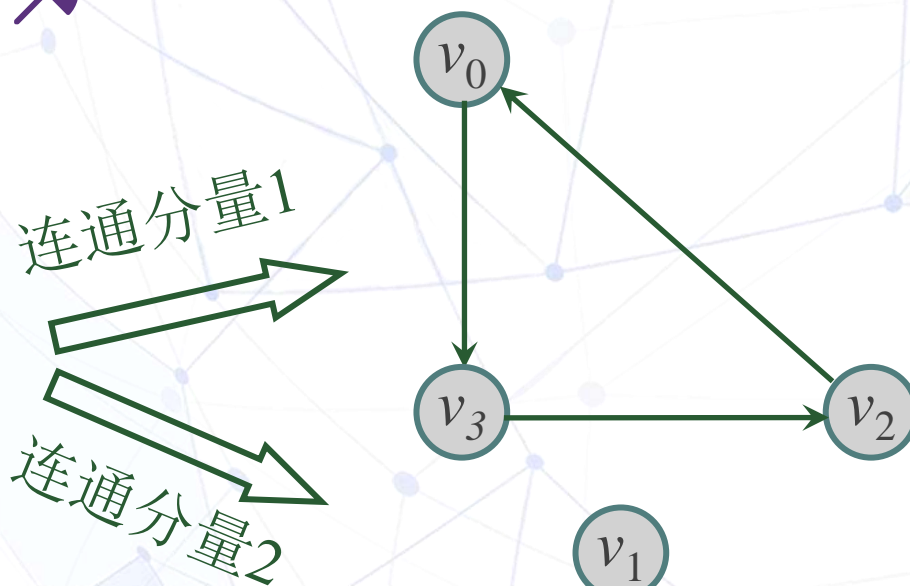
强连通图、强连通分量



📌 **强连通顶点**：在**有向图**中，如果从顶点 v_i 到顶点 v_j 和从顶点 v_j 到顶点 v_i 均有路径，则称顶点 v_i 和 v_j 是强连通的

📌 **强连通图**：在**有向图**中，如果任意两个顶点都是强连通的，则称该有向图是强连通图

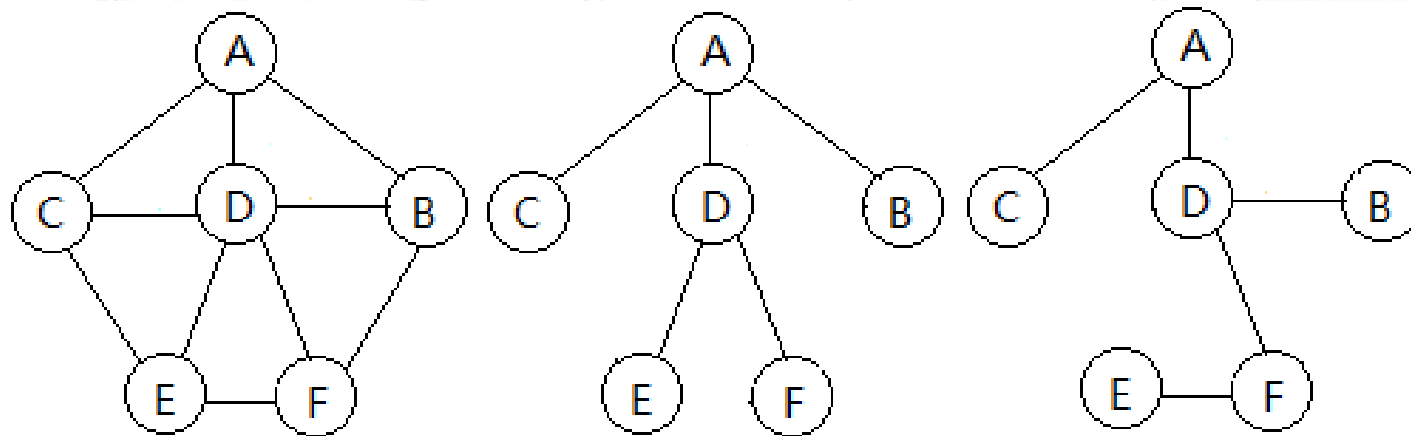
📌 **强连通分量**：非强连通图的**极大连通子图**



注意：无向图的任意一个顶点或一条边必定会出现在某个连通分量中；有向图中，任意某个顶点会出现在某个强连通分量中，但一些边可能不在任何强连通分量中出现

生成树

- **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) G_7

(b) G_7 的两个生成树

1. 带权图指的是（ ）。

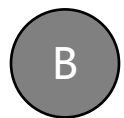
- ☐ A 顶点带权的无向图或有向图
- ☒ B 边上带权的无向图或有向图
- ☐ C 有回路的无向图或有向图
- ☐ D 无回路的无向图或有向图

提交

2. 在图结构中，逻辑关系表现为邻接，相互邻接的顶点之间具有逻辑关系。



正确



错误

提交

3. 最稀疏的图是（ ）。

- ☐ A 空图
- ☒ B 零图
- ☐ C 完全图
- ☐ D 满图

提交

4. 最稠密的图是（ ）。

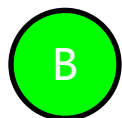
- ☐ A 空图
- ☐ B 零图
- ☒ C 完全图
- ☐ D 满图

提交

5. 在无向图中，路径可能不唯一，在有向图中，路径是唯一的。



正确



错误

提交

7. 若无向图 $G=(V, E)$ 有两个连通分量 $G_1=(V_1, E_1)$ 和 $G_2=(V_2, E_2)$, 则有 ()。

- ☒ A $|V_1|+|V_2|=|V|, |E_1|+|E_2|=|E|$
- ☐ B $|V_1|+|V_2|<|V|, |E_1|+|E_2|<|E|$
- ☐ C $|V_1|+|V_2|=|V|, |E_1|+|E_2|<|E|$
- ☐ D $|V_1|+|V_2|<|V|, |E_1|+|E_2|=|E|$

提交

8. 若有向图 $G=(V, E)$ 有两个连通分量 $G_1=(V_1, E_1)$ 和 $G_2=(V_2, E_2)$, 则有 ()。

- ☐ A $|V_1|+|V_2|=|V|, |E_1|+|E_2|=|E|$
- ☐ B $|V_1|+|V_2|<=|V|, |E_1|+|E_2|<=|E|$
- ☒ C $|V_1|+|V_2|=|V|, |E_1|+|E_2|<=|E|$
- ☐ D $|V_1|+|V_2|<=|V|, |E_1|+|E_2|=|E|$

提交

抽象数据类型定义

图是一种与具体应用密切相关的数据结构，基本操作有很大差别

ADT Graph

DataModel

顶点的有穷非空集合和顶点之间边的集合

Operation

CreatGraph: 图的建立

DestroyGraph: 图的销毁

DFTraverse: 深度优先遍历图

BFTraverse: 广度优先遍历图

.....

endADT

简单起见，先只讨论图的遍历

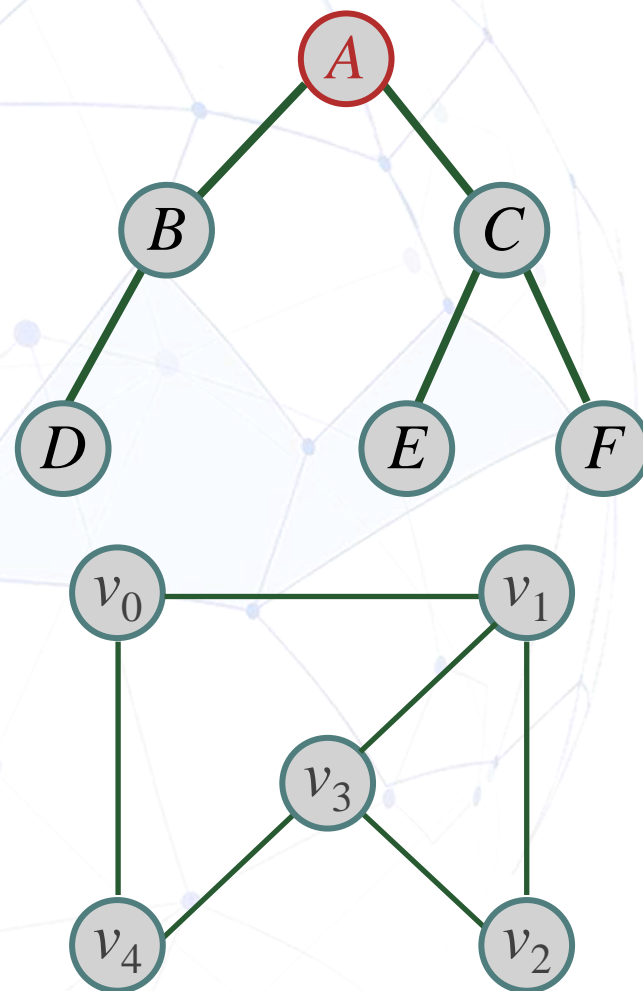
图的遍历

✚ 图的遍历：从图中**某**一顶点出发**访问**图中所有顶，并且每个结点仅被访问一次



🕒 在图中，如何选取遍历的起始顶点？

📄 解决方案：将图中的顶点按**任意**顺序排列起来，从编号最小的顶点开始



图的遍历

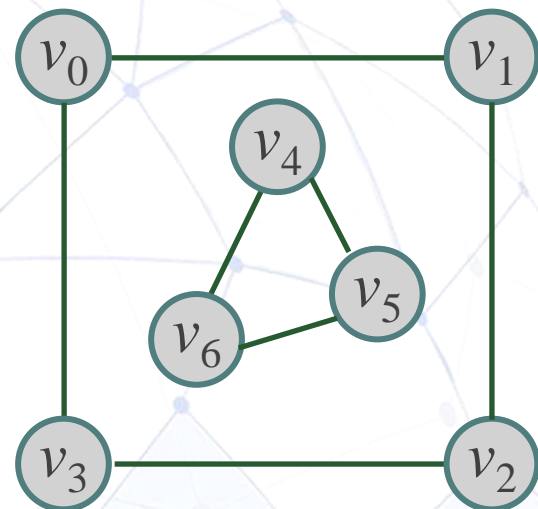
🕒 从某顶点出发能访问其他所有顶点吗？

🗑️ 解决方案：多次调用图遍历算法

下面仅讨论从某顶点出发遍历图的算法

🕒 如何避免遍历不会因回路而陷入死循环？

🗑️ 解决方案：附设访问标志数组visited[n]



图的遍历



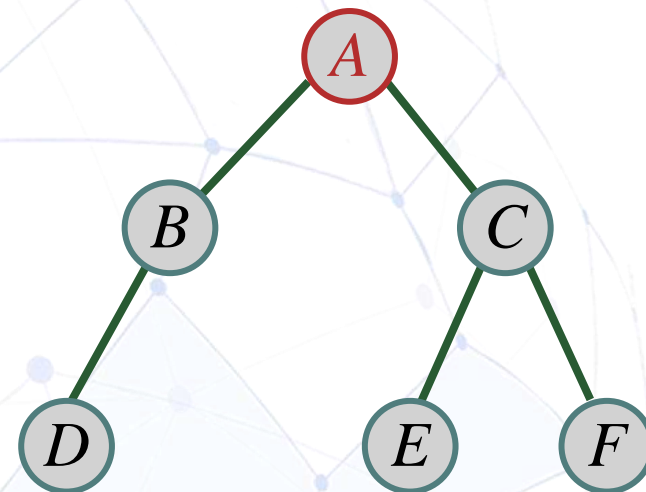
线性序: $a_1 a_2 \dots a_n$

前序: $A B D C E F$

中序: $D B A E C F$

后序: $D B E F C A$

层序: $A B C D E F$



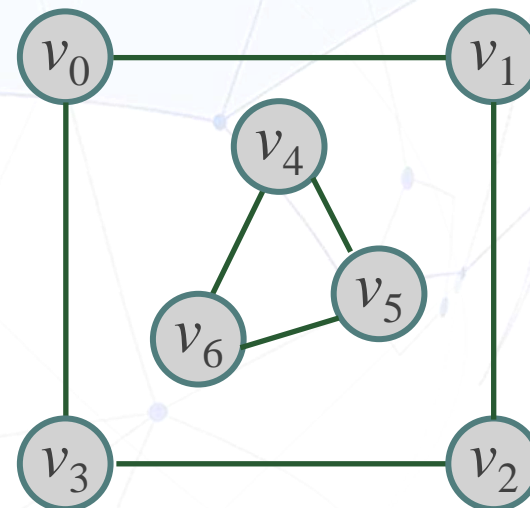
采用什么次序依次访问图中所有顶点?

解决方案:



深度优先遍历(Depth_first Search-DFS)

广度优先遍历(Breadth_first Search-BFS)



DFS基本思想

 从顶点 v 出发进行深度优先遍历过程：

- (1) 访问顶点 v ;
- (2) 对 v 的所有未被访问的邻接点 w ，从 w 出发进行深度优先遍历。

深度优先遍历是一个递归的过程

类似于树的先序遍历，是一种递归定义的遍历。

伪代码描述

 用伪代码描述深度优先遍历的操作定义

算法: DFSTraverse

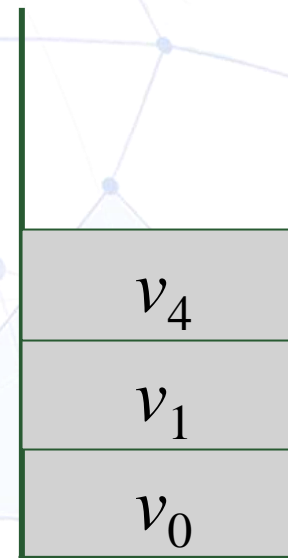
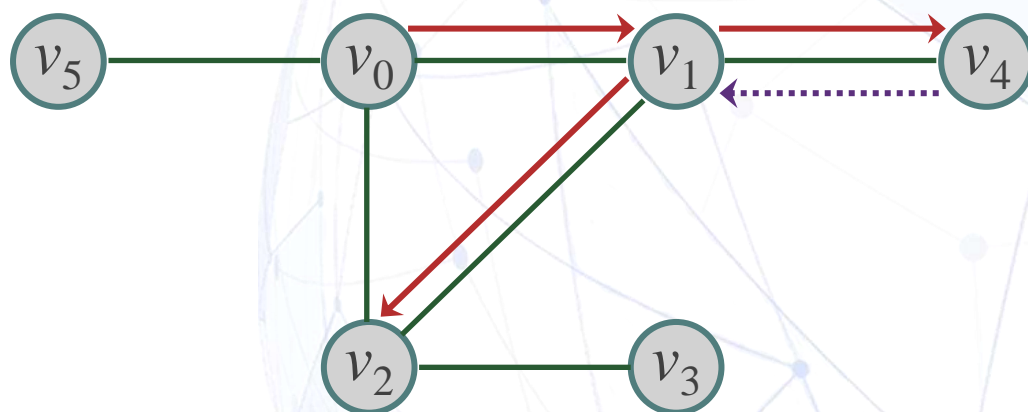
输入: 顶点的编号 v

输出: 无

1. 访问顶点 v ; 修改标志 $visited[v] = 1$;
2. $w =$ 顶点 v 的第一个邻接点;
3. while (w 存在)
 - 3.1 if (w 未被访问) 从顶点 w 出发递归执行该算法;
 - 3.2 $w =$ 顶点 v 的下一个邻接点;

运行实例

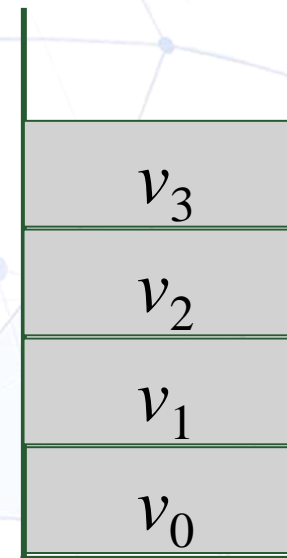
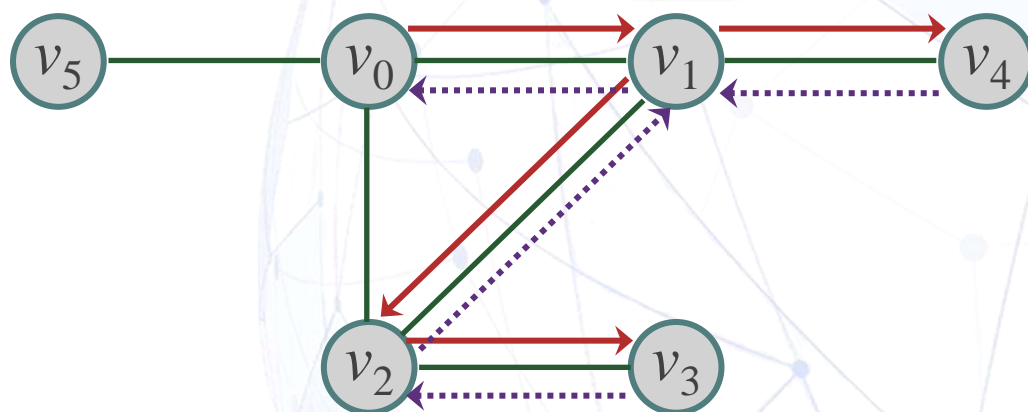
 运行实例——深入理解操作过程



深度优先遍历序列: v_0 v_1 v_4

运行实例

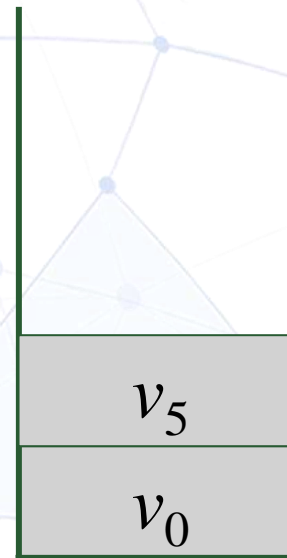
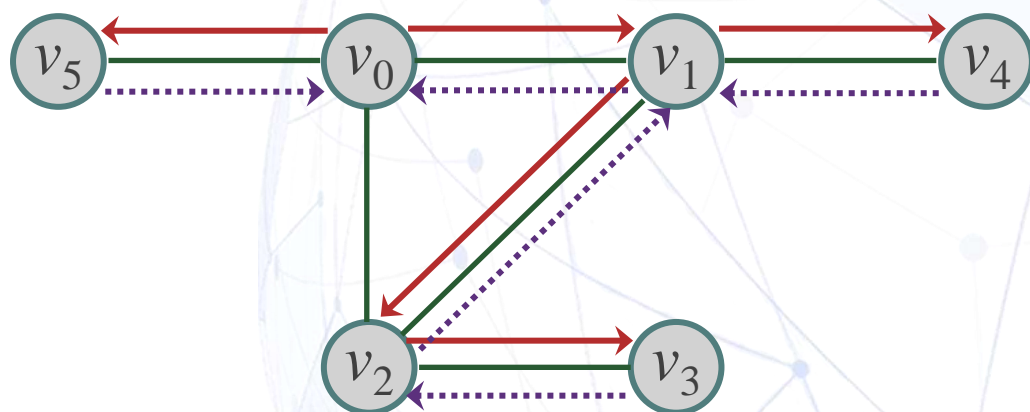
 运行实例——深入理解操作过程



深度优先遍历序列: v_0 v_1 v_4 v_2 v_3

运行实例

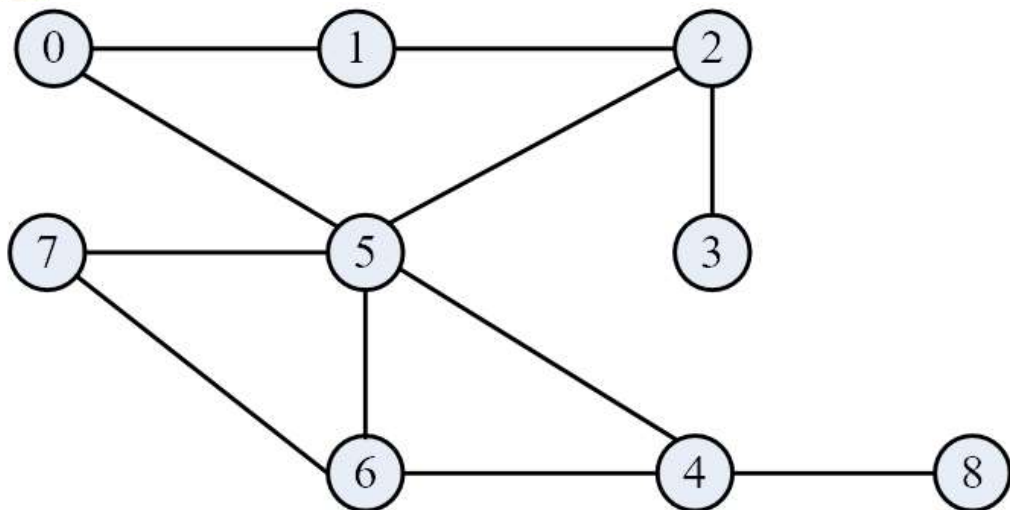
✍ 运行实例——深入理解操作过程



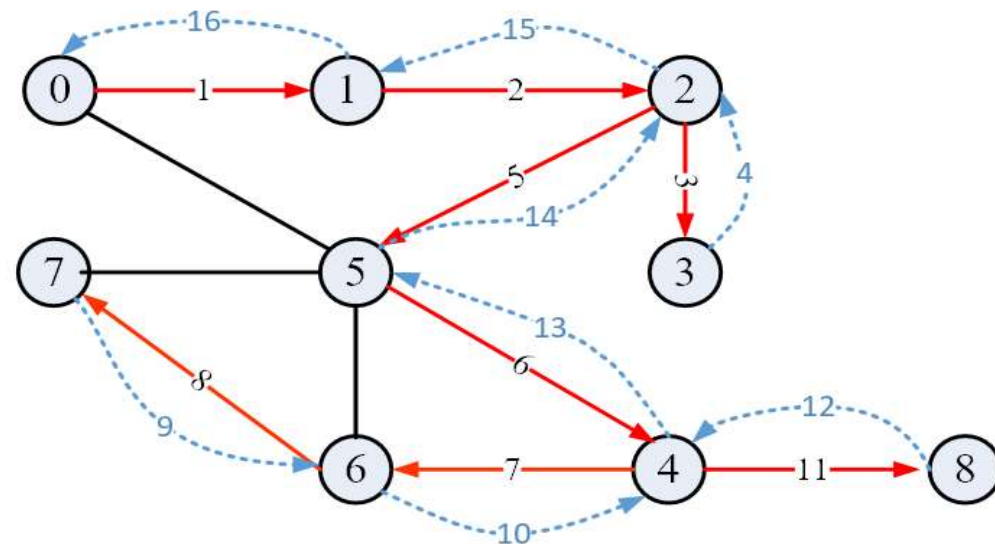
深度优先遍历序列: $v_0 \ v_1 \ v_4 \ v_2 \ v_3 \ v_5$



无向连通图的深度优先搜索（DFS）

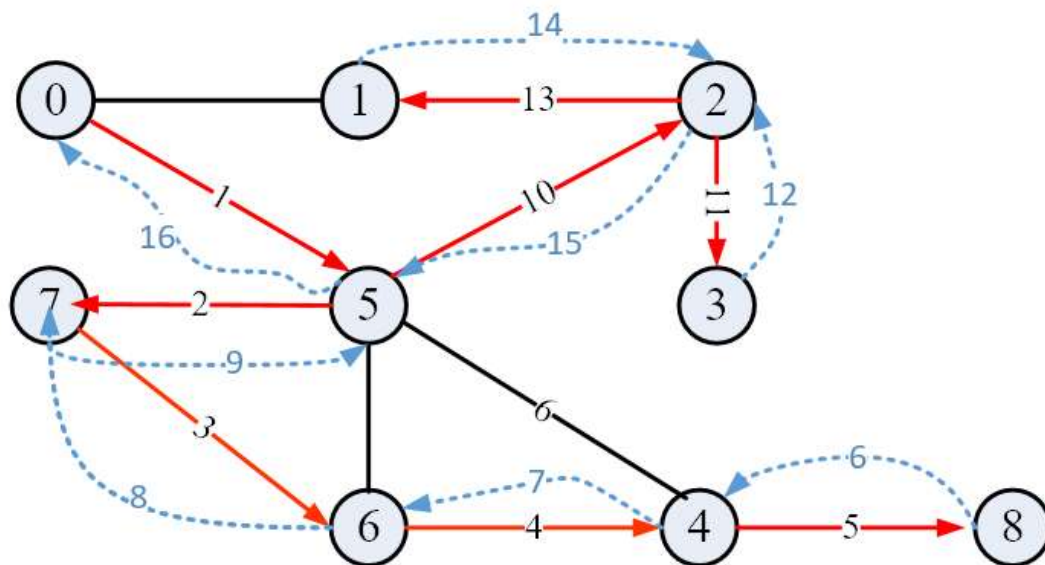


优先选编号小的邻接点



0 1 2 3 5 4 6 7 8

优先选编号大的邻接点

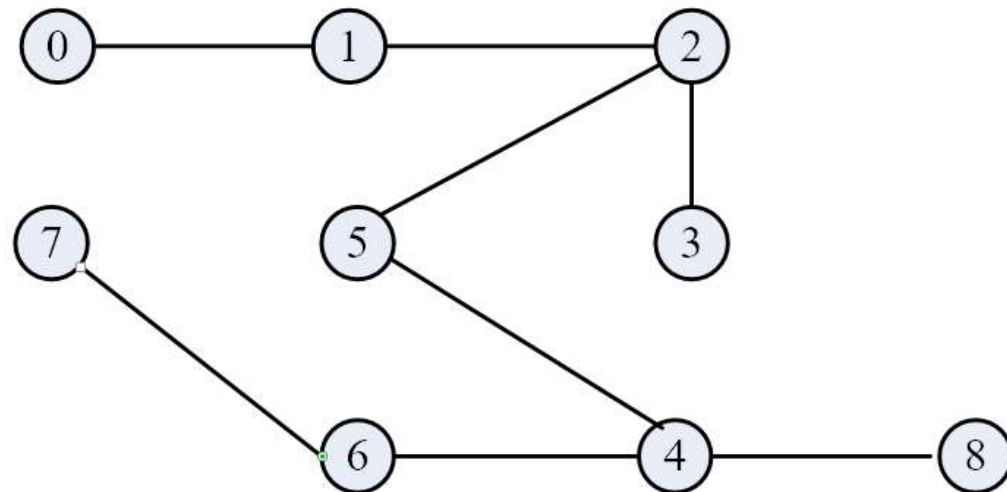
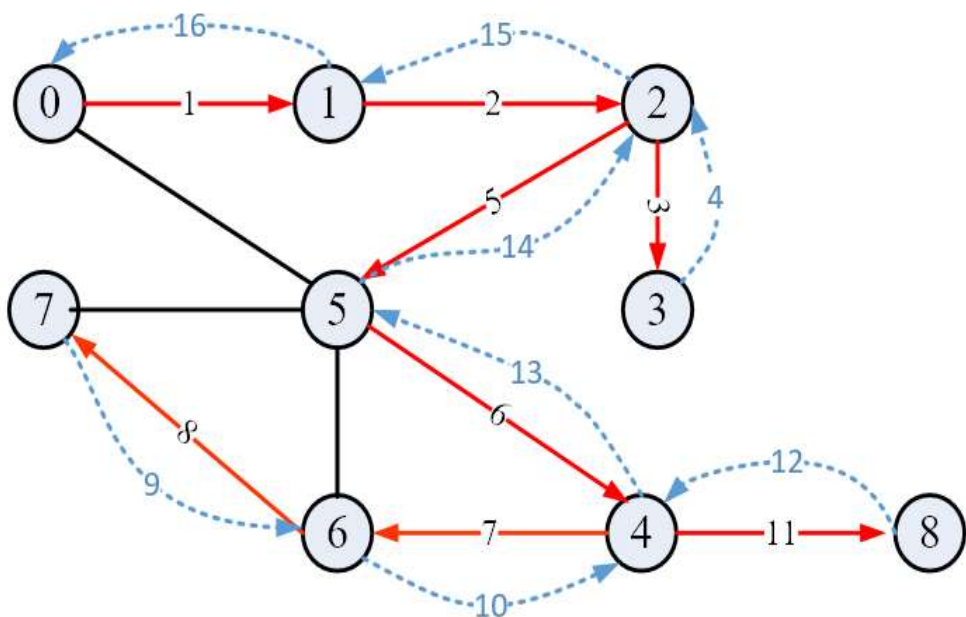


0 5 7 6 4 8 2 3 1



深度优先搜索生成树

- 搜索路径对应的生成树

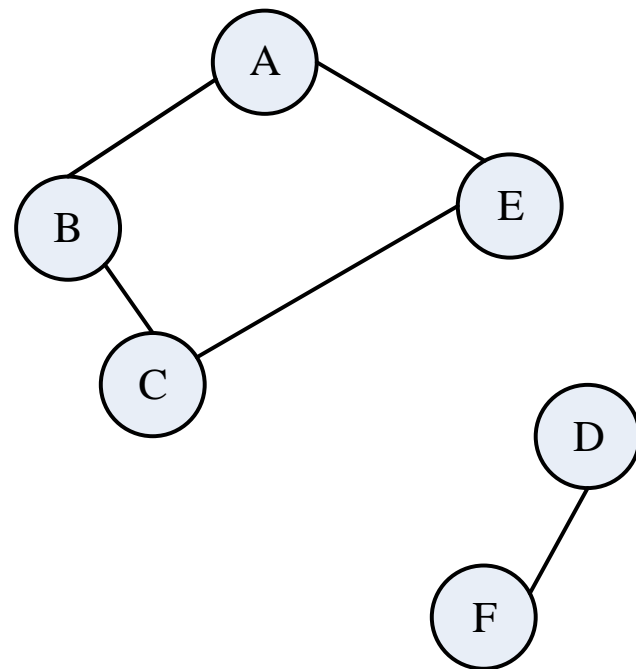




非连通图的遍历

- 对于不连通的图或有向图，只要依次检查图中每一个顶点 v_i ，如果顶点 v_i 未被访问过，则从它出发进行DFS(v_i)遍历即可。
- 如右图非连通图，依次检查A-F的每个顶点是否已被访问过；
- 首先从A出发，可访问到BCE；
- 依次检查顶点B和C，无需从它们出发遍历；
- 接下来从D出发，可访问到DF；
- 无需从E和F遍历，遍历结束。
- 整个遍历序列为ABCEDF。

优先选择序号小的邻接点

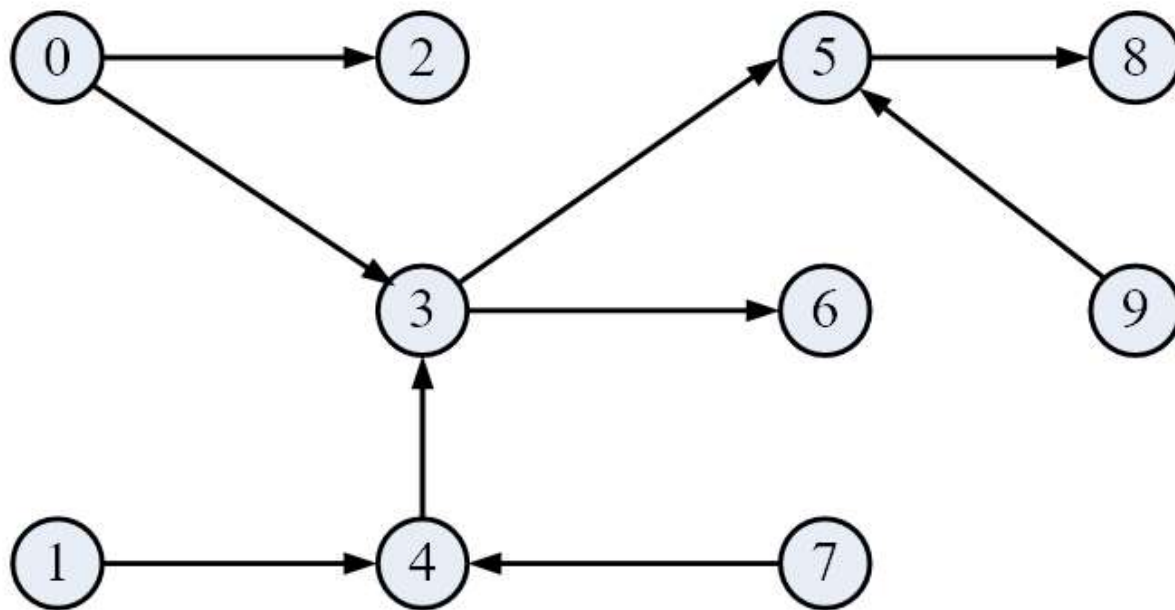


A B C D E F



有向图的遍历

- 遍历序列为:



邻接点编号小的优先

0235861479

BFS基本思想

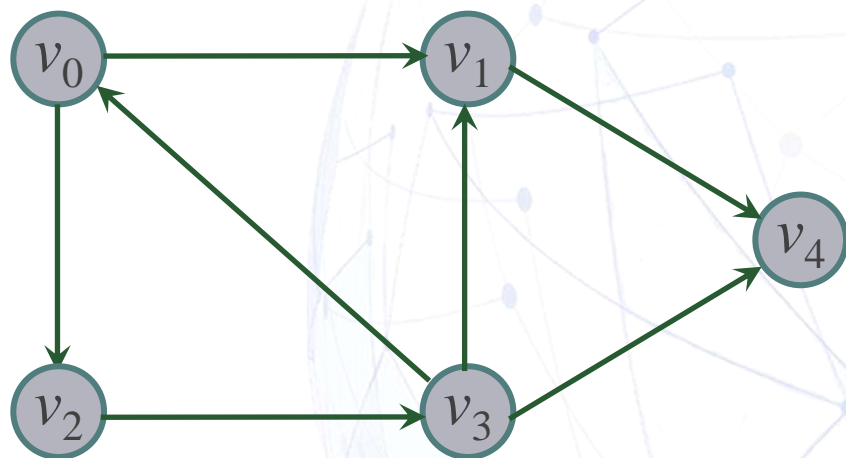
 从顶点 v 出发进行**广度优先遍历**的基本思想是：

- (1) 访问顶点 v ;
- (2) 依次访问 v 的**各个未被访问的邻接点** v_1, v_2, \dots, v_k ;
- (3) 分别从 v_1, v_2, \dots, v_k 出发依次访问它们未被访问的邻接点, 直至访问所有与顶点 v 有路径相通的顶点。

“先被访问顶点的邻接点” **先**于 “后被访问顶点的邻接点”

图的广度优先遍历

 运行实例——深入理解操作过程

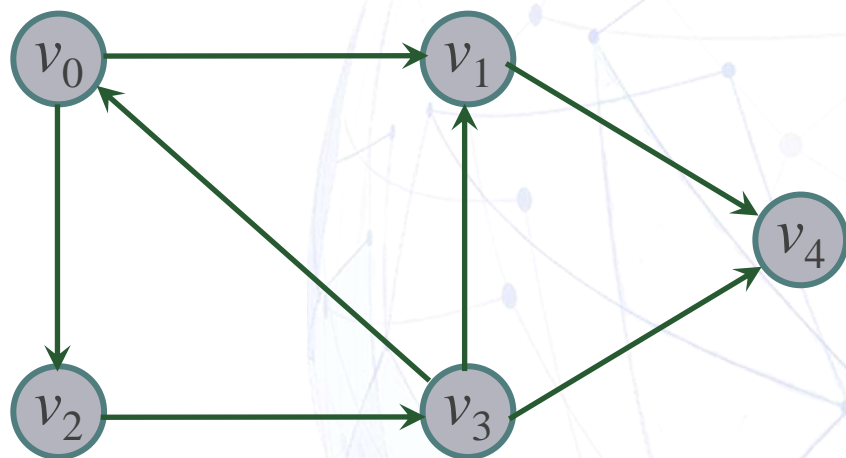


v_0 v_1 v_2

广度优先遍历序列: v_0 v_1 v_2

图的广度优先遍历

 运行实例——深入理解操作过程

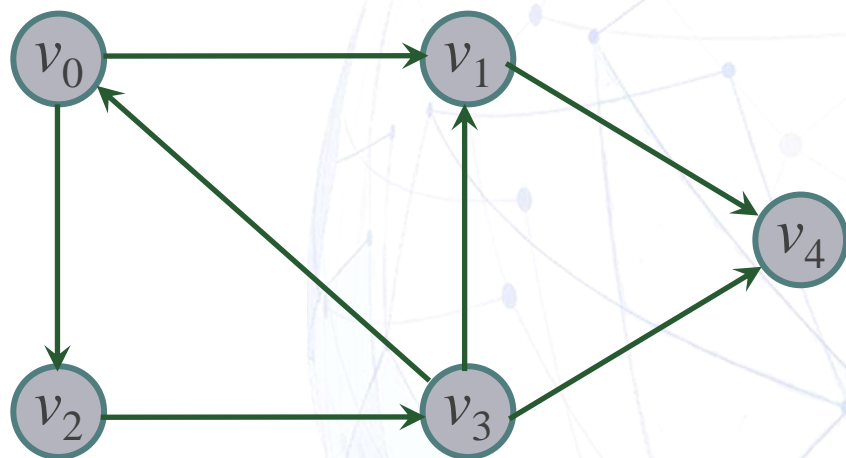


$v_1 \ v_2 \ v_4 \ v_3$

广度优先遍历序列: $v_0 \ v_1 \ v_2 \ v_4 \ v_3$

图的广度优先遍历

 运行实例——深入理解操作过程



v_4 v_3

广度优先遍历序列: v_0 v_1 v_2 v_4 v_3

伪代码描述

用伪代码描述广度优先遍历的操作定义

算法: BFS_Traverse

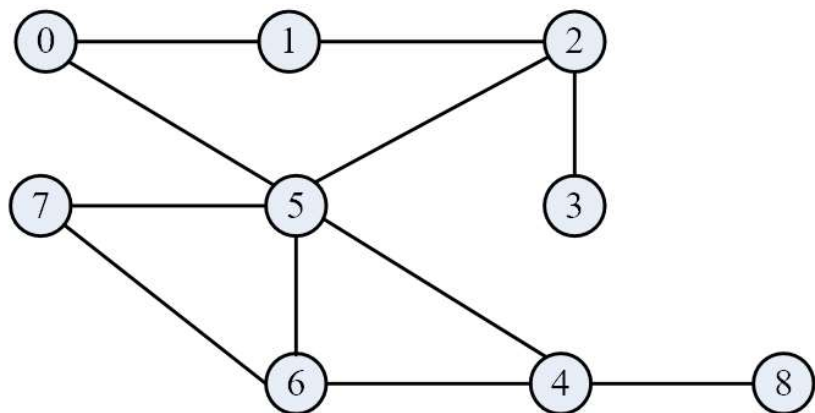
输入: 顶点的编号 v

输出: 无

1. 队列 Q 初始化;
2. 访问顶点 v ; 修改标志 $visited[v] = 1$; 顶点 v 入队列 Q ;
3. while (队列 Q 非空)
 - 3.1 v = 队列 Q 的队头元素出队;
 - 3.2 w = 顶点 v 的第一个邻接点;
 - 3.3 while (w 存在)
 - 3.3.1 如果 w 未被访问, 则
访问顶点 w ; 修改标志 $visited[w] = 1$; 顶点 w 入队列 Q ;
 - 3.3.2 w = 顶点 v 的下一个邻接点;

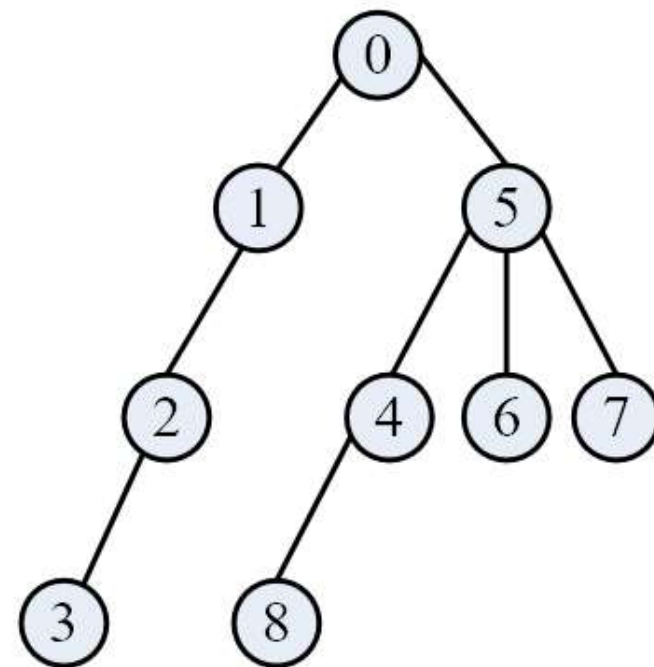
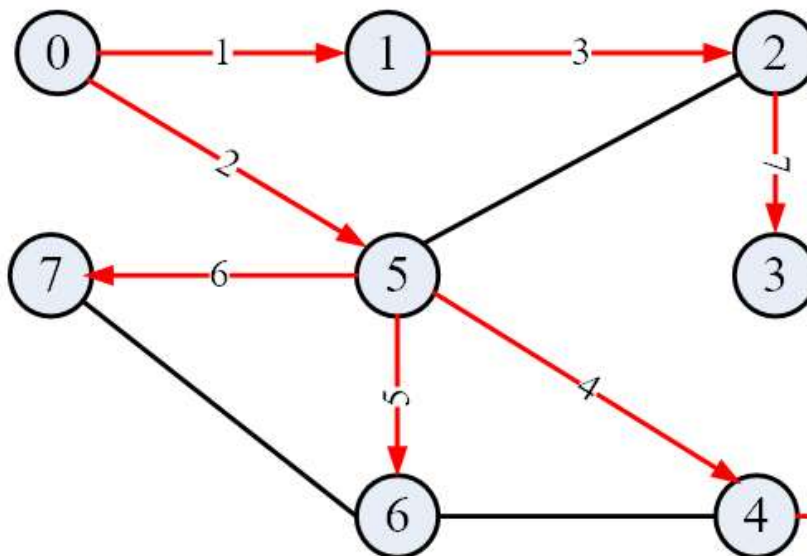


BFS遍历过程1



邻接点编号小的优先

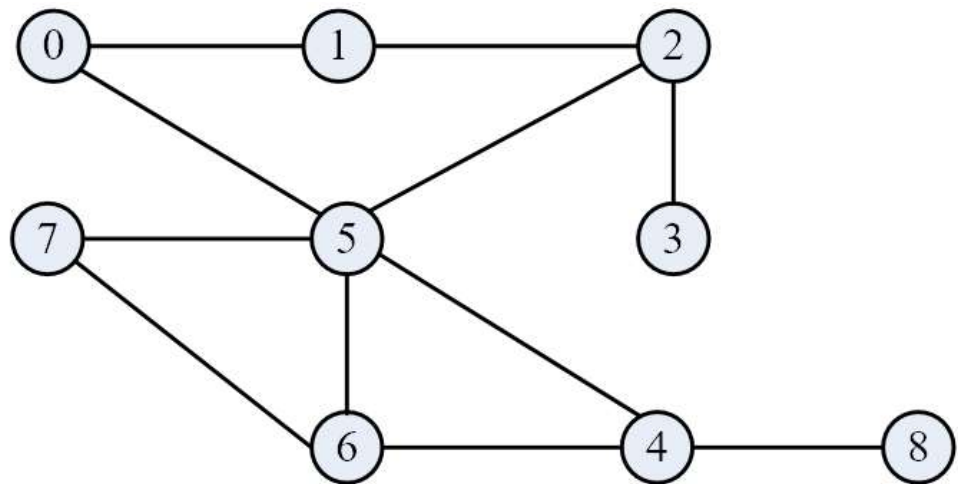
遍历序列015246738



广度优先搜索生成树

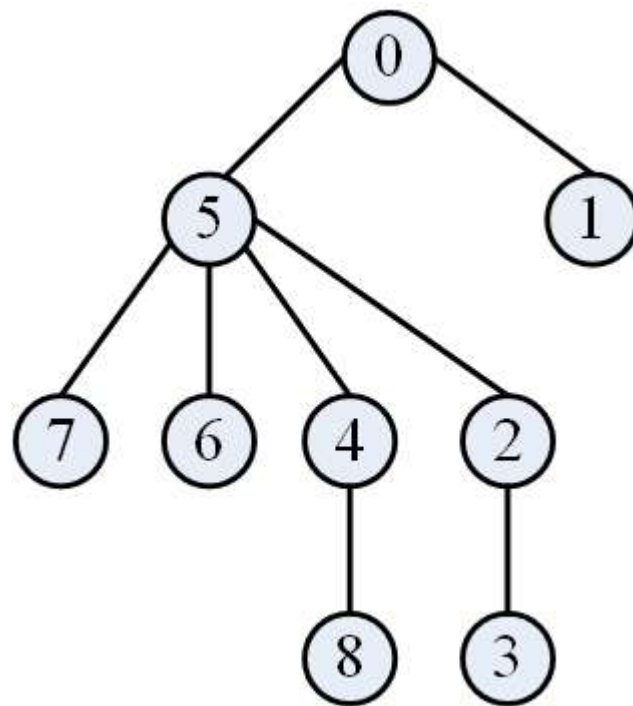


BFS遍历过程2



邻接点编号大的优先

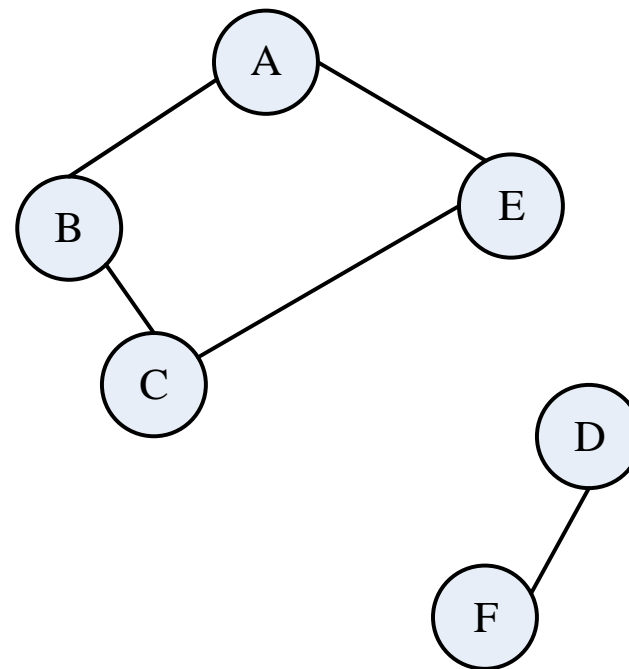
0 5 1 7 6 4 2 8 3





非连通图的BFS遍历

- 对于不连通的图或有向图，只要依次检查图中每一个顶点 v_i ，如果顶点 v_i 未被访问过，则从它出发进行 $\text{BFS}(v_i)$ 遍历即可。
- 如右图非连通图，依次检查A-F的每个顶点是否已被访问过；
- 首先从A出发，可访问到BEC；
- 依次检查顶点B和C，无需从它们出发遍历；
- 接下来从D出发，可访问到F；
- 无需从E和F遍历。遍历结束。整个遍历序列为ABECDF。



A B C D E F

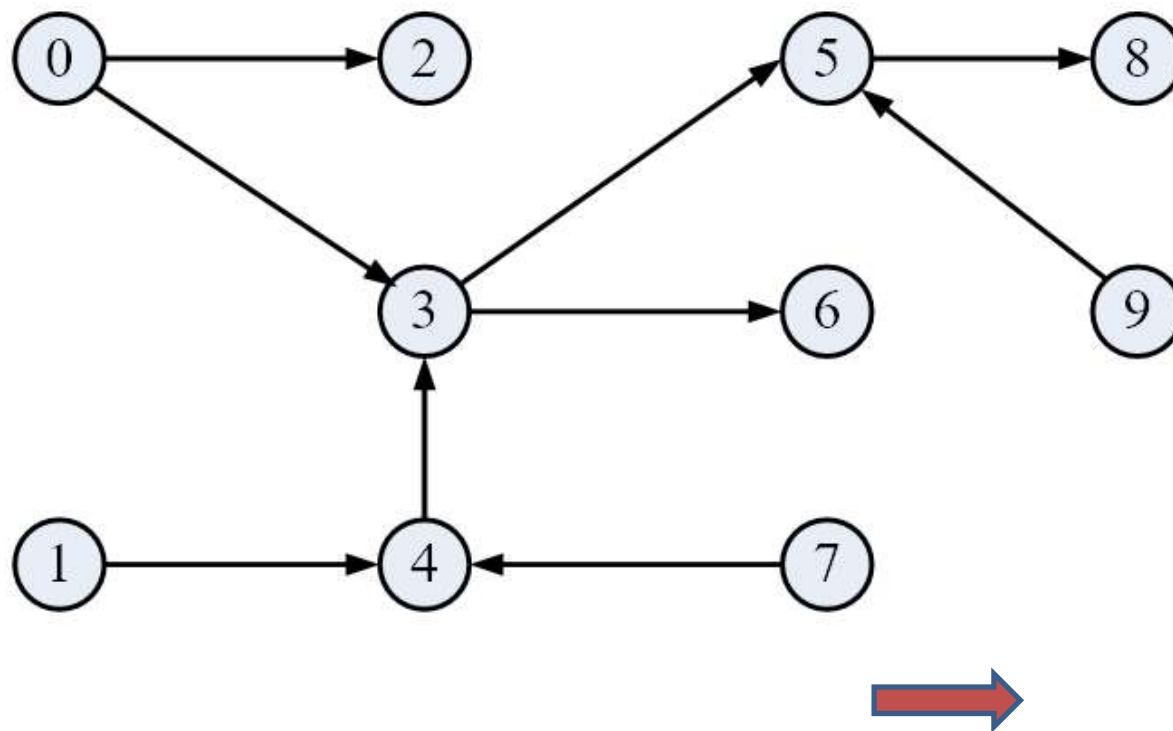


有向图的BFS遍历

- 遍历序列为:
邻接点编号小的优先

023568 1479

0 1 2 3 4 5 6 7 8 9





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

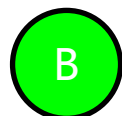
1. 深度优先遍历的特点是“一条路跑到黑”，如果**面临的问题是能找到一个解就可以**，深度优先遍历一般是首选。其搜索深度一般比广度优先遍历要搜索的宽度小很多。
2. 广度优先遍历的特点是层层扩散。如果**面临的问题是要找到一个距离出发点最近的解**，那么广度优先遍历是最好的选择。
3. 广度优先遍历需要程序员自己写个队列，代码比较长。而且这个队列要能同时存储一整层顶点，如果是一棵满二叉树，每层顶点的个数是呈指数级增长的，所以耗费的空间会比较大。

9. 对任意一个图，从某顶点出发进行一次深度优先或广度优先遍历，可访问图的所有顶点。

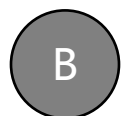


A

正确



B



B

错误

提交

10. 对于图6-2所示无向图，顶点 v_0 到 v_1 的最短路径长度是（ ）。

A

A

1

B

2

C

3

D

4

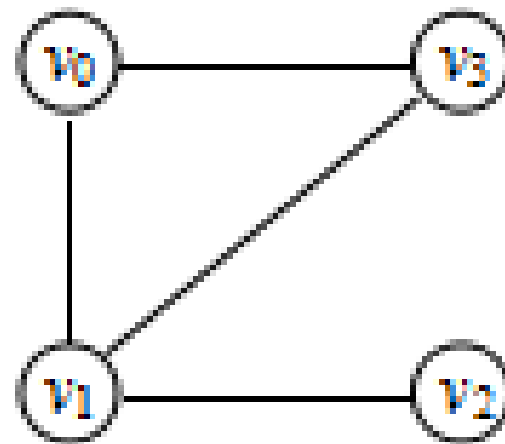


图 6-2 无向图

提交

11. 对于图6-2所示无向图，从顶点 v_0 出发的深度优先遍历序列可能是（ ）。

A $v_0v_2v_3v_1$

B $v_0v_3v_2v_1$

C $v_0v_2v_1v_3$

D $v_0v_1v_3v_2$

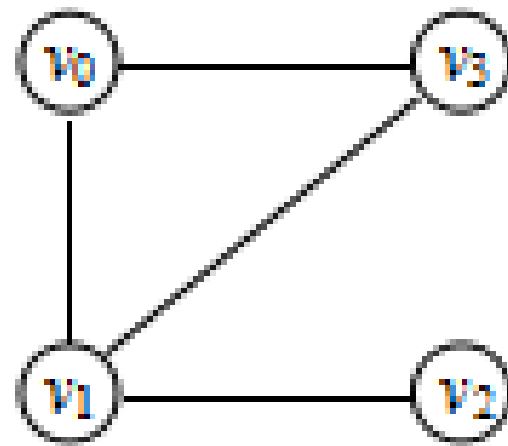


图 6-2 无向图

提交

12. 对于图6-2所示无向图，从顶点 v_0 出发的广度优先遍历序列可能是（ ）。

A $v_0v_2v_3v_1$

B $v_0v_3v_2v_1$

C $v_0v_2v_1v_3$

☒ D $v_0v_1v_3v_2$

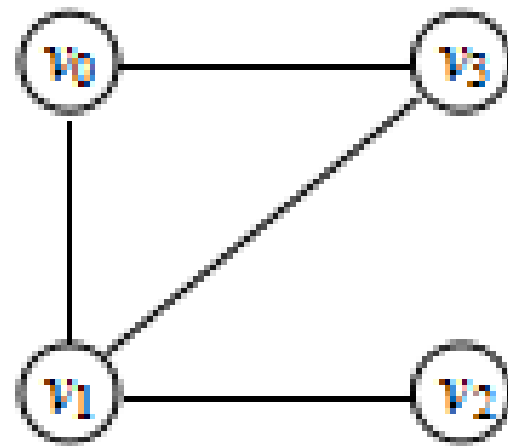


图 6-2 无向图

提交

13. 对于图6-3所示有向网图，顶点 v_0 到 v_3 的最短路径长度是（ ）。

A 1

B 2

C 7

D 8

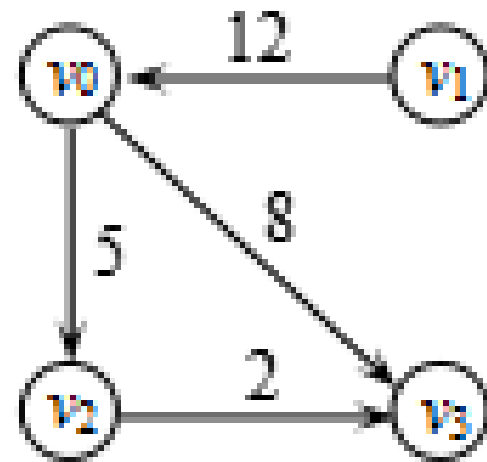


图 6-3 有向网图

提交

14. 对于图6-3所示有向网图，深度优先遍历序列可能是（ ）。

A v0v1v2v3

B v0v2v3v1

C v0v3v1v2

D v0v1v3v2

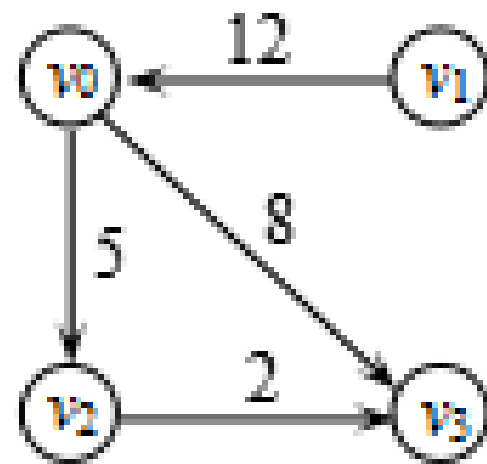


图 6-3 有向网图

提交

15. 对于图6-3所示有向网图，广度优先遍历序列可能是（ ）。

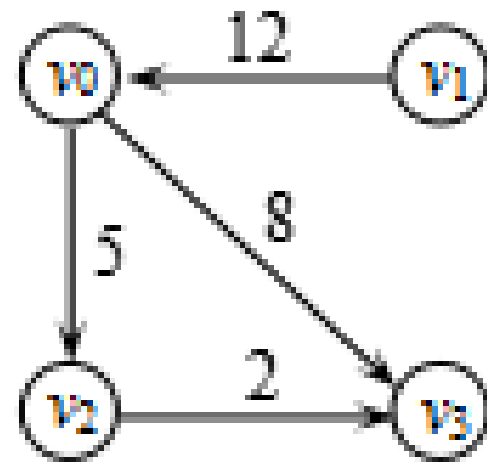


图 6-3 有向网图

A v0v1v2v3

B v0v2v3v1

C v0v3v1v2

D v0v1v3v2

提交

The background of the slide features a complex network diagram. It consists of numerous nodes, represented by small blue and grey dots, interconnected by thin, light blue lines. These connections form a web-like structure that fills the entire frame. Some areas of the network are highlighted with semi-transparent blue polygons, suggesting clusters or specific regions of interest within the graph.

图的存储结构及实现



邻接矩阵存储结构



邻接表存储结构

A complex network diagram with numerous nodes and edges, some highlighted in light blue, forming a spherical shape.

讲什么？



图的邻接矩阵存储结构



邻接矩阵的实现——建立

存储思想

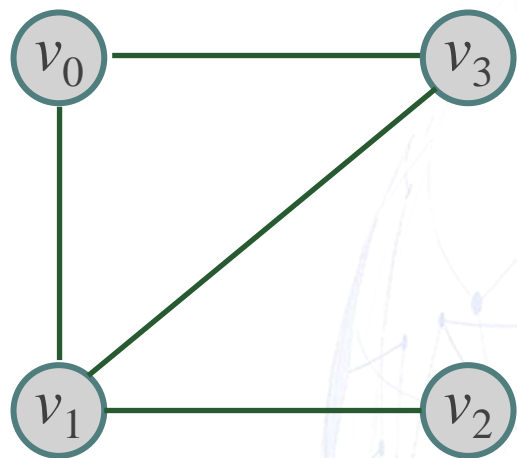
 邻接矩阵也称**数组**表示法，其基本思想是：

- 一维数组：存储图中顶点的信息
- 二维数组（邻接矩阵）：存储图中各顶点之间的邻接关系

设 $G=(V, E)$ 有 n 个顶点，则邻接矩阵是一个 $n \times n$ 的方阵，定义为：

$$\text{edge}[i][j] = \begin{cases} 1 & \text{若}(v_i, v_j) \in E \text{ (或 } \langle v_i, v_j \rangle \in E) \\ 0 & \text{其它} \end{cases}$$

存储示意图



vertex =

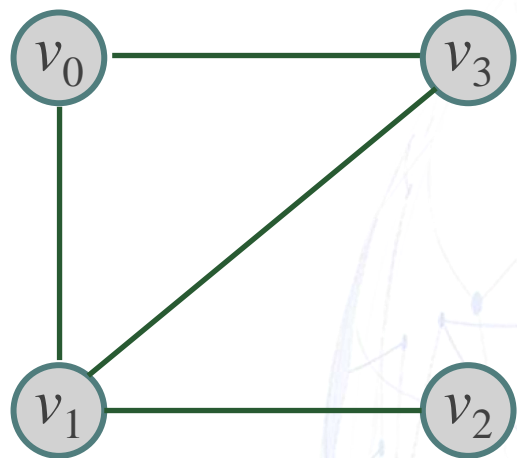
	0	1	2	3
	v_0	v_1	v_2	v_3

edge =

	v_0	v_1	v_2	v_3
v_0	0	1	0	1
v_1	1	0	1	1
v_2	0	1	0	0
v_3	1	1	0	0

- 🕒 无向图的邻接矩阵有什么特点? \Rightarrow 对称矩阵
- 🕒 如何求顶点 v 的度? \Rightarrow 第 v 行 (或第 v 列) 非零元素的个数
- 🕒 如何判断顶点 i 和 j 之间是否存在边? \Rightarrow if (edge[i][j] == 1)

存储示意图



vertex =

	0	1	2	3
	v_0	v_1	v_2	v_3

edge =

	v_0	v_1	v_2	v_3
v_0	0	1	0	1
v_1	1	0	1	1
v_2	0	1	0	0
v_3	1	1	0	0



如何求顶点 i 的所有邻接点?



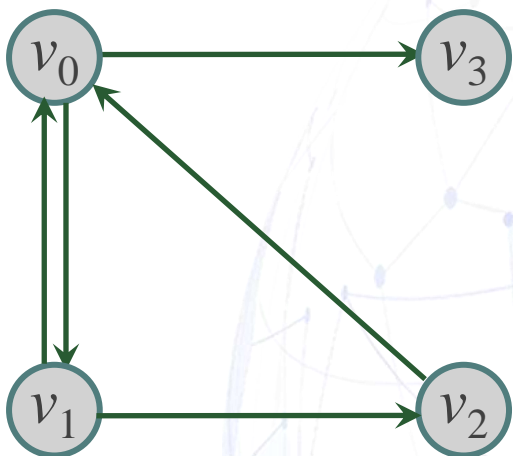
扫描第 i 行

```
for (j = 0; j < n; j++)
```

```
    if (edge[i][j] == 1)
```

顶点 j 是顶点 i 的邻接点

存储示意图



vertex =

	0	1	2	3
	v_0	v_1	v_2	v_3

edge =

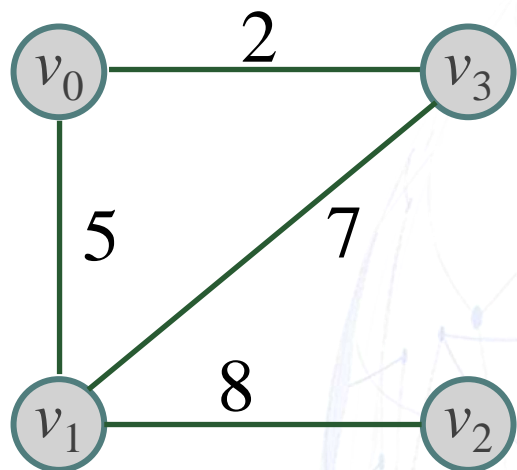
	v_0	v_1	v_2	v_3
v_0	0	1	0	1
v_1	1	0	1	0
v_2	1	0	0	0
v_3	0	0	0	0

🕒 有向图的邻接矩阵一定不对称吗？ \Rightarrow 顶点间存在方向相反的弧

🕒 如何求顶点 v 的出度？ \Rightarrow 第 v 行非零元素的个数

🕒 如何求顶点 v 的入度？ \Rightarrow 第 v 列非零元素的个数

存储示意图



vertex =

	0	1	2	3
v_0				
v_1				
v_2				
v_3				

edge =

	v_0	v_1	v_2	v_3
v_0	0	5	∞	2
v_1	5	0	8	7
v_2	∞	8	0	∞
v_3	2	7	∞	0

网的邻接矩阵可定义为:

$$\text{arc}[i][j] = \begin{cases} w_{ij} & \text{若 } (v_i, v_j) \in E \text{ (或 } \langle v_i, v_j \rangle \in E) \\ 0 & \text{若 } i = j \\ \infty & \text{其他} \end{cases}$$



邻接矩阵的类定义



图的抽象数据类型定义?

ADT Graph

DataModel

...

Operation

CreatGraph: 图的建立

DestroyGraph: 图的销毁

DFSTraverse: 深度优先遍历图

BFSTraverse: 广度优先遍历图

endADT



```
const int MaxSize = 10;
template <typename DataType>
class MGraph
{
public:
    MGraph(DataType a[ ], int n, int e);
    ~MGraph( );
    void DFSTraverse(int v);
    void BFSTraverse(int v);
private:
    DataType vertex[MaxSize];
    int edge[MaxSize][MaxSize];
    int vertexNum, edgeNum;
};
```

构造函数

🕒 建立一个图的函数原型是什么？

🕒 构造函数做什么？

CreatGraph

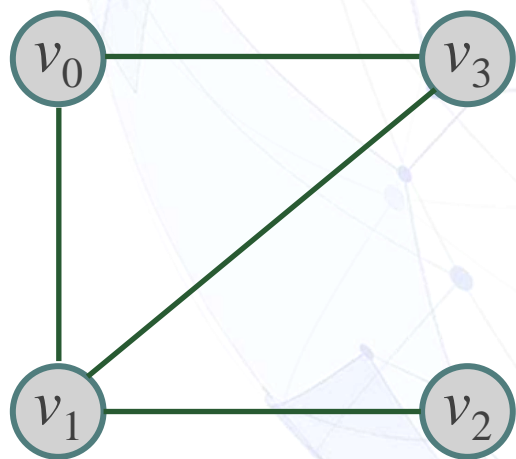
输入： n 个顶点 e 条边

功能： 图的建立

输出： 构造一个含有 n 个顶点 e 条边的图

$a =$

v_0	v_1	v_2	v_3
-------	-------	-------	-------



$vertex =$

v_0	v_1	v_2	v_3
-------	-------	-------	-------

$edge =$

	v_0	v_1	v_2	v_3
v_0	0	1	0	1
v_1	1	0	1	1
v_2	0	1	0	0
v_3	1	1	0	0

$vertexNum$ $edgeNum$

4	4
---	---

伪代码

算法: MGraph

输入: 顶点的数据 $a[n]$, 顶点个数 n , 边的个数 e

输出: 图的邻接矩阵

1. 存储图的顶点个数和边的个数;
2. 将顶点信息存储在一维数组 $vertex$ 中;
3. 初始化邻接矩阵 $edge$;
4. 依次输入每条边并存储在邻接矩阵 $edge$ 中:
 - 4.1 输入边依附的两个顶点的编号 i 和 j ;
 - 4.2 将 $edge[i][j]$ 和 $edge[j][i]$ 的值置为 1;

算法描述

```
template <typename DataType>
MGraph<DataType> :: MGraph(DataType a[ ], int n, int e)
{
    int i, j, k;
    vertexNum = n; edgeNum = e;
    for (i = 0; i < vertexNum; i++)           //存储顶点
        vertex[i] = a[i];
    for (i = 0; i < vertexNum; i++)           //初始化邻接矩阵
        for (j = 0; j < vertexNum; j++)
            edge[i][j] = 0;
    for (k = 0; k < edgeNum; k++)             //依次输入每一条边
    {
        cin >> i >> j;                       //输入边依附的两个顶点的编号
        edge[i][j] = 1; edge[j][i] = 1;      //置有边标志
    }
}
```

析构函数



图的邻接矩阵存储需要销毁吗？

图的邻接矩阵存储是静态存储分配



在图变量退出作用域时自动释放所占内存单元



图的邻接矩阵存储无需销毁



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，空间消耗比较大。



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。



图的邻接表存储结构及实现

讲什么？



图的邻接表存储结构



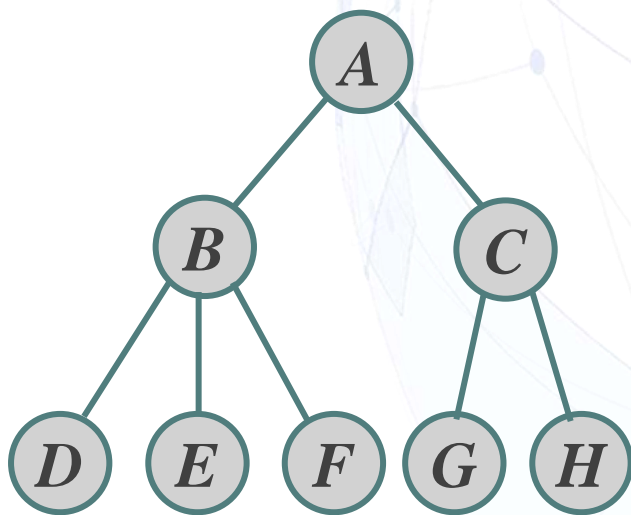
邻接表的实现——建立



邻接表的实现——销毁

存储思想

- 🕒 邻接矩阵存储结构的空间复杂度是多少? $\Rightarrow O(n^2)$
- 🕒 如果采用邻接矩阵存储**稀疏图**, 会出现什么情况? \Rightarrow 稀疏矩阵
- 🕒 树的孩子表示法?



0	A		→	1	→	2	Λ		
1	B		→	3	→	4	→	5	Λ
2	C		→	6	→	7	Λ		
3	D	Λ							
4	E		→	8	Λ				
5	F	Λ							
6	G	Λ							
7	H	Λ							

存储思想

✎ 邻接表存储的基本思想是：

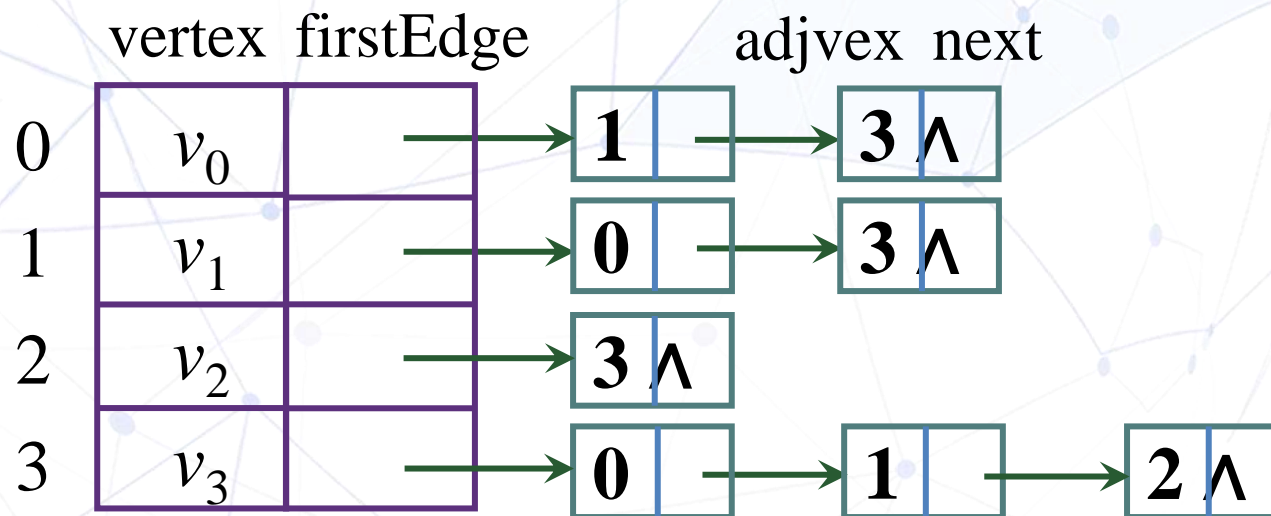
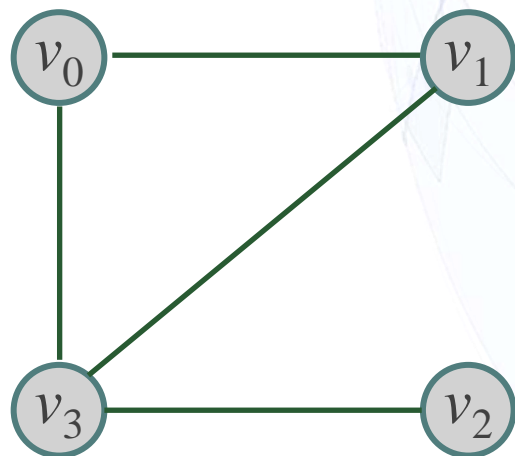
- 边表（邻接表）：顶点 v 的所有邻接点链成的单链表
- 顶点表：所有边表的头指针和存储顶点信息的一维数组



设图有 n 个顶点 e 条边，邻接表的空间复杂度是多少？

⇒ $O(n+2e)$

⇒ $O(n+e)$



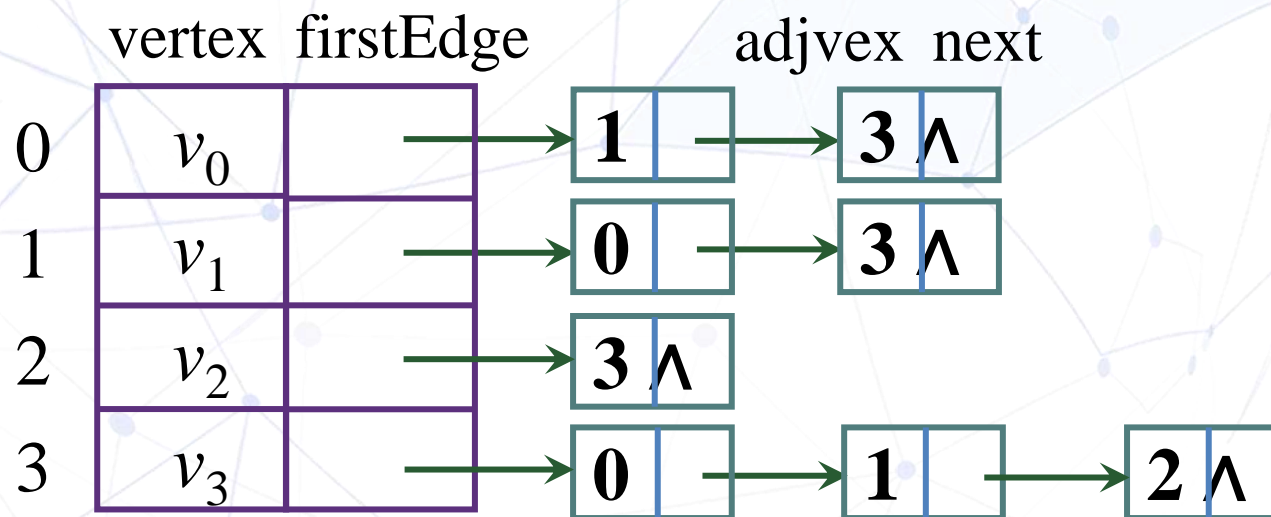
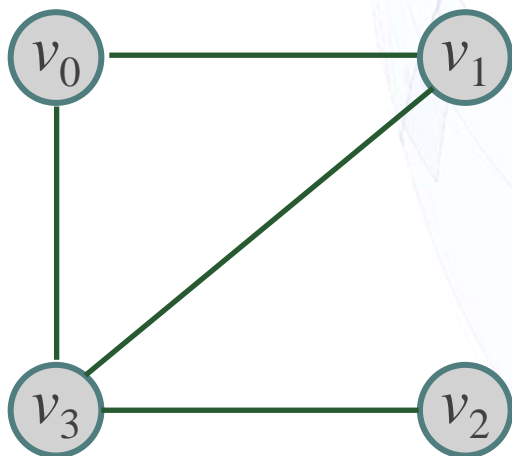
邻接表表示不唯一

基本操作

🕒 边表中的结点表示什么? \Rightarrow 对应图中的一条边

🕒 如何判断顶点 i 和顶点 j 之间是否存在边?

\Rightarrow 测试顶点 i 的边表中是否存在数据域为 j 的结点

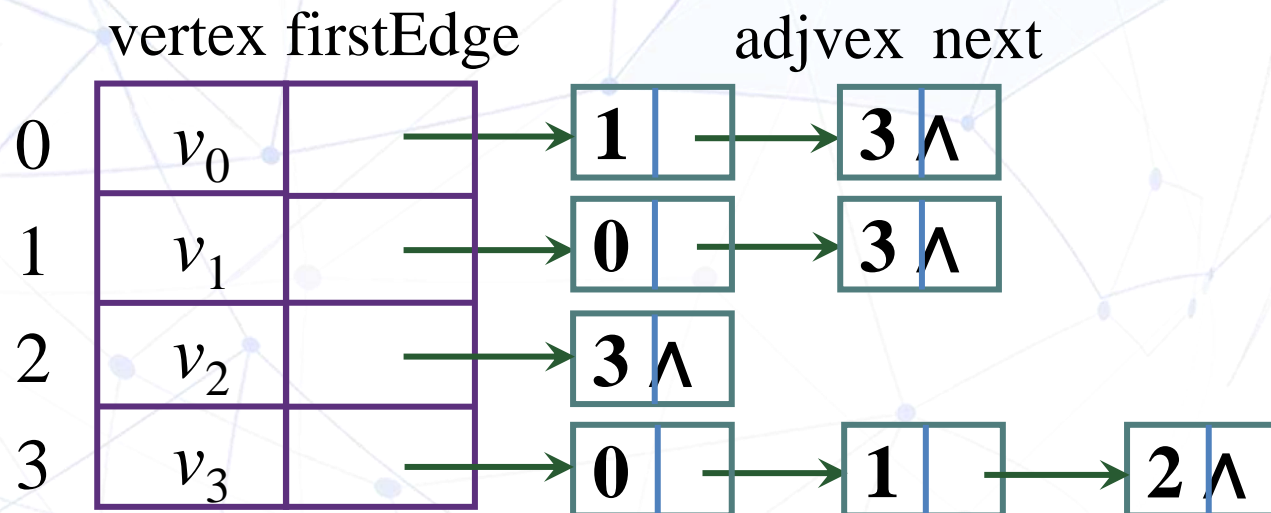
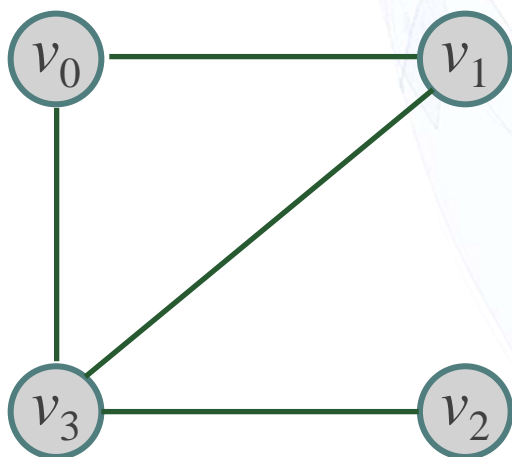


基本操作

🕒 如何求顶点 v 的所有邻接点? \Rightarrow 顶点 i 的边表中的所有结点

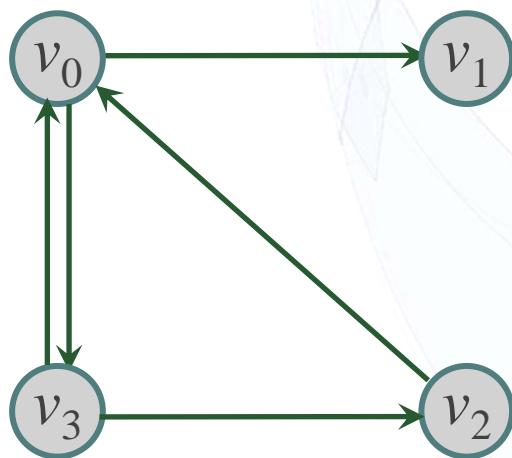
🕒 如何求顶点 v 的度? \Rightarrow 顶点 v 的边表中结点的个数

```
p = adjlist[v].firstEdge; count = 0;  
while (p != nullptr)  
{  
    count++; p = p->next;  
}
```



存储有向图

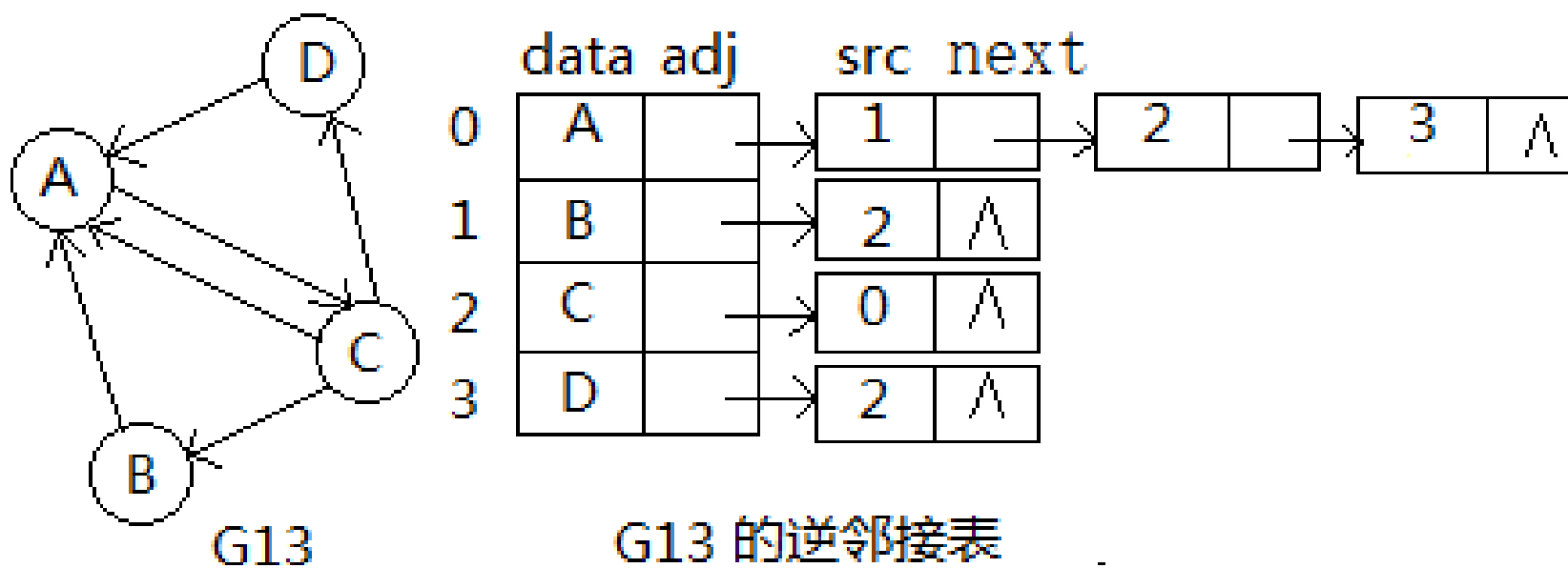
- 🕒 邻接表如何存储有向图? \Rightarrow 出边表
- 🕒 如何求顶点 v 的出度? \Rightarrow 顶点 v 的出边表中结点的个数
- 🕒 如何求顶点 v 的入度? \Rightarrow 所有出边表中数据域为 v 的结点个数



	vertex	firstEdge		adjvex	next
0	v_0		\rightarrow	1	\rightarrow 3 \wedge
1	v_1	\wedge			
2	v_2		\rightarrow	0	\wedge
3	v_3		\rightarrow	0	\rightarrow 2 \wedge

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。



存储带权图



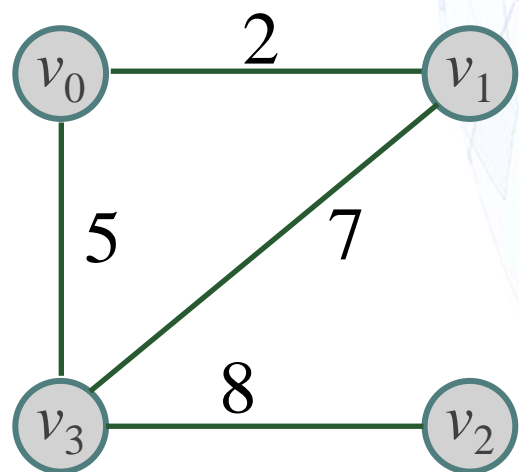
邻接表如何存储带权图?



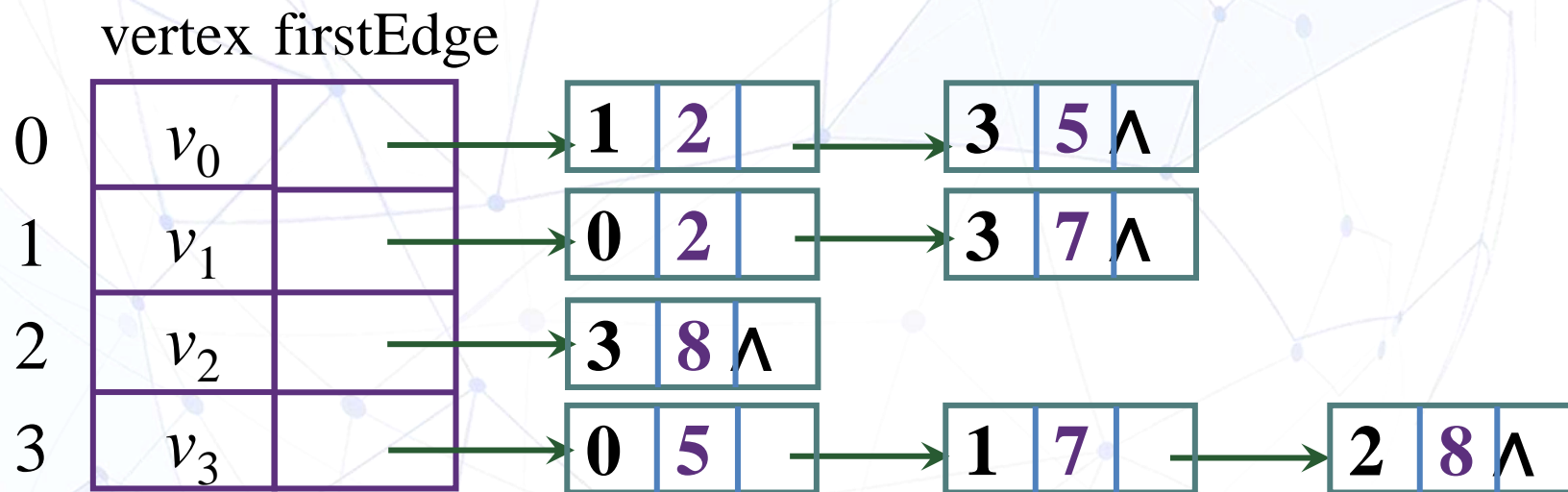
权值存储在边表中



如何定义带权图的邻接表存储结构?



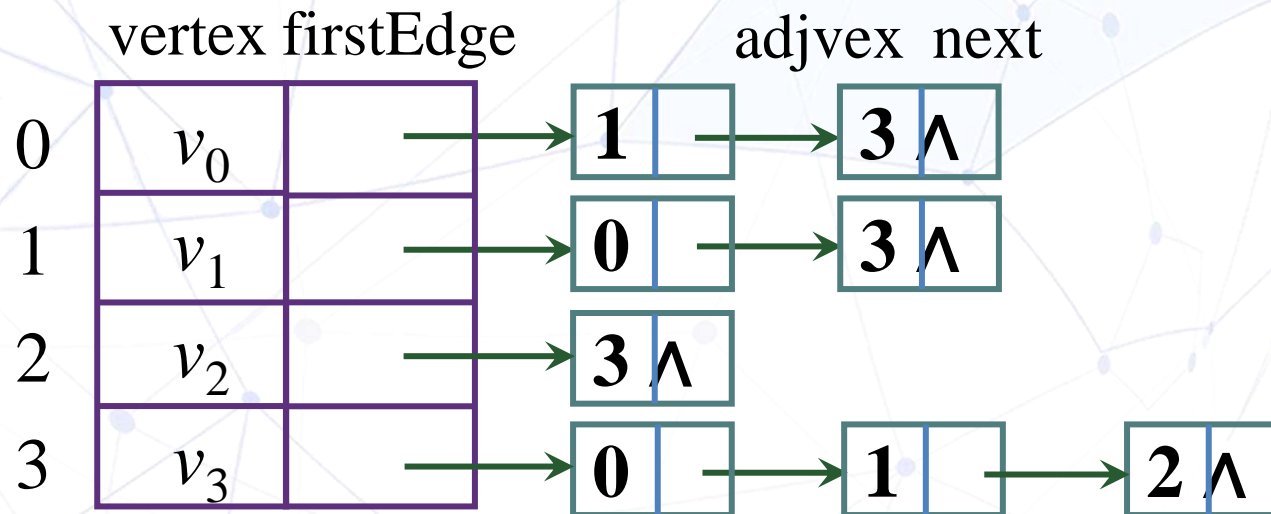
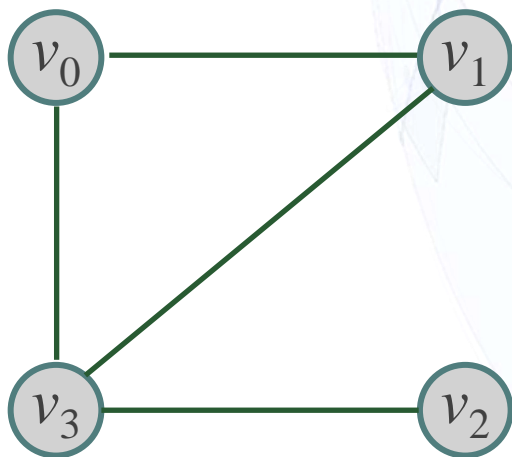
adjvex	weight	next
--------	--------	------



邻接表的类定义

```
struct EdgeNode
{
    int adjvex;
    EdgeNode *next;
};
```

```
template <typename DataType>
struct VertexNode
{
    DataType vertex;
    EdgeNode *firstEdge;
};
```



邻接表的类定义



图的抽象数据类型定义？

ADT Graph

DataModel

...

Operation

CreatGraph: 图的建立

DestroyGraph: 图的销毁

DFTraverse: 深度优先遍历图

BFTraverse: 广度优先遍历图

endADT



```
const int MaxSize = 10;
template <typename DataType>
class ALGraph
{
public:
    ALGraph(DataType a[ ], int n, int e);
    ~ALGraph( );
    void DFTraverse(int v);
    void BFTraverse(int v);
private:
    VertexNode<DataType> adjlist[MaxSize];
    int vertexNum, edgeNum;
};
```

构造函数

🕒 建立一个图的函数原型是什么？

🕒 构造函数做什么？

CreatGraph

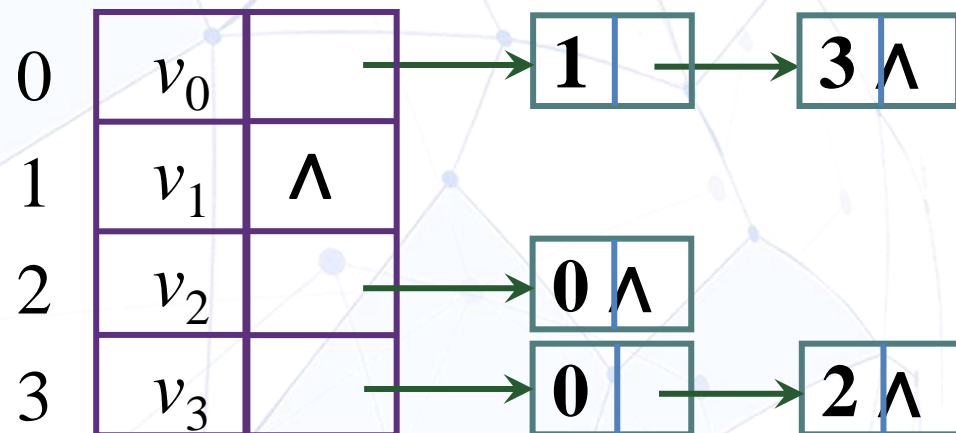
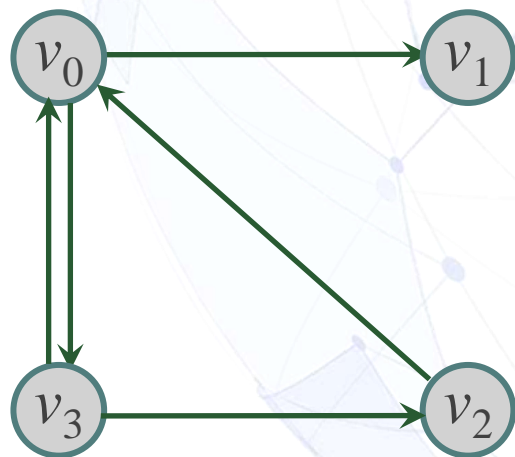
输入： n 个顶点 e 条边

功能： 图的建立

输出： 构造一个含有 n 个顶点 e 条边的图

$a =$

v_0	v_1	v_2	v_3
-------	-------	-------	-------



vertexNum edgeNum

4	4
---	---

伪代码

算法: ALGraph

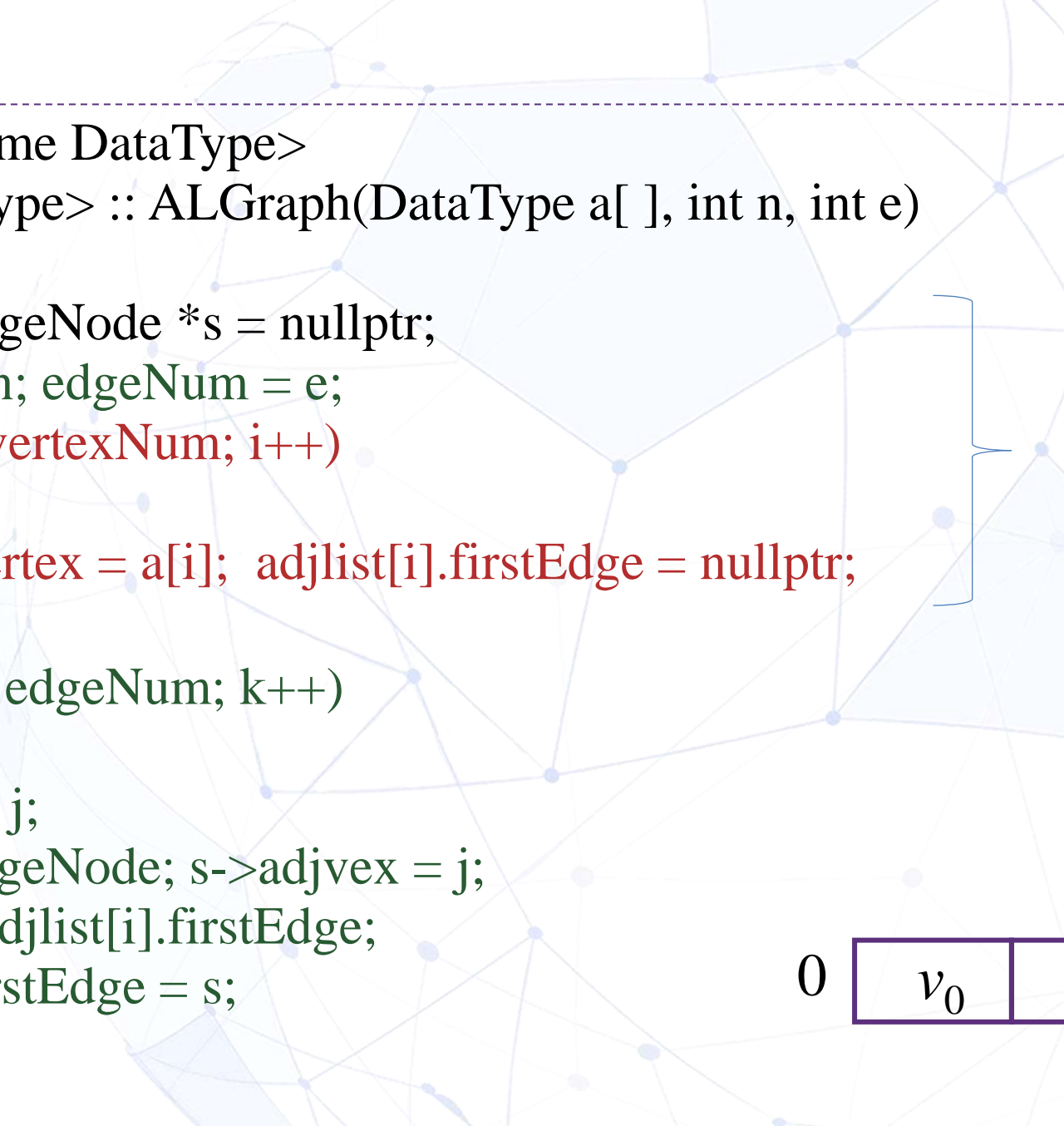
输入: 顶点的数据信息 $a[n]$, 顶点个数 n , 边的个数 e

输出: 图的邻接表

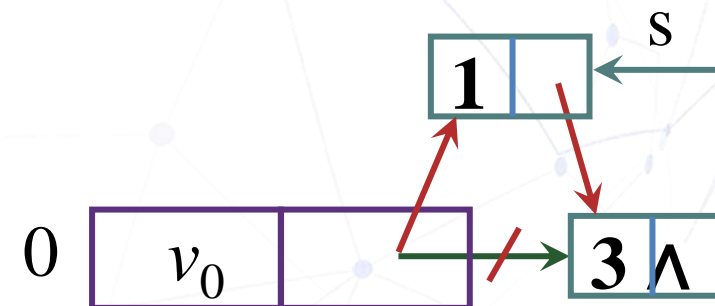
1. 存储图的顶点个数和边的个数;
2. 将顶点信息存储在顶点表中, 将该顶点边表的头指针初始化为`nullptr`;
3. 依次输入边的信息并存储在边表中:
 - 3.1 输入边所依附的两个顶点的编号 i 和 j ;
 - 3.2 生成边表结点 s , 其邻接点的编号为 j ;
 - 3.3 将结点 s 插入到第 i 个边表的表头;

算法描述

```
template <typename DataType>
ALGraph<DataType> :: ALGraph(DataType a[ ], int n, int e)
{
    int i, j, k;  EdgeNode *s = nullptr;
    vertexNum = n; edgeNum = e;
    for (i = 0; i < vertexNum; i++)
    {
        adjlist[i].vertex = a[i]; adjlist[i].firstEdge = nullptr;
    }
    for (k = 0; k < edgeNum; k++)
    {
        cin >> i >> j;
        s = new EdgeNode; s->adjvex = j;
        s->next = adjlist[i].firstEdge;
        adjlist[i].firstEdge = s;
    }
}
```



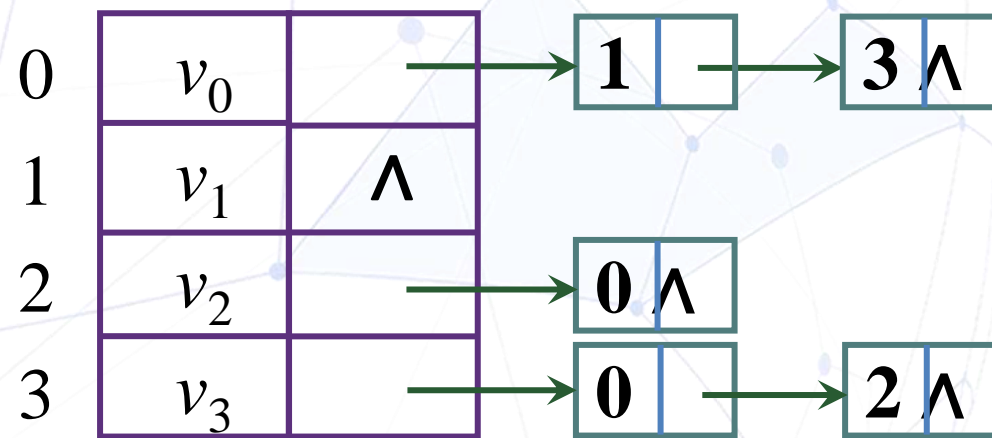
v_0	Λ
v_1	Λ
v_2	Λ
v_3	Λ



析构函数

在邻接表存储中，须释放所有在程序运行过程中申请的边表结点

```
template <typename DataType>
ALGraph<DataType>::~~ALGraph( )
{
    EdgeNode *p = nullptr, *q = nullptr;
    for (int i = 0; i < vertexNum; i++)
    {
        p = q = adjlist[i].firstEdge;
        while (p != nullptr)
        {
            p = p->next;
            delete q;
            q = p;
        }
    }
}
```





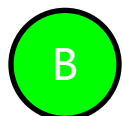
邻接表存储特点

- 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。

2. 无向图的邻接矩阵一定是对称的，有向图的邻接矩阵一定是不对称的。



正确



错误

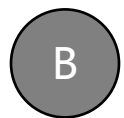
提交

3. 用邻接矩阵存储图，所占用的存储空间大小只与图中顶点个数有关，与图的边数无关。



A

正确



B

错误

提交

4. 图采用邻接矩阵存储，查找某顶点的所有邻接点，时间复杂度是（ ）。

- ☐ A $O(1)$
- ☒ B $O(n)$
- ☐ C $O(n+e)$
- ☐ D $O(n^2)$

提交

5. 对于图6-4所示有向图，画出邻接矩阵存储示意图。

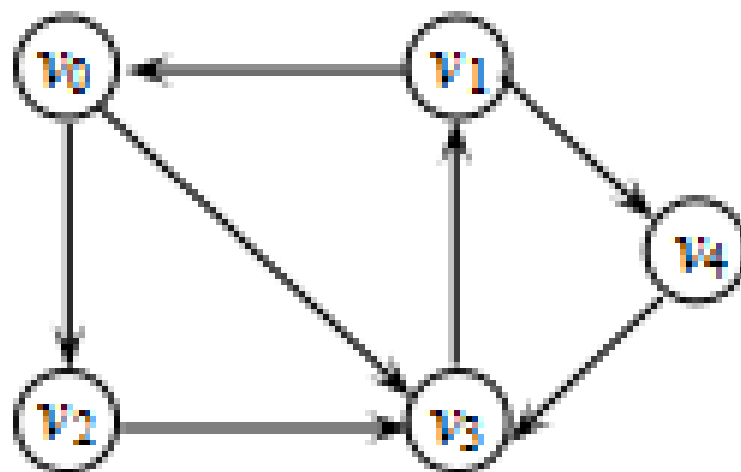
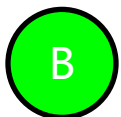


图 6-4 有向图

1. 图采用邻接表存储，空间复杂度只与顶点个数有关，和边数无关。



正确



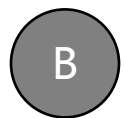
错误

提交

2. 在图的邻接表存储中，存在两类结点：顶点表结点和边表结点。



正确



错误

提交

3. 对于图6-6所示无向网图，给出邻接表存储示意图。

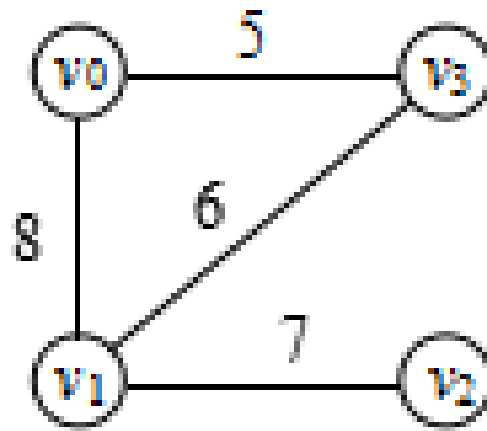


图 6-6 无向网图

不同存储结构下的图遍历算法实现



深度优先遍历——邻接矩阵/邻接表的实现



广度度优先遍历——邻接矩阵/邻接表的实现

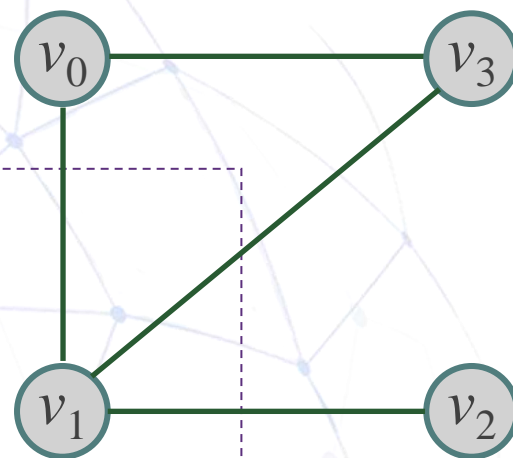
深度遍历

算法: DFTraverse

输入: 顶点的编号 v

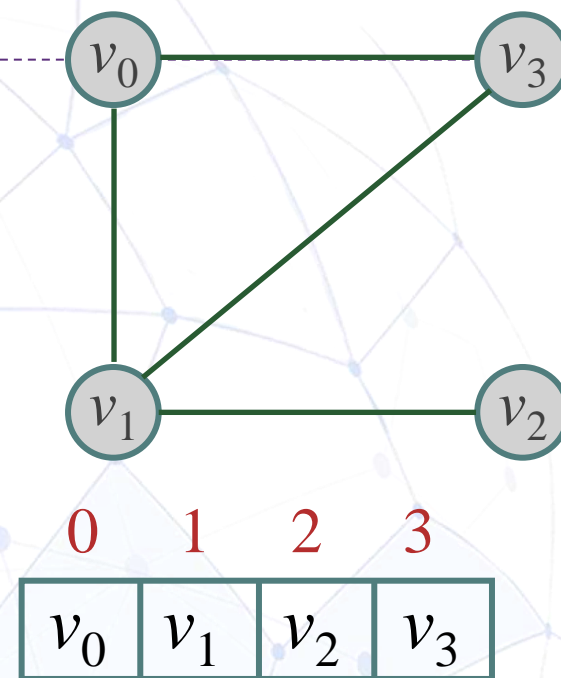
输出: 图的深度优先遍历序列

1. 访问顶点 v ; 修改标志 $\text{visited}[v] = 1$;
2. $j =$ 顶点 v 的第一个邻接点;
3. while (j 存在)
 - 3.1 if (j 未被访问) 从顶点 j 出发递归执行该算法;
 - 3.2 $j =$ 顶点 v 的下一个邻接点;



邻接矩阵中实现的深度遍历

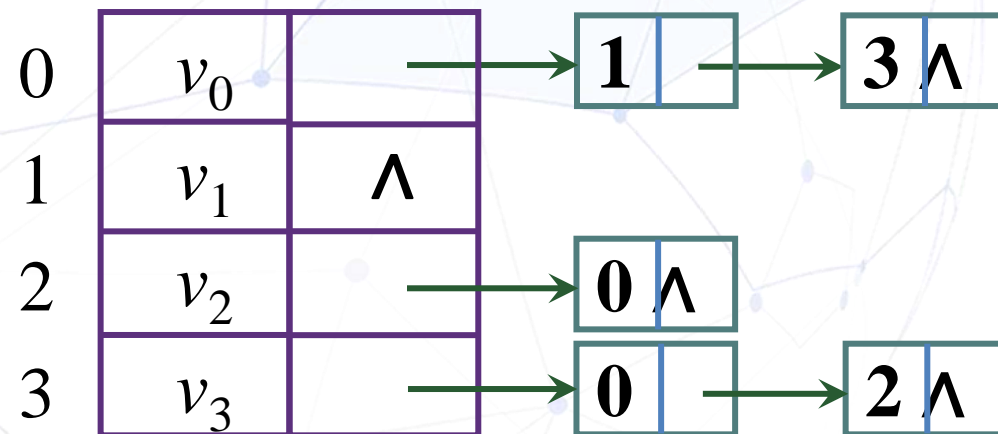
```
template <typename DataType>
void MGraph<DataType> :: DFTraverse(int v)
{
    cout << vertex[v]; visited[v] = 1;
    for (int j = 0; j < vertexNum; j++)
        if (edge[v][j] == 1 && visited[j] == 0)
            DFTraverse( j );
}
```



	v_0	v_1	v_2	v_3
v_0	0	1	0	1
v_1	1	0	1	1
v_2	0	1	0	0
v_3	1	1	0	0

邻接表中实现的深度遍历

```
template <typename DataType>
void ALGraph<DataType> :: DFTraverse(int v)
{
    int j; EdgeNode *p = nullptr;
    cout << adjlist[v].vertex; visited[v] = 1;
    p = adjlist[v].firstEdge;
    while (p != nullptr)
    {
        j = p->adjvex;
        if (visited[j] == 0) DFTraverse(j);
        p = p->next;
    }
}
```



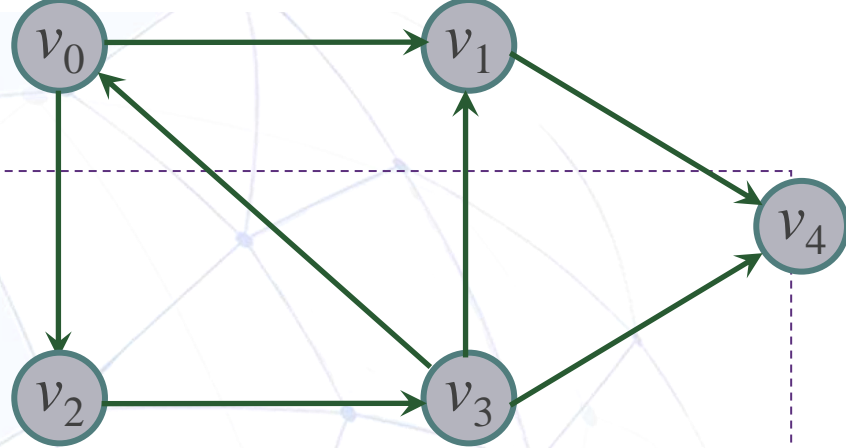
广度遍历

算法: BFTTraverse

输入: 顶点的编号 v

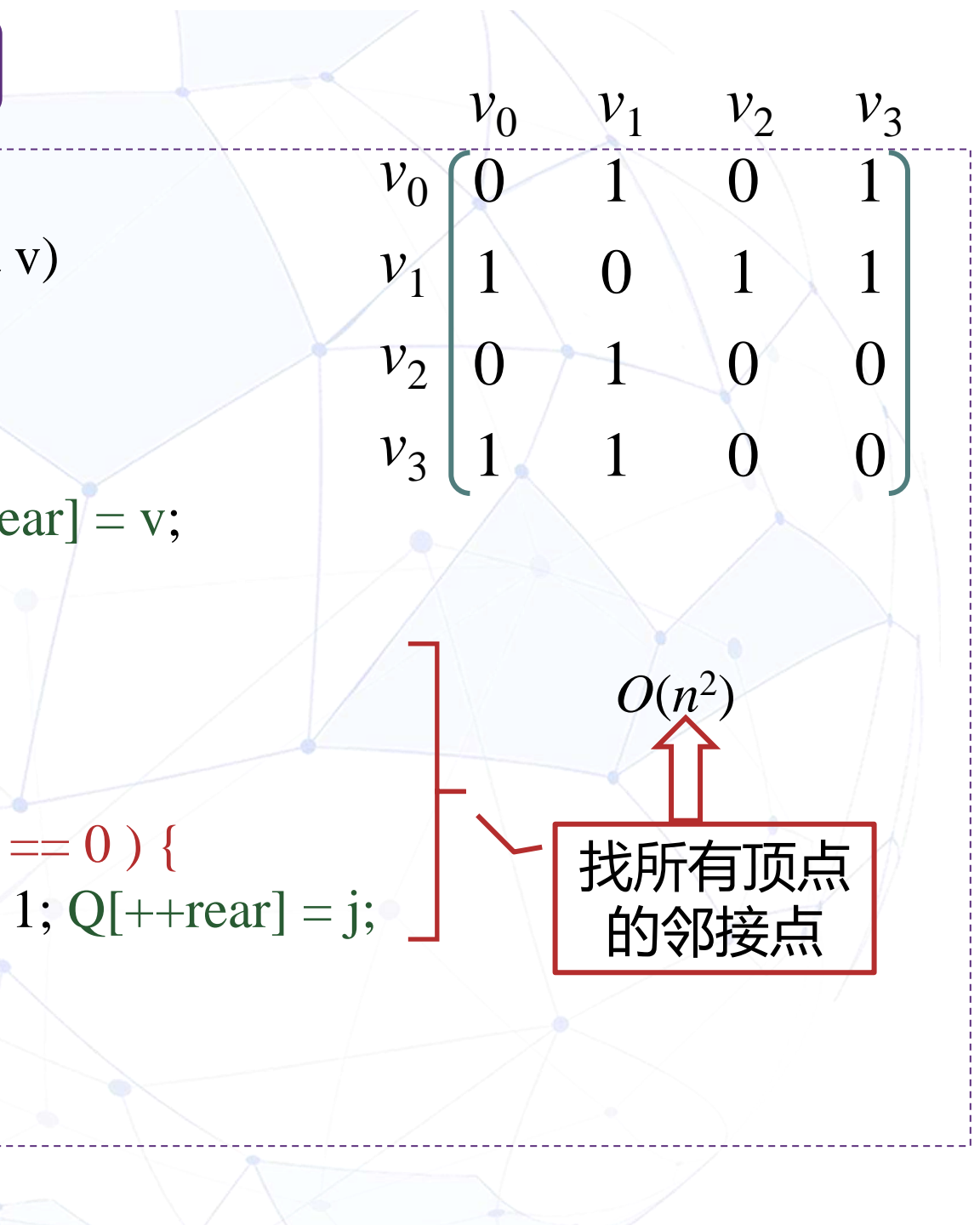
输出: 图的广度优先遍历序列

1. 队列 Q 初始化;
2. 访问顶点 v ; 修改标志 $\text{visited}[v] = 1$; 顶点 v 入队列 Q ;
3. while (队列 Q 非空)
 - 3.1 i = 队列 Q 的队头元素出队;
 - 3.2 j = 顶点 v 的第一个邻接点;
 - 3.3 while (j 存在)
 - 3.3.1 如果 j 未被访问, 则
访问顶点 j ; 修改标志 $\text{visited}[j] = 1$; 顶点 j 入队列 Q ;
 - 3.3.2 j = 顶点 i 的下一个邻接点;



邻接矩阵中实现的广度遍历

```
template <typename DataType>
void MGraph<DataType> :: BFTraverse(int v)
{
    int w, j, Q[MaxSize];
    int front = -1, rear = -1;
    cout << vertex[v]; visited[v] = 1; Q[++rear] = v;
    while (front != rear)
    {
        w = Q[++front];
        for (j = 0; j < vertexNum; j++)
            if (edge[w][j] == 1 && visited[j] == 0 ) {
                cout << vertex[j]; visited[j] = 1; Q[++rear] = j;
            }
    }
}
```



	v_0	v_1	v_2	v_3
v_0	0	1	0	1
v_1	1	0	1	1
v_2	0	1	0	0
v_3	1	1	0	0

$O(n^2)$

找所有顶点的邻接点

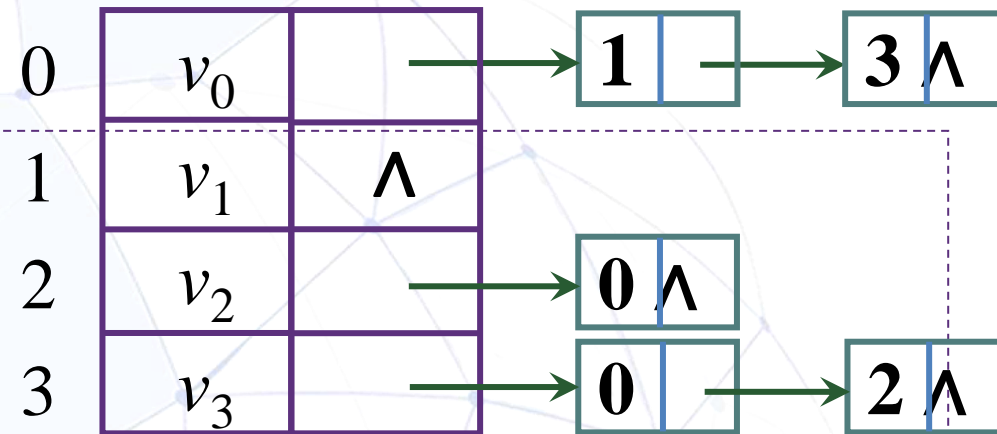
邻接表中实现的广度遍历

```
template <typename DataType>
void ALGraph<DataType> :: BFTraverse(int v)
{
    int w, j, Q[MaxSize]; int front = -1, rear = -1;
    EdgeNode *p = nullptr;
    cout << adjlist[v].vertex; visited[v] = 1; Q[++rear] = v;
    while (front != rear)
    {
        w = Q[++front];
        p = adjlist[w].firstEdge;
        while (p != nullptr)
        {
            j = p->adjvex;
            if (visited[j] == 0) {
                cout << adjlist[j].vertex; visited[j] = 1; Q[++rear] = j;
            }
            p = p->next;
        }
    }
}
```

每个顶点出队 1 次

$$\Rightarrow O(n) + O(e) = O(n+e)$$

找所有顶点的邻接点



6. 基于邻接矩阵存储的广度优先遍历图算法如下，请在每个横线处填写适当的语句或表达式。

```
void MGraph<DataType> :: BFTraverse(int v)
{
    int w, j, Q[MaxSize];
    int front = -1, rear = -1;
    cout << vertex[v]; visited[v] = 1; Q[++rear] = v;
    while (front != rear)
    {
        w = Q[++front];
        for (j = 0; j < vertexNum; j++)
            if ( _____ ① _____ && visited[j] == 0 ) {
                cout << vertex[j]; visited[j] = 1; _____ ② _____
            }
    }
}
```

6. 基于邻接表存储的深度优先遍历图算法如下，请在每个横线处填写适当的语句或表达式。

```
void ALGraph<DataType> :: DFTraverse(int v)
{
    int j;
    EdgeNode *p = nullptr;
    cout << adjlist[v].vertex; visited[v] = 1;
    ①
    while (p != nullptr)
    {
        ②
        if (visited[j] == 0) DFTraverse(j);
        ③
    }
}
```

作答

更复杂的存储结构 (*)



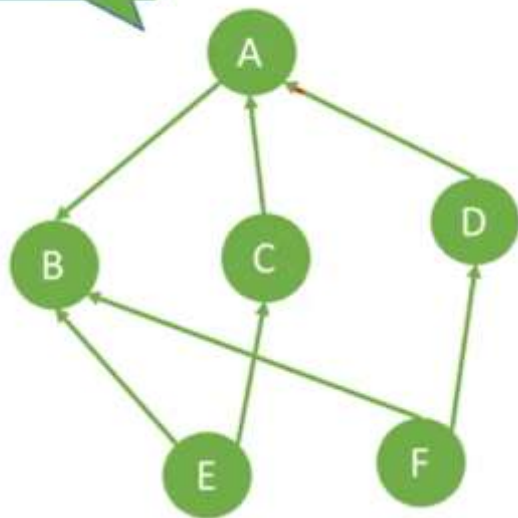
十字链表-有向图



邻接多重表-无向图

Why 十字链表?

有向图



	data	*first
0	A	→ 1 ^
1	B	^
2	C	→ 0 ^
3	D	→ 0 ^
4	E	→ 1 → 2 ^
5	F	→ 1 → 3 ^

	A	B	C	D	E	F
A	0	1	1	1	0	0
B	1	0	0	0	1	1
C	1	0	0	0	1	0
D	1	0	0	0	0	1
E	0	1	1	0	0	0
F	0	1	0	1	0	0

找顶点的入边不方便

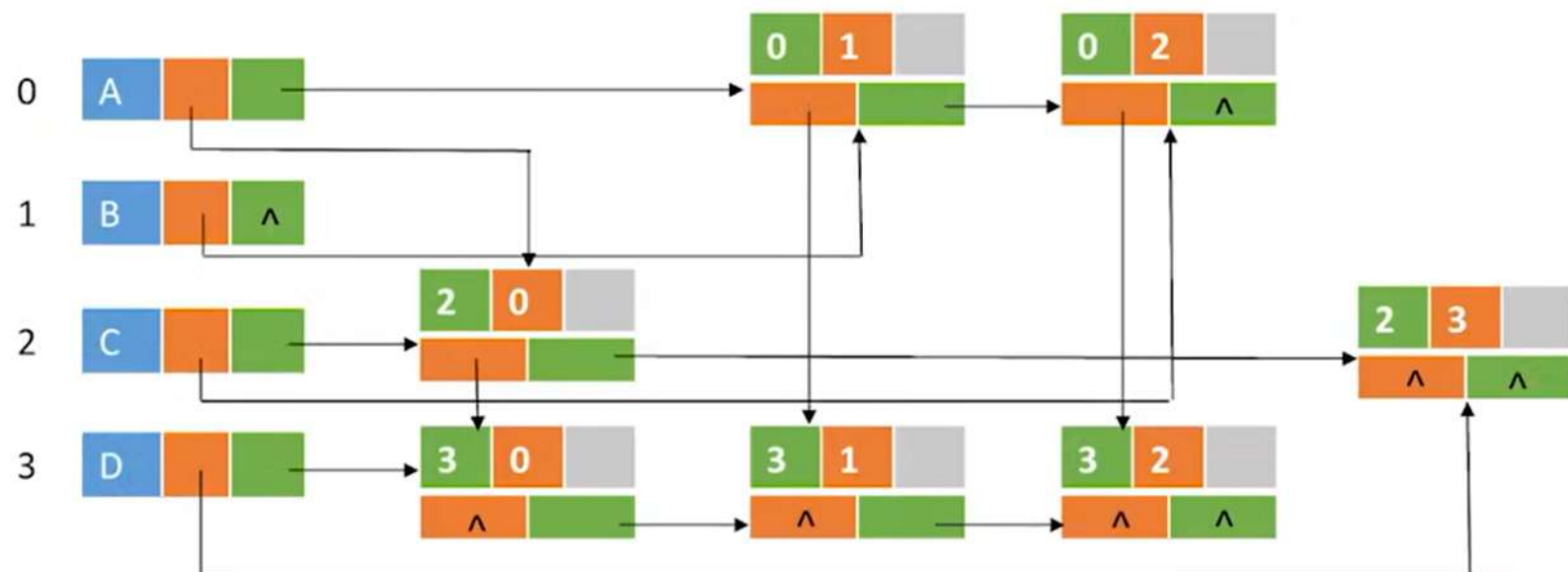
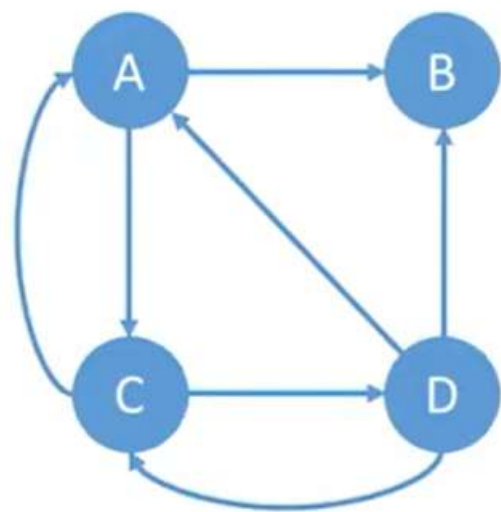
	邻接表	邻接矩阵
空间复杂度	无向图 $O(V + 2 E)$ ；有向图 $O(V + E)$	$O(V ^2)$
计算度/出度/入度	计算有向图的度、入度不方便，其余很方便	必须遍历对应行或列
找相邻的边	找有向图的入边不方便，其余很方便	必须遍历对应行或列

空间复杂度
高 $O(|V|^2)$



十字链表 (*)

十字链表(针对有向图)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。



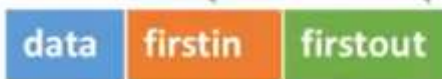
用数组顺序存储

数据域

该顶点作为弧头的第一条弧

该顶点作为弧尾的第一条弧

顶点结点:



弧尾顶点编号

弧头顶点编号

权值

弧结点:



弧头相同的下一条弧

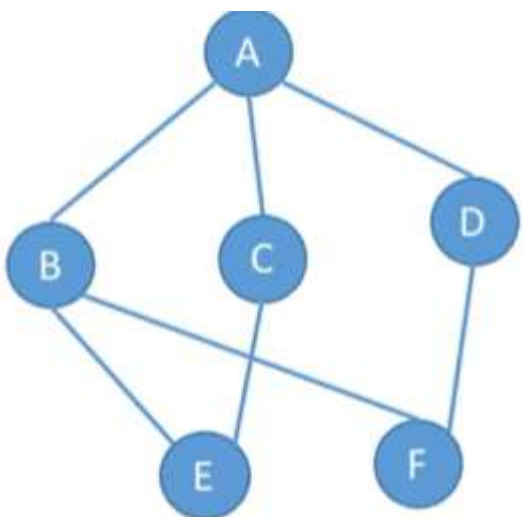
弧尾相同的下一条弧

空间复杂度: $O(|V|+|E|)$

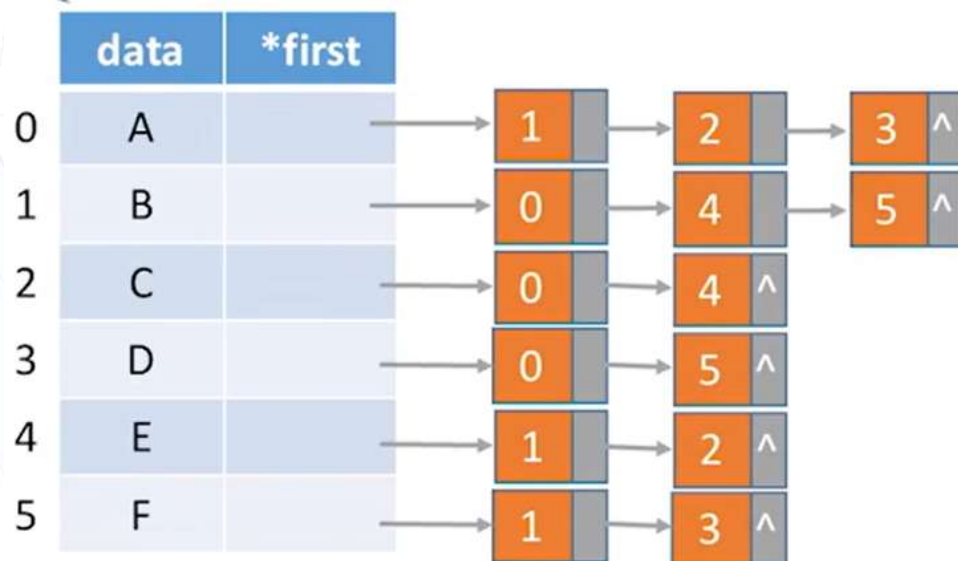
邻接多重表 (*)

邻接表(针对无向图)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，如删除、遍历所有边时因重复而不方便



无向图



每条边对应两份冗余信息，
删除顶点、删除边等操作
时间复杂度高

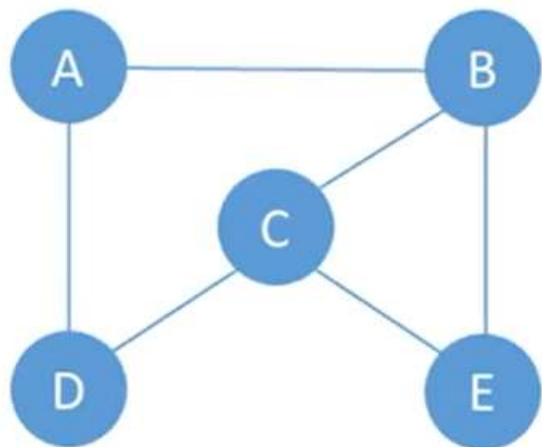
	A	B	C	D	E	F
A	0	1	1	1	0	0
B	1	0	0	0	1	1
C	1	0	0	0	1	0
D	1	0	0	0	0	1
E	0	1	1	0	0	0
F	0	1	0	1	0	0

空间复杂度高
 $O(|V|^2)$

邻接多重表*

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。



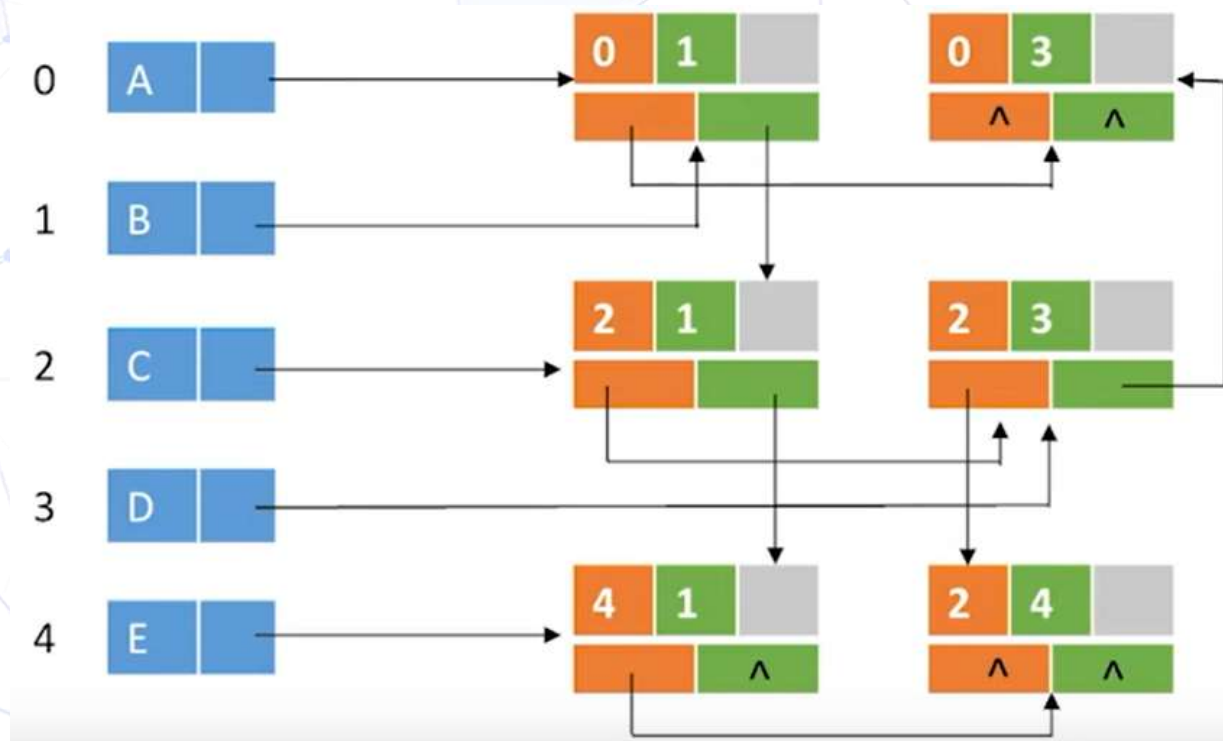
数据域

与该顶点相连
的第一条边

顶点结点:

data

firstedge



每条边只对
应一份数据

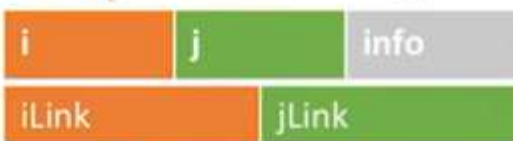
空间复杂度: $O(|V|+|E|)$

注意: 邻接多重表只适用于无向图, 十字链表适用于有向图, 结构上最大的不同表现在“顶点结点”

边的两个顶
点编号i,j

权值

边结点:



依附于顶点i
的下一条边

依附于顶点j
的下一条边

图的存储结构总结

	邻接矩阵	邻接表	<u>十字链表</u>	邻接多重表
空间复杂度	$O(V ^2)$	无向图 $O(V + 2 E)$ 有向图 $O(V + E)$	$O(V + E)$	$O(V + E)$
找相邻边	遍历对应行或列 时间复杂度为 $O(V)$	找有向图的入边必须遍历整个邻接表	很方便	很方便
删除边或顶点	删除边很方便，删除顶点需要大量移动数据	无向图中删除边或顶点都不方便	很方便	很方便
适用于	稠密图	稀疏图和其他	只能存有向图	只能存无向图
表示方式	唯一	不唯一	不唯一	不唯一

The background of the slide features a complex network diagram. It consists of numerous nodes, represented by small blue and purple dots, interconnected by thin, light blue lines. These connections form a dense web of triangles and other polygons, creating a spherical or globe-like structure. The overall color palette is light blue and white, with the network lines and nodes providing a subtle texture.

图的连通性



无向图的连通性



有向图的连通性



无向图的连通性（掌握）

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 $\text{IsConnect}(\text{graph})$

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  $v \leftarrow 0$  to  $\text{graph}.n\_verts-1$  do //初始化各顶点的访问标志为未访问
2.    $visited[v] \leftarrow \text{false}$ 
3. end
4.  $count \leftarrow 0$ 
5. for  $v \leftarrow 0$  to  $\text{graph}.n\_verts-1$  do
6. | if  $visited[v] = \text{false}$  then
7. | |  $count \leftarrow count + 1$ 
8. | |  $\text{BFS}(\text{graph}, v, visited)$  |
9. | end
10. end
```



无向图的连通性

```
11. if  $count=1$  then  
12. |  $ret \leftarrow \text{true}$   
13. else  
14. | print  $count$   
15. |  $ret \leftarrow \text{false}$   
16. end  
17. return  $ret$ 
```



有向图的连通性（了解）

- 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。
- 前面描述的深度、广度优先遍历都只是计算了单向路径。



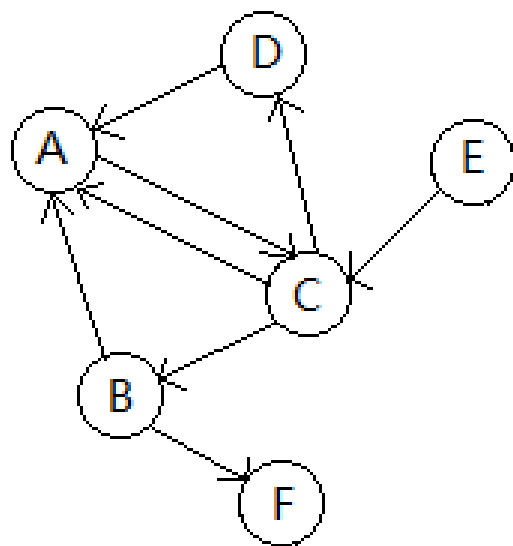
有向图的连通性

依然可利用有向图的深度优先遍历DFS，通过以下算法获得：

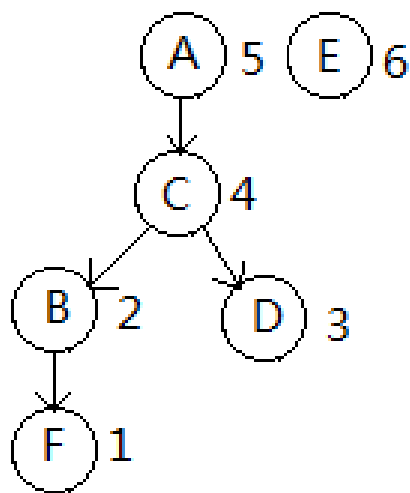
1. 对有向图**G**进行深度优先遍历，按照遍历中回退顶点的次序给每个顶点进行编号。最先回退的顶点的编号为**1**，其它顶点的编号按回退先后逐次增大**1**。
2. 将有向图**G**的所有有向边反向，构造新的有向图**G_r**。
3. 选取未访问顶点中编号最大的顶点，以该顶点为起始点在有向图**G_r**上进行深度优先遍历。如果没有访问到所有的顶点，再次返回**3**，反复如此，直至所有的顶点都被访问到。



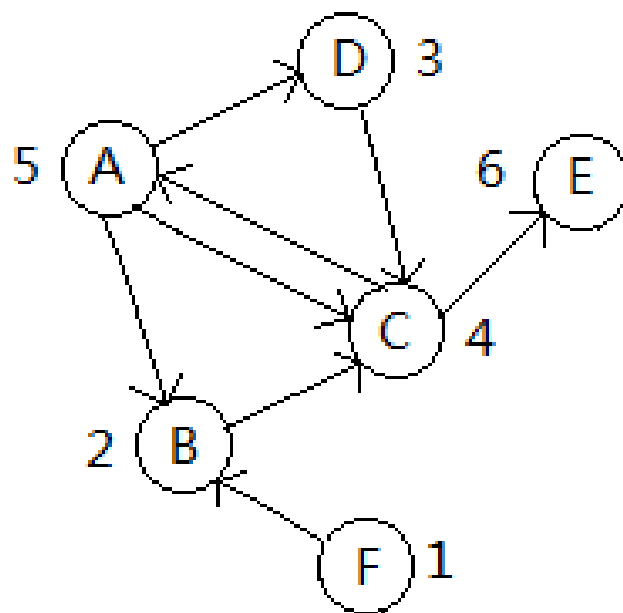
有向图的连通性



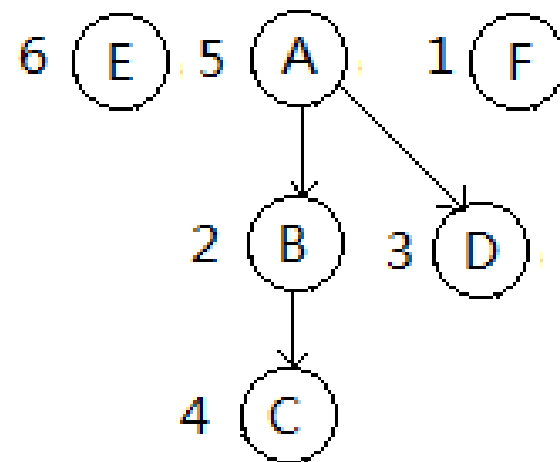
(a) G19



(b) G19的深度优先遍历



(c) G19的边逆向得Gr

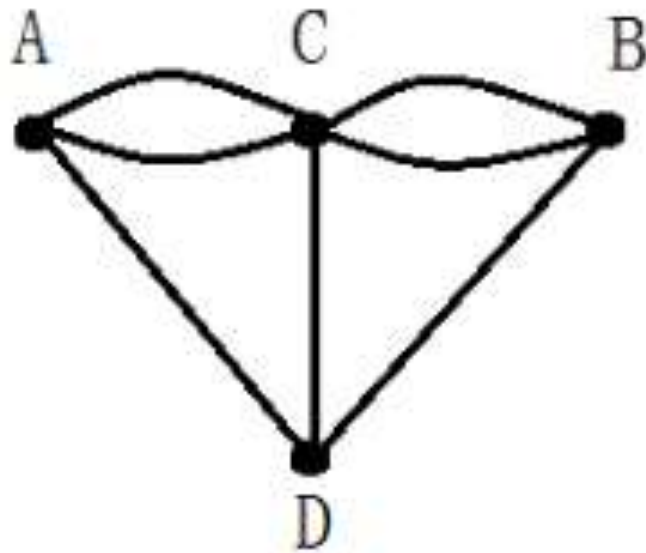


(d) Gr的深度优先遍历



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。（相似问题：**哈密顿路径**）

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**），

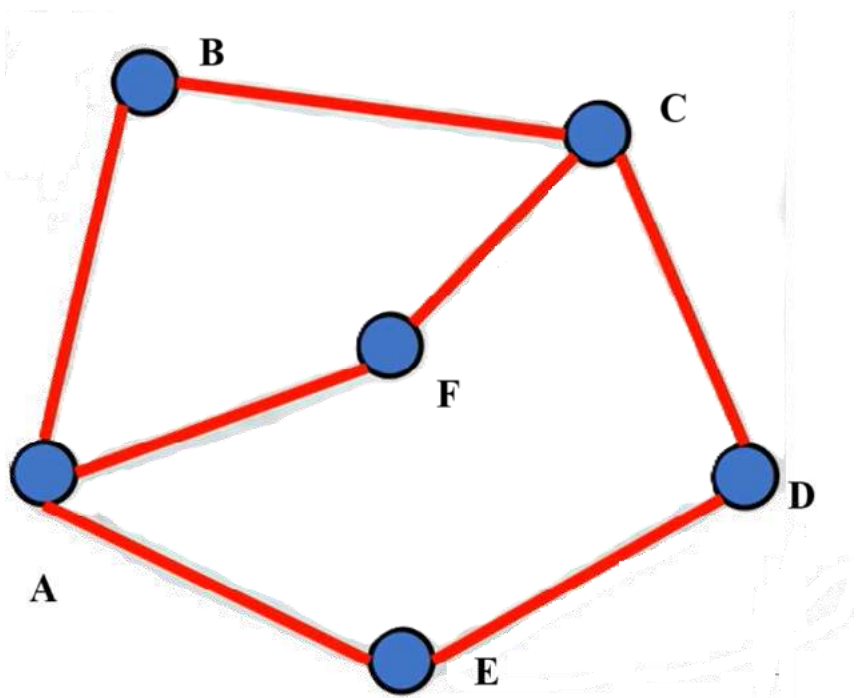
半欧拉图：具有欧拉路径但不具有欧拉回路的图称半欧拉图。

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。



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。（相近的一个问题：**哈密顿路径**）



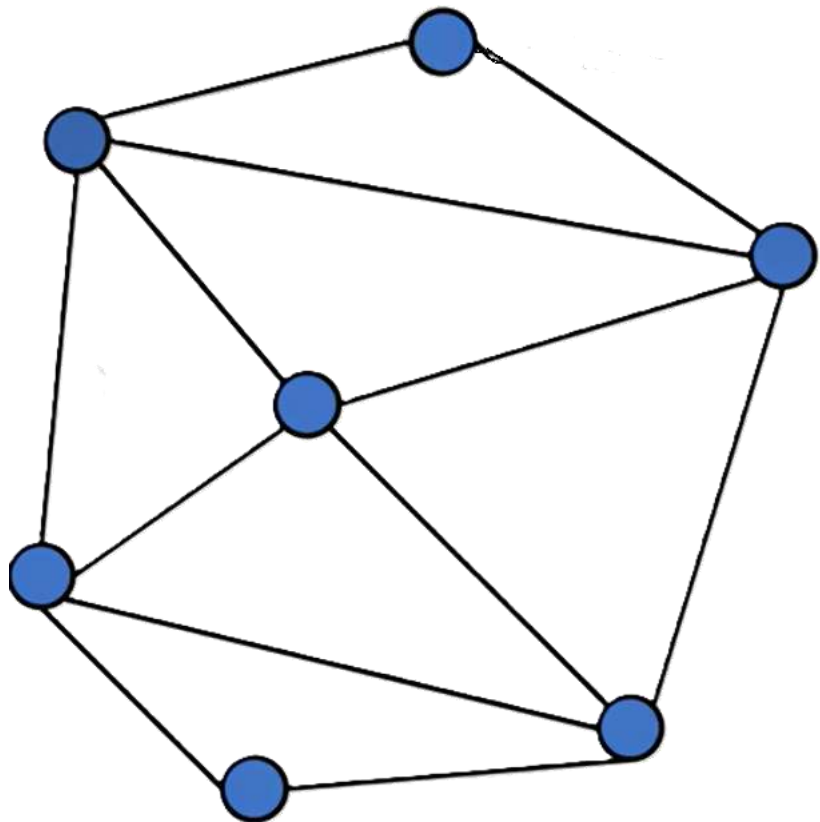
欧拉路径可以有 multiple 条

图可以说无向图，也可以是有向图



相关术语

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。



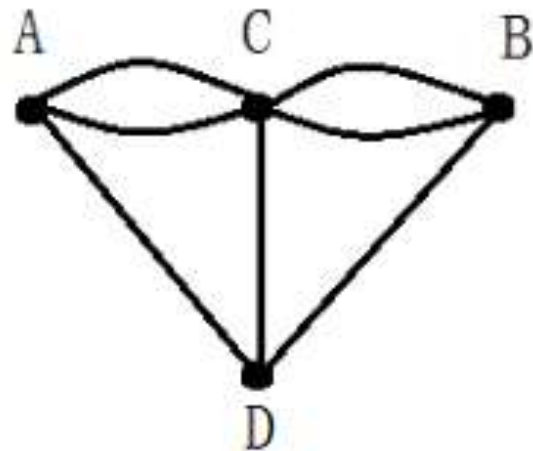
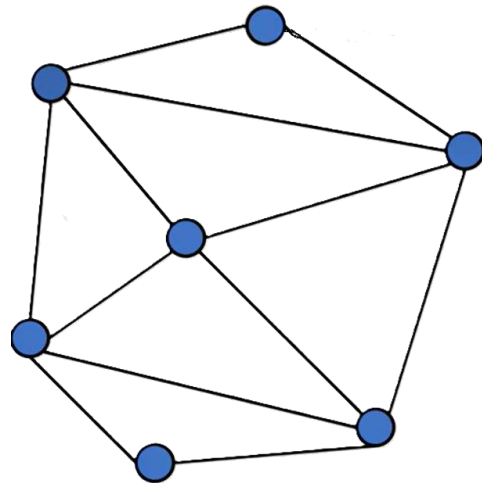
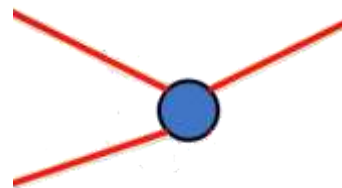
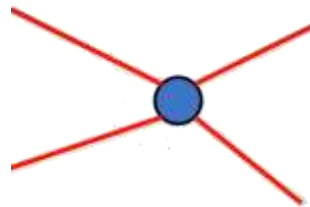
欧拉图： 具有欧拉回路的图称欧拉图(简称E图)

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



欧拉定理

- 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）。
- 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，如果度为奇数的顶点超过了2个，则欧拉路径是不存在的。



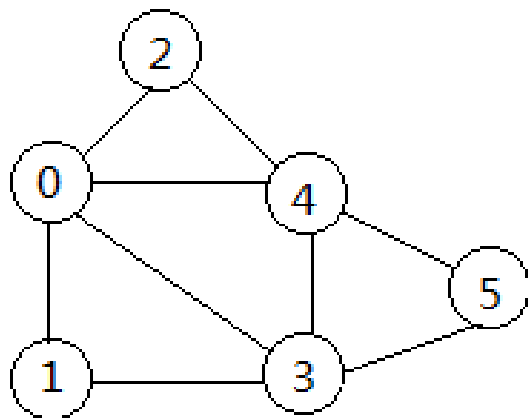


欧拉回路求解方法(选学)

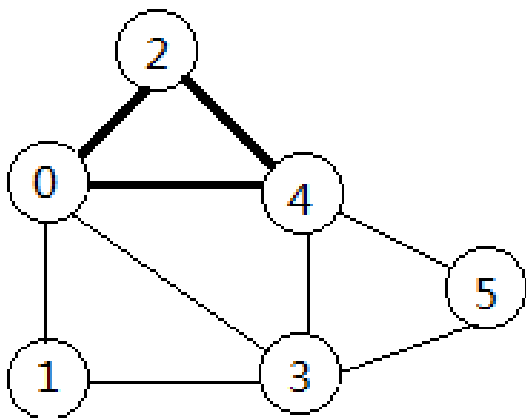
1. 任选一个顶点 v ，从该顶点出发开始深度优先搜索，搜索路径上都是由未访问过的边构成，搜索中访问这些边，最后直到回到顶点 v 且 v 没有尚未被访问的边，此时便得到了一个回路，此回路为当前结果回路。
2. 在搜索路径上另外找一个尚有未访问边的顶点，继续如上操作，找到另外一个回路，将该回路拼接在当前结果回路上，形成一个大的、新的当前结果回路。
3. 如果在当前结果回路中，还有中间某结点有尚未访问的边，回到2)；如果没有任何中间顶点尚余未访问的边，访问结束，当前结果回路即欧拉回路。



欧拉回路求解方法(选学)



(a) G20



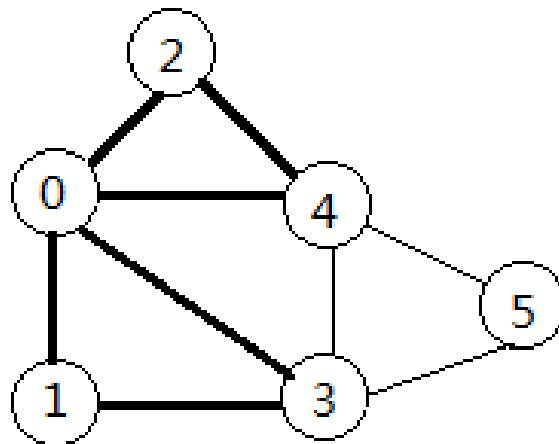
(b)

第1个回路: 2->0->4->2

当前结果回路: **2->0->4->2**

第2个回路: 0->1->3->0

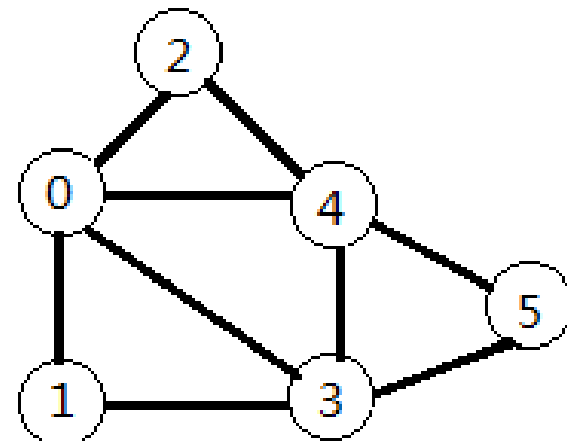
当前结果回路: **2->0->1->3->0->4->2**



(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**



(d)



欧拉回路求解算法(选学)

算法7-17: 从给定点出发获得一条回路 $\text{GetCircuit}(\text{graph}, \text{start})$

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. $\text{new_node} \leftarrow \text{new EulerNode}(\text{start}, \text{NIL})$ //从 start 顶点开始, 构造回路的第一个结点
2. $\text{circuit} \leftarrow \text{new CircPtrNode}$
3. $\text{circuit.first} \leftarrow \text{new_node}$
4. $\text{circuit.last} \leftarrow \text{new_node}$
5. $p \leftarrow \text{graph.ver_list}[\text{start}].\text{adj}$
6. $\text{head} \leftarrow \text{start}$



欧拉回路求解算法(选学)

```
7. while  $p \neq \text{NIL}$  且  $p.\text{dest} \neq \text{start}$  do  
8. |  $\text{tail} \leftarrow p.\text{dest}$   
9. |  $\text{RmoveEdge}(\text{graph}, \text{head}, \text{tail})$   
10. |  $\text{RmoveEdge}(\text{graph}, \text{tail}, \text{head})$   
11. |  $\text{new\_node} \leftarrow \text{new EulerNode}(\text{tail}, \text{NIL})$   
12. |  $\text{circuit.last.next} \leftarrow \text{new\_node}$   
13. |  $\text{circuit.last} \leftarrow \text{circuit.last.next}$   
14. |  $p \leftarrow \text{graph.ver\_list}[\text{tail}].\text{adj}$   
15. |  $\text{head} \leftarrow \text{tail}$   
16. end  
17. return  $\text{circuit}$ 
```



欧拉回路求解算法(选学)

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  $v \leftarrow 0$  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. |  $p \leftarrow graph.ver\_list[v].adj$ 
3. |  $degree[v] \leftarrow 0$ 
4. | while  $p \neq NIL$  do
5. | |  $degree[v] \leftarrow degree[v]+1$ 
6. | |  $p \leftarrow p.next$ 
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
```



欧拉回路求解算法(选学)

```
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
```

访问到每条边一次, 顶点共 m 次,
时间复杂度 $O(m)$



六度空间理论(选学)

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 $\text{SixDegreesOfSeparation}(\text{graph}, v)$

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** $v \leftarrow 0$ **to** $\text{graph}.n_verts-1$ **do** //初始化各顶点的访问标志为未访问
2. | $visited[v] \leftarrow false$
3. **end**
4. $count \leftarrow 0$
5. $\text{InitQueue}(\text{ver_queue})$
6. $\text{InitQueue}(\text{level_queue})$
7. $\text{EnQueue}(\text{ver_queue}, v)$
8. $\text{EnQueue}(\text{level_queue}, 0)$



六度空间理论的验证算法

```
9.  while IsEmpty(ver_queue)=false do
10. | cur_ver  $\leftarrow$  DeQueue(ver_queue)
11. | cur_level  $\leftarrow$  DeQueue(level_queue)
12. | if cur_level  $\leq$  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]  $\leftarrow$  true
15. | | | count  $\leftarrow$  count + 1
16. | | | p  $\leftarrow$  graph.ver_list[cur_ver].adj //向cur_ver的下一层搜索
17. | | | while p  $\neq$  NIL do
18. | | | | if visited[p.dest]=false then
19. | | | | | EnQueue(ver_queue, p.dest)
20. | | | | | EnQueue(level_queue, cur_level+1)
21. | | | | end
```



六度空间理论的验证算法

```
22. | | | |  $p \leftarrow p.next$ 
23. | | | end
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 $O(n+m)$