



Data Structure & Algorithm Analysis

Graph

Zibin Zheng (郑子彬)

School of Data and Computer Science , SYSU

<http://www.inpluslab.com>

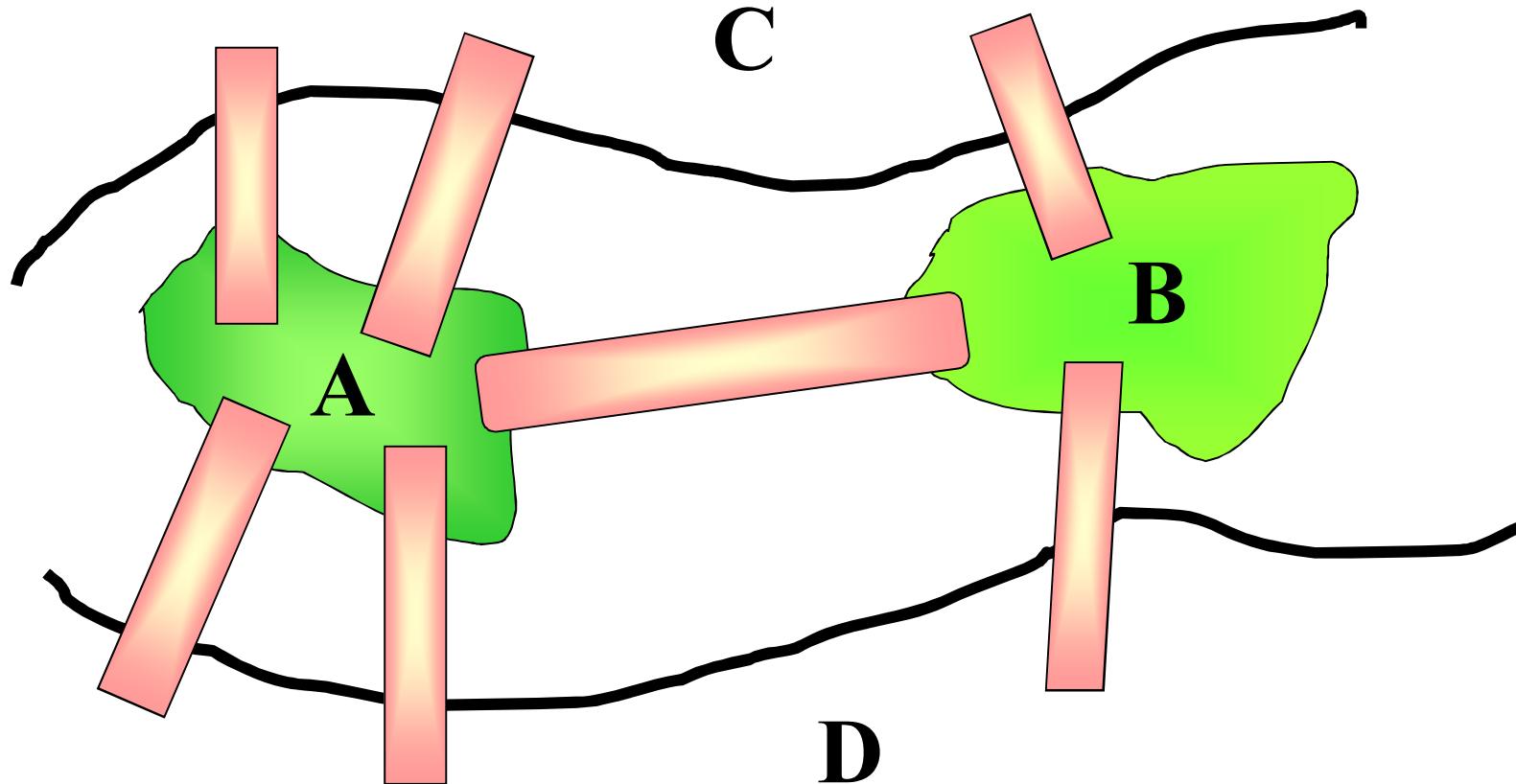
课程主页: <http://inpluslab.sysu.edu.cn/dsa2016/>

图论——欧拉



欧拉1707年出生在瑞士的巴塞尔城，19岁开始发表论文，直到76岁。几乎每一个数学领域都可以看到欧拉的名字，从初等几何的欧拉线，多面体的欧拉定理，立体解析几何的欧拉变换公式，四次方程的欧拉解法到数论中的欧拉函数，微分方程的欧拉方程，级数论的欧拉常数，变分学的欧拉方程，复变函数的欧拉公式等等。据统计他一生共写下了886本书籍和论文，其中分析、代数、数论占40%，几何占18%，物理和力学占28%，天文学占11%，弹道学、航海学、建筑学等占3%。欧拉对著名的哥尼斯堡七桥问题的解答开创了图论的研究。

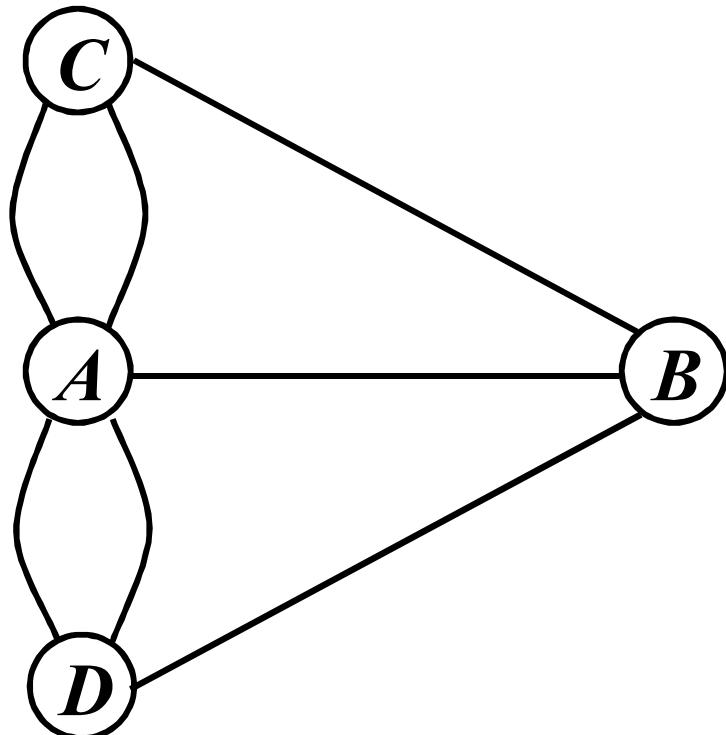
哥尼斯堡七桥问题



能否从某个地方出发，穿过所有的桥仅一次
后再回到出发点？

哥尼斯堡七桥问题

七桥问题的图模型



欧拉回路的判定规则：

1. 如果通奇数桥的地方多于两个，则不存在欧拉回路；
2. 如果只有两个地方通奇数桥，可以从这两个地方之一出发，找到欧拉回路；
3. 如果没有一个地方是通奇数桥的，则无论从哪里出发，都能找到欧拉回路。

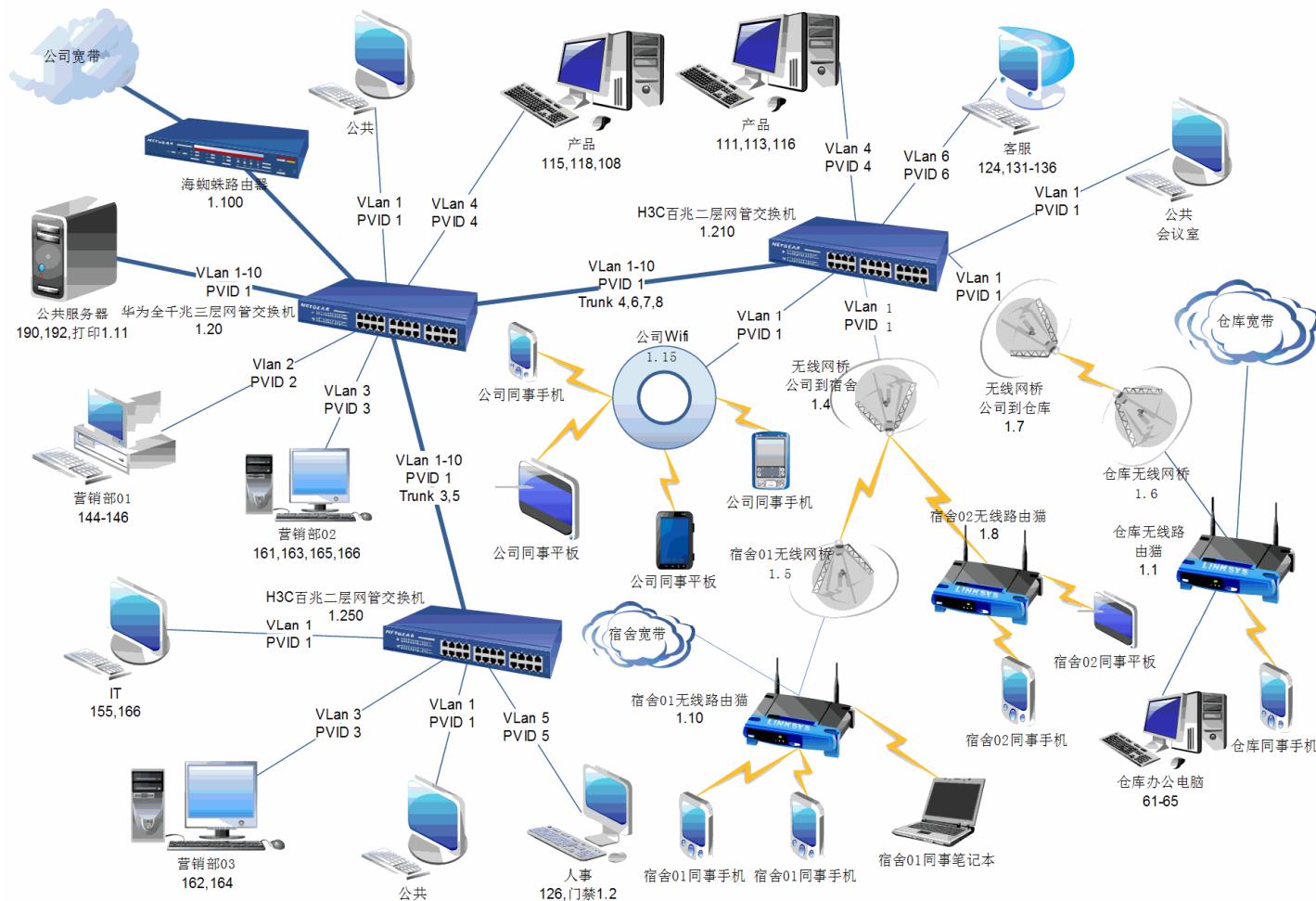
Graph示例

社交网络图谱



Graph示例

计算机网络结构



图的基本概念

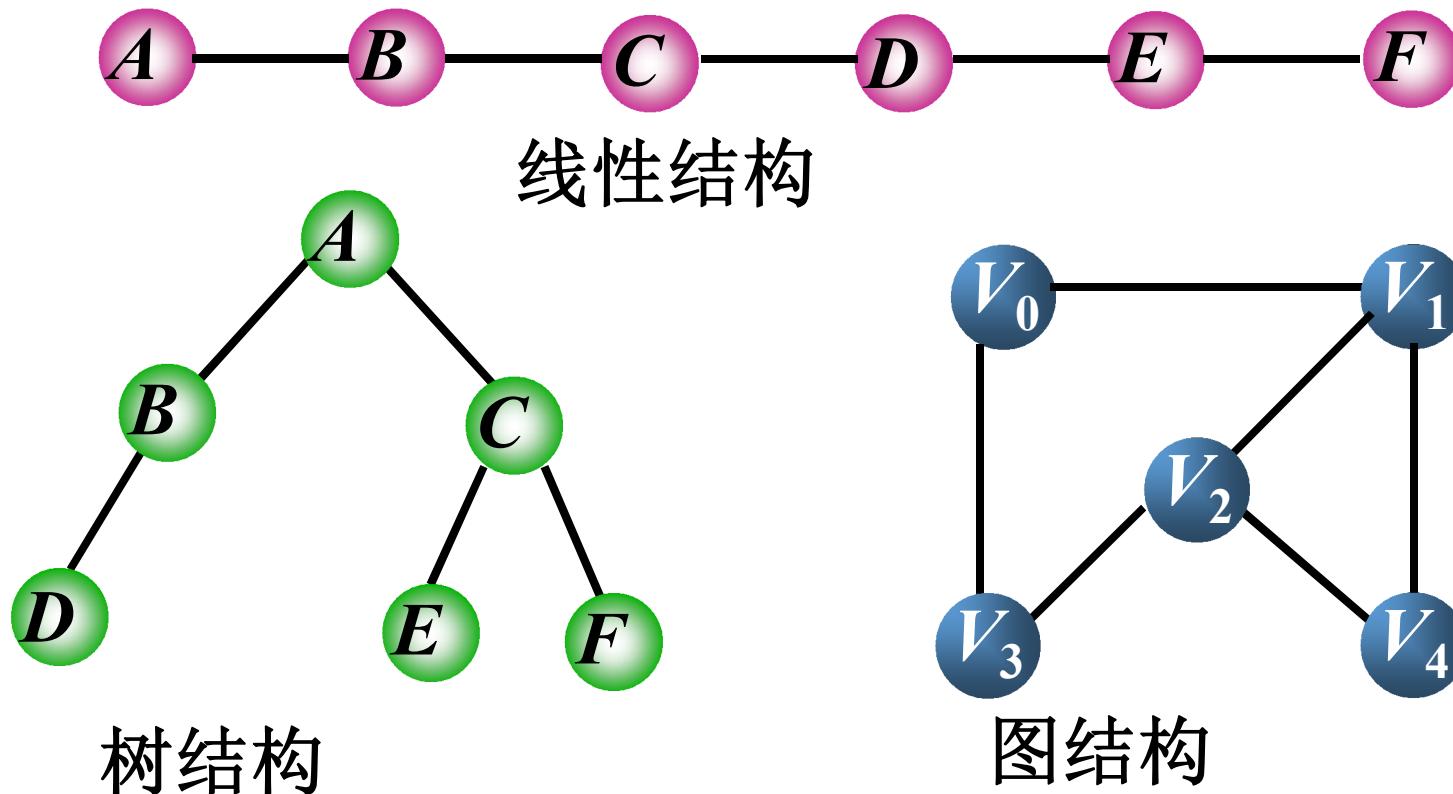
图是由顶点的有穷非空集合和顶点之间边的集合组成，通常表示为：

$$G=(V, E)$$

其中： G 表示一个图， V 是图 G 中顶点的集合， E 是图 G 中顶点之间边的集合。

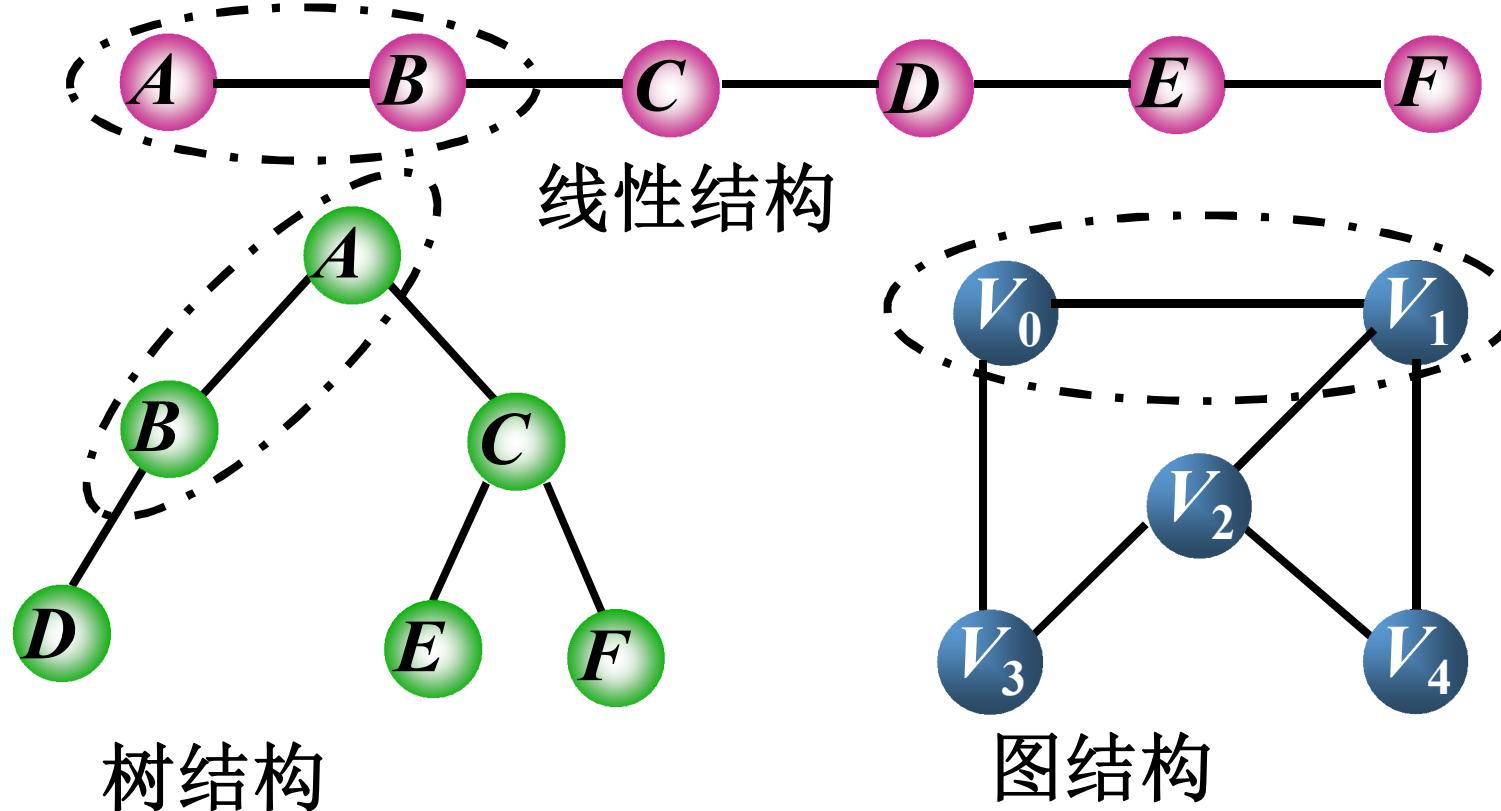
在线性表中，元素个数可以为零，称为空表；
在树中，结点个数可以为零，称为空树；
在图中，顶点个数不能为零，但可以没有边。

不同结构中逻辑关系的对比



在线性结构中，数据元素之间仅具有线性关系；
在树结构中，结点之间具有层次关系；
在图结构中，任意两个顶点之间都可能有关系。

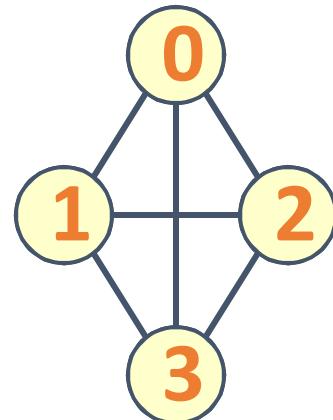
不同结构中逻辑关系的对比



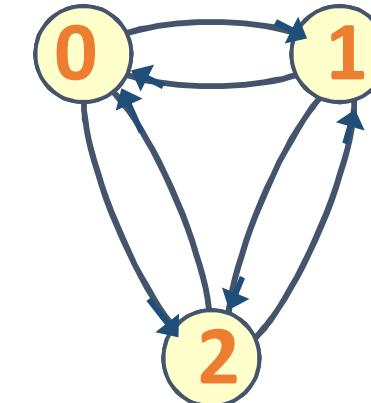
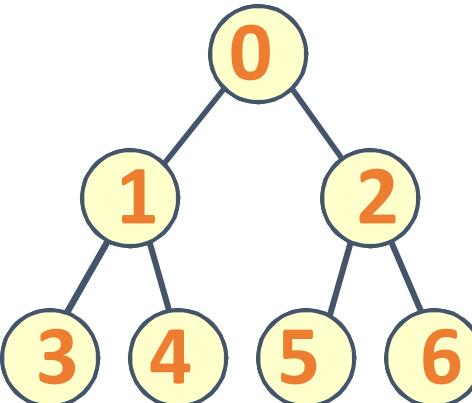
在线性结构中，元素之间的关系为前驱和后继；
在树结构中，结点之间的关系为双亲和孩子；
在图结构中，顶点之间的关系为邻接。

图的基本概念

- 若代表一条边的顶点序偶是无序的(即该边无方向)，则称此图为**无向图**。
- 若代表一条边的顶点序偶是有序的(即边有方向)，则称此图为**有向图**。
- 通常用n表示图中顶点的数目，用e表示边或弧的数目。无向图中e的取值范围是从0到 $n(n - 1)/2$ ，有向图中e的取值范围是从0到 $n(n - 1)$



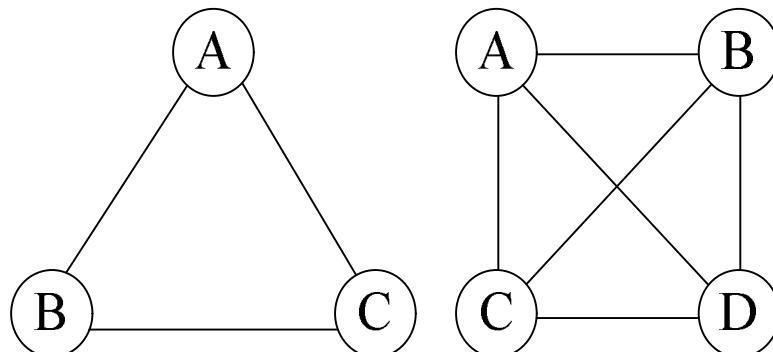
无向图



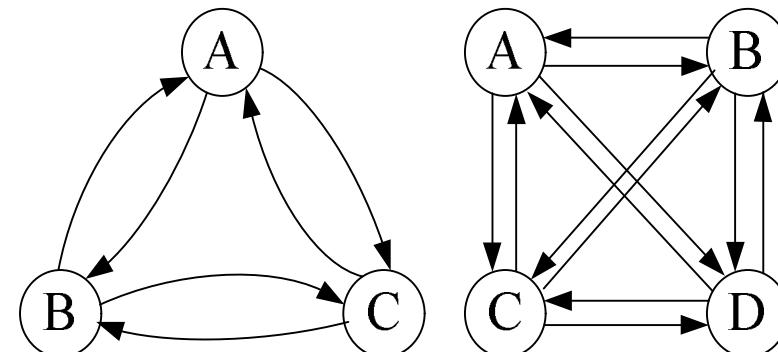
有向图

图的基本概念

- 边数相对较少的图称为稀疏图(sparse graph)，边数相对较多的图称为稠密图(dense graph)
- 任何两顶点间都有边相关联的图称为完全图(complete graph)，完全图显然具有最大的边数。
- 若有 n 个顶点的无向图有 $n(n-1)/2$ 条边，则此图为完全无向图。有 n 个顶点的有向图有 $n(n-1)$ 条边，则此图为完全有向图。



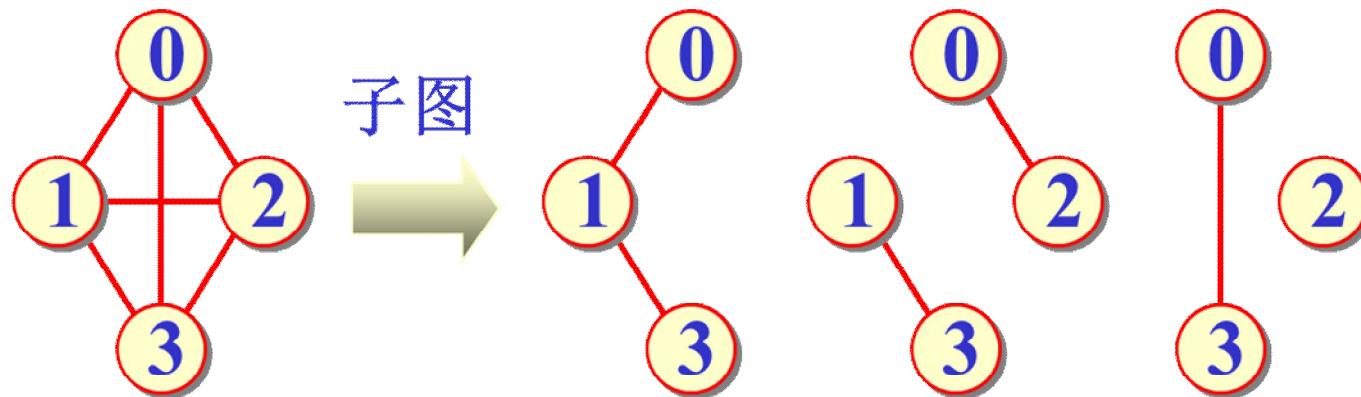
(a) 无向完全图



(b) 有向完全图

图的基本概念

- 邻接顶点 如果 (u, v) 是 $E(G)$ 中的一条边，则称 u 与 v 互为邻接顶点。
- 子图 设有两个图 $G = (V, E)$ 和 $G' = (V', E')$ 。若 $V' \subseteq V$ 且 $E' \subseteq E$, 则称 图 G' 是 图 G 的子图。



- 权 某些图的边具有与它相关的数, 称之为权。这种带权图叫做网络 (network)。

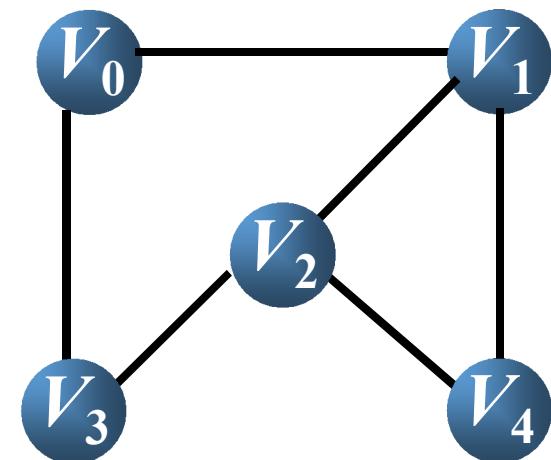
图的基本概念

- 顶点的度 一个顶点 v 的度是与它相关联的边的条数。记作 $TD(v)$ 。在有向图中, 顶点的度等于该顶点的入度与出度之和。
- 顶点 v 的入度是以 v 为终点的有向边的条数, 记作 $ID(v)$; 顶点 v 的出度是以 v 为始点的有向边的条数, 记作 $OD(v)$ 。

路径：在无向图 $G=(V, E)$ 中，从顶点 v_p 到顶点 v_q 之间的**路径**是一个顶点序列 ($v_p=v_{i0}, v_{i1}, v_{i2}, \dots, v_{im}=v_q$)，其中， $(v_{ij-1}, v_{ij}) \in E$ ($1 \leq j \leq m$)。若 G 是有向图，则路径也是有方向的，顶点序列满足 $\langle v_{ij-1}, v_{ij} \rangle \in E$ 。

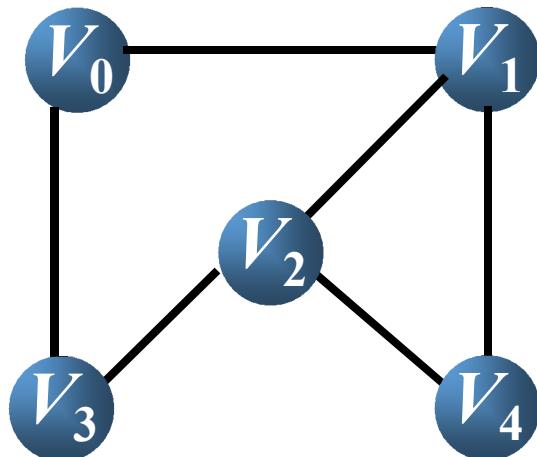
V_0 到 V_3 的路径：
 $V_0 V_3$
 $V_0 V_1 V_2 V_3$
 $V_0 V_1 V_4 V_2 V_3$

一般情况下，图中的路径不惟一



路径长度：

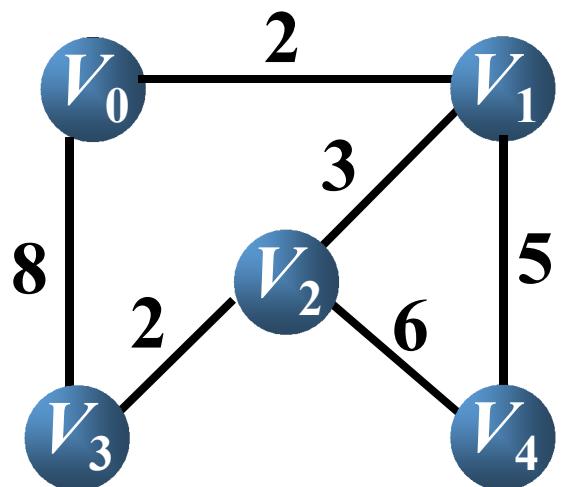
{ 非带权图——路径上边的个数
带权图——路径上各边的权之和



V_0V_3 : 长度为1
 $V_0V_1V_2V_3$: 长度为3
 $V_0V_1V_4V_2V_3$: 长度为4

路径长度：

{ 非带权图——路径上边的个数
带权图——路径上各边的权之和



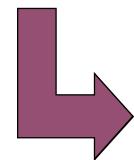
$V_0 V_3$: 长度为8
 $V_0 V_1 V_2 V_3$: 长度为7
 $V_0 V_1 V_4 V_2 V_3$: 长度为15

图的基本概念

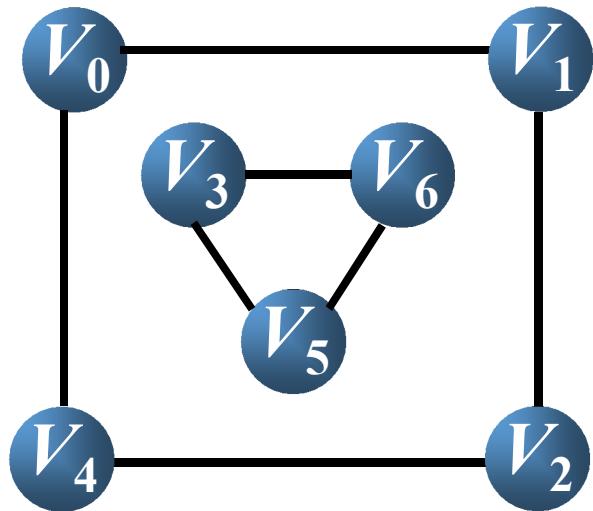
- **简单路径** 若路径上各顶点 v_1, v_2, \dots, v_m 均不互相重复, 则称这样的路径为简单路径。
- **回路** 若路径上第一个顶点 v_1 与最后一个顶点 v_m 重合, 则称这样的路径为回路或环。
- **简单回路（简单环）** : 除了第一个顶点和最后一个顶点外，其余顶点不重复出现的回路。

连通图：在无向图中，如果从一个顶点 v_i 到另一个顶点 $v_j(i \neq j)$ 有路径，则称顶点 v_i 和 v_j 是连通的。如果图中任意两个顶点都是连通的，则称该图是连通图。

连通分量：非连通图的极大连通子图称为连通分量。

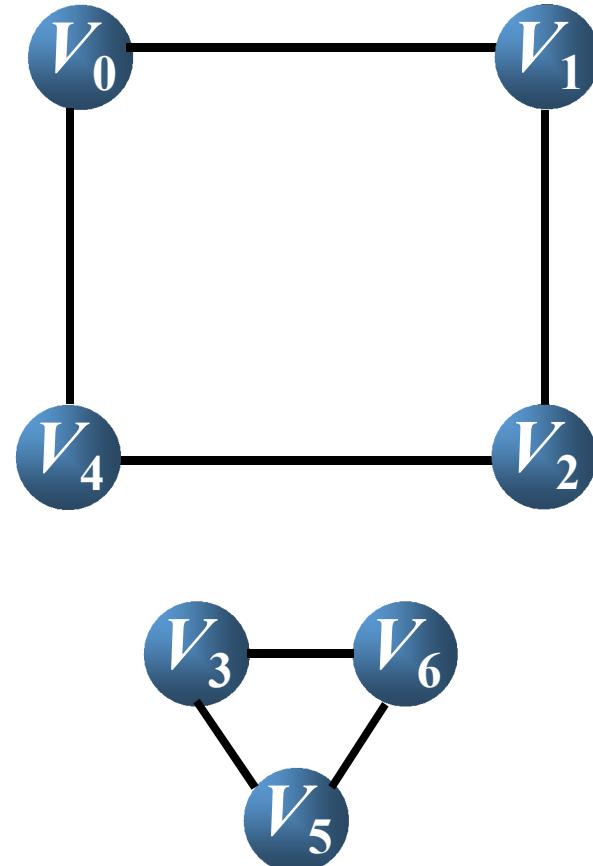


- 1. 含有极大**顶点数**；
- 2. 依附于这些顶点的所有**边**。



连通分量1

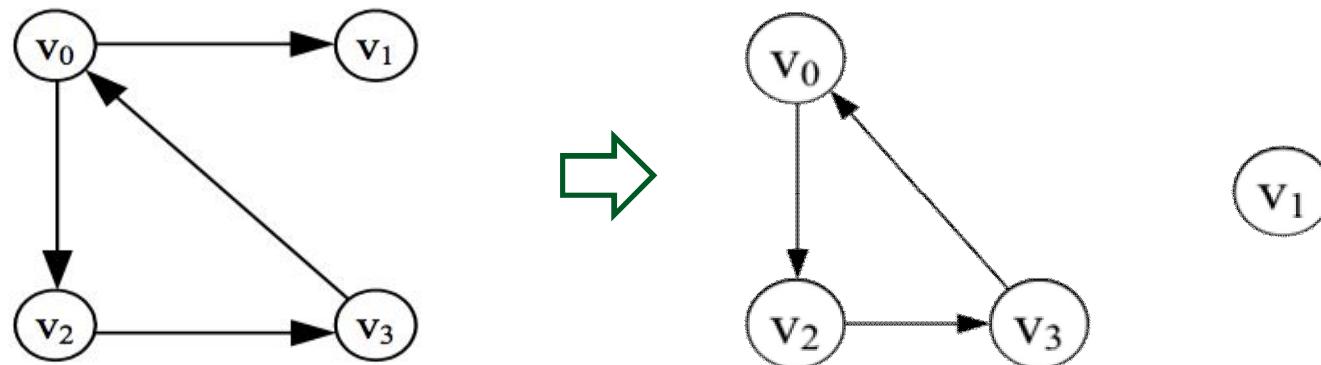
连通分量2



连通分量是对无向图的一种划分

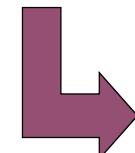
图的基本概念

- 对于有向图 $G = \langle V, E \rangle$ ，若 G 中任意两个顶点 v_i 和 v_j ($v_i \neq v_j$)，都有一条从 v_i 到 v_j 的有向路径，同时还有一条从 v_j 到 v_i 的有向路径，则称有向图 G 是强连通图。有向图强连通的极大子图称为该有向图的**强连通分支**或者**强连通分量**。



有向图 G_2 的两个强连通分量

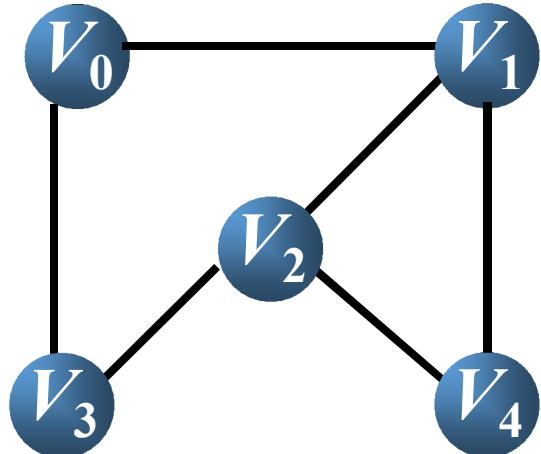
生成树： n 个顶点的连通图 G 的生成树是包含 G 中**全部顶点**的一个极小连通子图。

 含有 $n-1$ 条边 $\left\{ \begin{array}{l} \text{多——构成回路} \\ \text{少——不连通} \end{array} \right.$

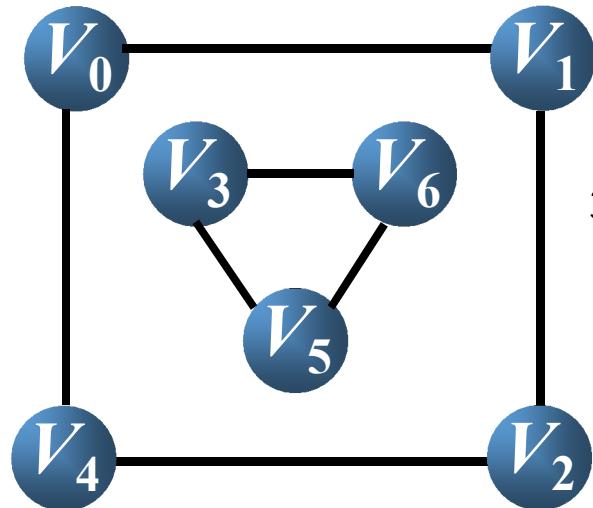
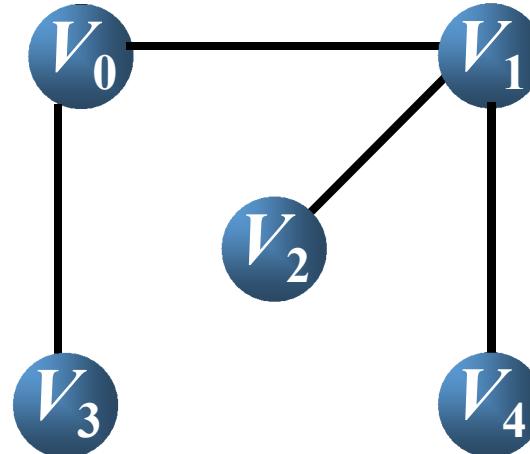
生成森林：在非连通图中，由每个连通分量都可以得到一棵生成树，这些连通分量的生成树就组成了一个非连通图的**生成森林**。

极大连通子图 vs 极小连通子图

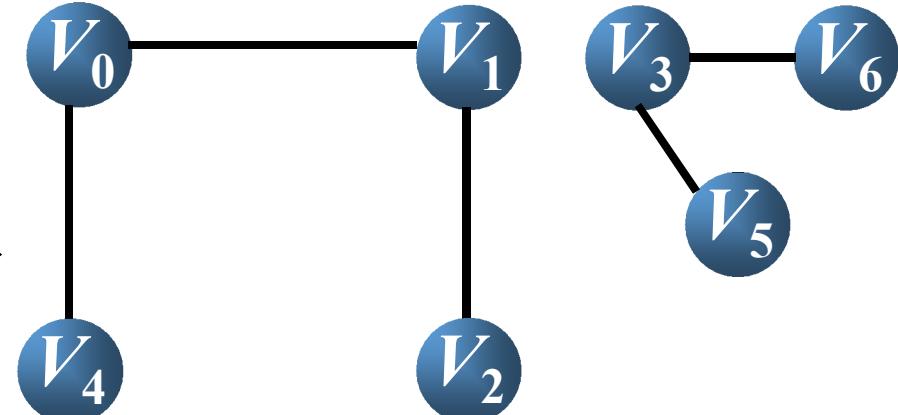
- 极大连通子图可以存在于无向图中,也可以存在于有向图中
- 连通的无向图、只有一个极大连通子图,即它本身,因为不存在另一个连通的子图包含的点和边比它本身还要多
- 不连通的无向图. 可以拆分为若干个连通的无向图,如果我们在拆分时注意把能连通的点边都放在一个连通子图中,使这个连通子图足够大,以至于再多包含一个点或边它就变成不连通的了,我们称这个连通子图为极大连通子图,也叫连通分量.
- 极小连通子图只存在于连通的无向图中,该图中只有一个连通分量(极大连通子图),之所以说它极小,是因为极小连通子图只要求包含图中所有顶点及其比顶点数量少一个的边(且不能成环),也就是说如果给极小连通子图任意两个顶点间加入一条边,则必有环.
- 这里的极大和极小不是指一个意思,容易弄混,极大连通子图是讨论连通分量的,极小连通子图是讨论生成树的.



生成树
→

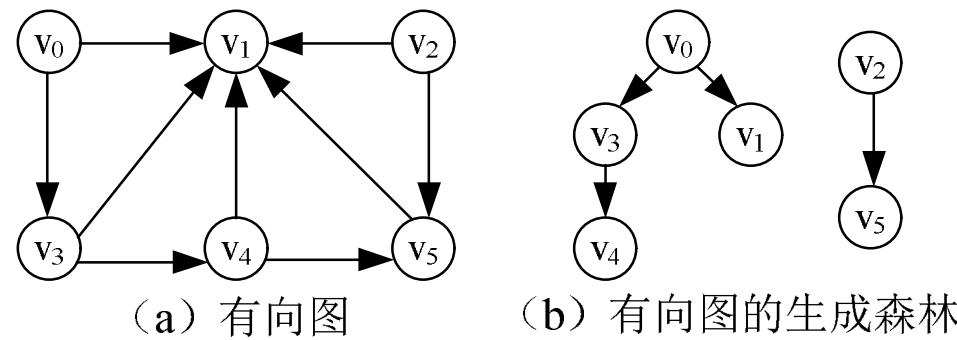


生成森林
→



图的基本概念

- 如果一个有向图只有一个顶点的入度为0，其余顶点的入度均为1，则称为有向树。一个有向图的生成森林由若干棵有向树组成，这些树的并集包含了原图所有顶点，各有向树的弧不相交。图7.9就是有向图生成森林的示例。



图的抽象数据类型

```
class Graph {  
    public:  
        Graph ();  
        void InsertVertex ( Type & vertex );           // 插入一个顶点  
        void InsertEdge ( int v1, int v2, int weight ); // 插入一条边  
        void RemoveVertex ( int v );                  // 删除一个顶点  
        void RemoveEdge ( int v1, int v2 );            // 删除一条边  
        int IsEmpty ( );                            // 判断图是否为空  
        Type GetWeight ( int v1, int v2 );           // 获取边的权重值  
        int GetFirstNeighbor ( int v );              // 获取第一个邻接顶点  
        int GetNextNeighbor ( int v1, int v2 );        // 获取下一个邻接顶点  
}
```

图的存储表示

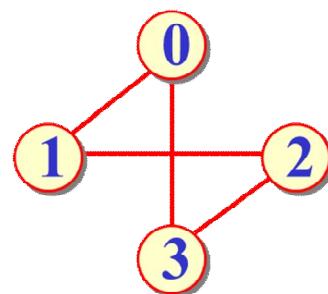
- **邻接矩阵 (Adjacency Matrix)**
- 在图的邻接矩阵表示中，有一个记录各个顶点信息的**顶点表**，还有一个表示各个顶点之间关系的**邻接矩阵**。
- 设图 $A = (V, E)$ 是一个有 n 个顶点的图，图的邻接矩阵是一个二维数组 $A.edge[n][n]$ ，定义：

$$A.Edge[i][j] = \begin{cases} 1, & \text{如果 } \langle i, j \rangle \in E \text{ 或者 } (i, j) \in E \\ 0, & \text{否则} \end{cases}$$

对于 n 个顶点的图，相邻矩阵的空间代价都为 $O(n^2)$ ，与边数无关。

图的存储表示

- 无向图的邻接矩阵是对称的;
- 有向图的邻接矩阵可能是不对称的。

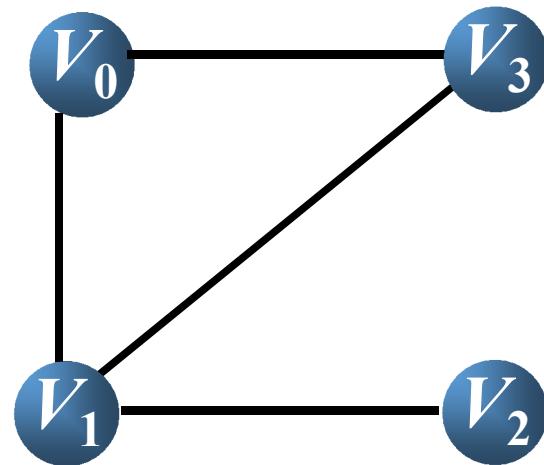


$$A.\text{edge} = \begin{bmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{bmatrix}$$



$$A.\text{edge} = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}$$

无向图的邻接矩阵



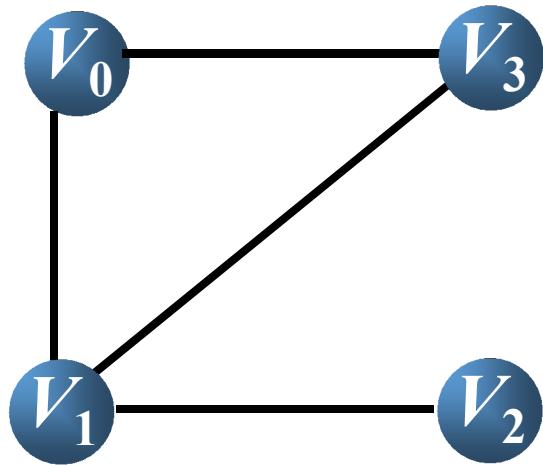
vertex= $\begin{array}{c} V_0 \\ V_1 \\ V_2 \\ V_3 \end{array}$

arc= $\begin{array}{ccccc} & V_0 & V_1 & V_2 & V_3 \\ V_0 & \left[\begin{array}{cccc} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \end{array} \right] \\ V_1 \\ V_2 \\ V_3 \end{array}$

② 无向图的邻接矩阵的特点？

主对角线为 0 且一定是对称矩阵。

无向图的邻接矩阵



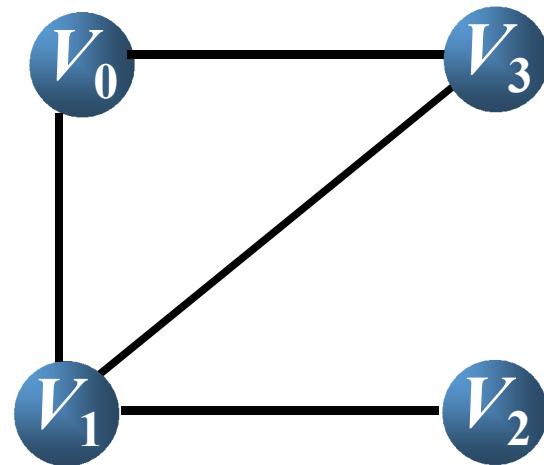
vertex = $\begin{array}{c} V_0 \\ V_1 \\ V_2 \\ V_3 \end{array}$

arc = $\begin{array}{ccccc} & V_0 & V_1 & V_2 & V_3 \\ V_0 & 0 & 1 & 0 & 1 \\ V_1 & 1 & 0 & 1 & 1 \\ V_2 & 0 & 1 & 0 & 0 \\ V_3 & 1 & 1 & 0 & 0 \end{array}$

① 如何求顶点*i*的度？

邻接矩阵的第*i*行（或第*i*列）非零元素的个数。

无向图的邻接矩阵



vertex =

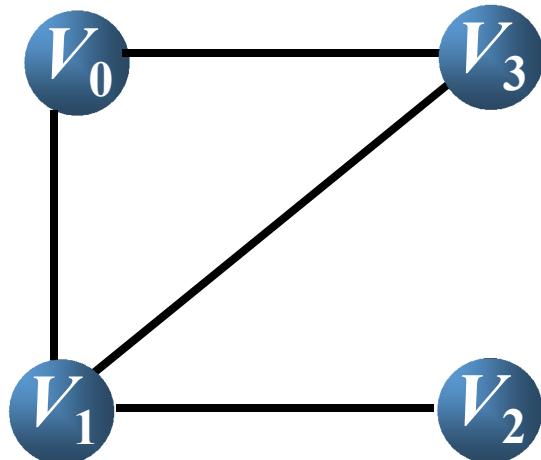
| | | | |
|-------|-------|-------|-------|
| V_0 | V_1 | V_2 | V_3 |
|-------|-------|-------|-------|

arc =
$$\begin{matrix} & \begin{matrix} V_0 & V_1 & V_2 & V_3 \end{matrix} \\ \begin{matrix} V_0 \\ V_1 \\ V_2 \\ V_3 \end{matrix} & \left[\begin{matrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \end{matrix} \right] \end{matrix}$$

② 如何判断顶点 i 和 j 之间是否存在边？

测试邻接矩阵中相应位置的元素 $\text{arc}[i][j]$ 是否为 1。

无向图的邻接矩阵



vertex=

| | | | |
|-------|-------|-------|-------|
| V_0 | V_1 | V_2 | V_3 |
|-------|-------|-------|-------|

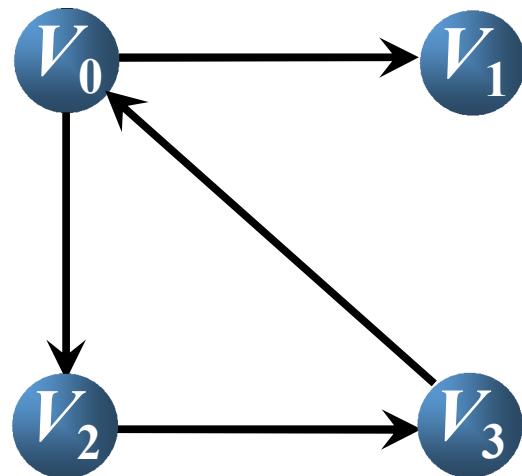
arc=

| | | | | |
|-------|-------|-------|-------|---|
| 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 |

① 如何求顶点 i 的所有邻接点？

将数组中第 i 行元素扫描一遍，若 $\text{arc}[i][j]$ 为 1，则顶点 j 为顶点 i 的邻接点。

有向图的邻接矩阵



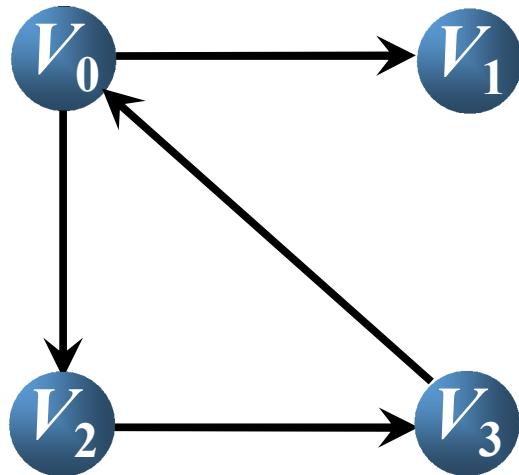
vertex= $\boxed{V_0 \quad V_1 \quad V_2 \quad V_3}$

$$\text{arc=} \begin{matrix} & V_0 & V_1 & V_2 & V_3 \\ V_0 & \left[\begin{array}{cccc} 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{array} \right] \\ V_1 \\ V_2 \\ V_3 \end{matrix}$$

① 有向图的邻接矩阵一定不对称吗？

不一定，例如有向完全图。

有向图的邻接矩阵



vertex=

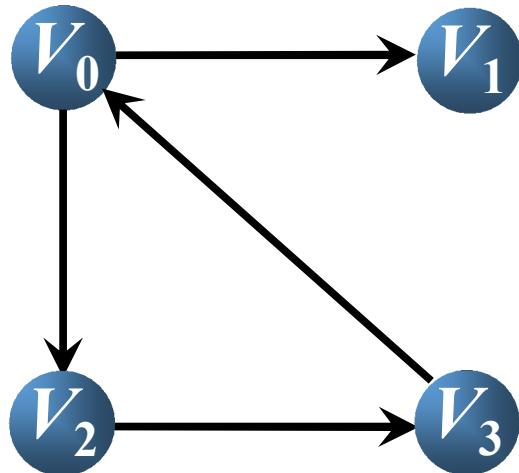
| | | | |
|-------|-------|-------|-------|
| V_0 | V_1 | V_2 | V_3 |
|-------|-------|-------|-------|

arc=
$$\begin{matrix} & V_0 & V_1 & V_2 & V_3 \\ V_0 & \boxed{0} & 1 & 1 & 0 \\ V_1 & 0 & \boxed{0} & 0 & 0 \\ V_2 & 0 & 0 & \boxed{0} & 1 \\ V_3 & 1 & 0 & 0 & 0 \end{matrix}$$

① 如何求顶点 i 的出度？

邻接矩阵的第 i 行元素之和。

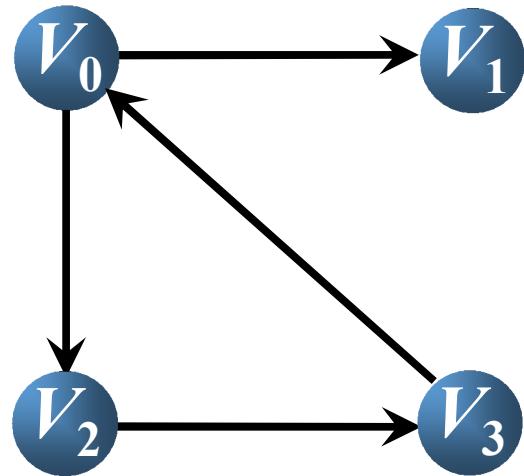
有向图的邻接矩阵


$$\text{vertex} = \begin{array}{c|c|c|c|c} & V_0 & V_1 & V_2 & V_3 \end{array}$$
$$\text{arc} = \left[\begin{array}{ccccc} V_0 & V_1 & V_2 & V_3 \\ \hline V_0 & 0 & 1 & 1 & 0 \\ V_1 & 0 & 0 & 0 & 0 \\ V_2 & 0 & 0 & 0 & 1 \\ V_3 & 1 & 0 & 0 & 0 \end{array} \right]$$

② 如何求顶点 i 的入度？

邻接矩阵的第 i 列元素之和。

有向图的邻接矩阵



vertex=

| | | | |
|-------|-------|-------|-------|
| V_0 | V_1 | V_2 | V_3 |
|-------|-------|-------|-------|

arc=

| | | | | |
|-------|-------|-------|-------|---|
| V_0 | V_1 | V_2 | V_3 | |
| V_0 | 0 | 1 | 1 | 0 |
| V_1 | 0 | 0 | 0 | 0 |
| V_2 | 0 | 0 | 0 | 1 |
| V_3 | 1 | 0 | 0 | 0 |

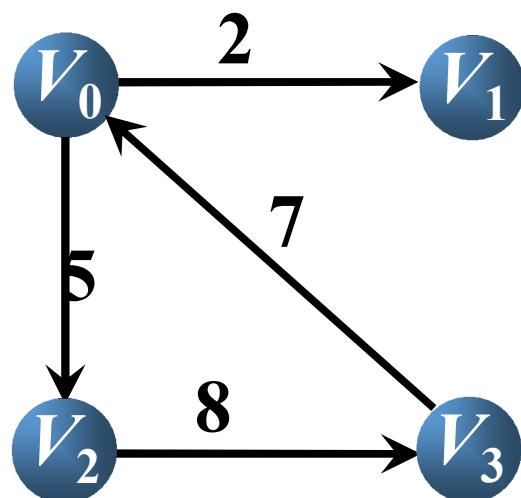
① 如何判断从顶点 i 到顶点 j 是否存在边？

测试邻接矩阵中相应位置的元素 $\text{arc}[i][j]$ 是否为 1。

网图的邻接矩阵

网图的邻接矩阵可定义为：

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



$$\text{arc} = \begin{bmatrix} \infty & 2 & 5 & \infty \\ \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & 8 \\ 7 & \infty & \infty & \infty \end{bmatrix}$$

邻接矩阵的实现

图的邻接矩阵的实现比较容易，定义两个数组分别存储顶点信息(数据元素)和边或弧的信息(数据元素之间的关系)。其存储结构形式定义如下：

```
#define INFINITY MAX_VAL /* 最大值∞ */  
  
/* 根据图的权值类型，分别定义为最大整数或实数 */  
#define MAX_VEX 30 /* 最大顶点数目 */  
  
/* {有向图，无向图，带权有向图，带权无向图} */  
typedef enum {DG, AG, WDG,WAG} GraphKind ;
```

邻接矩阵的实现

```
typedef struct ArcType
{ VexType vex1, vex2 ;          /* 弧或边所依附的两个顶点 */
  ArcValType ArcVal ;           /* 弧或边的权值 */
  ArcInfoType ArcInfo ;         /* 弧或边的其它信息 */
}ArcType; /* 弧或边的结构定义 */
```

```
typedef struct
{ GraphKind kind ;             /* 图的种类标志 */
  int vexnum , arcnum ;        /* 图的当前顶点数和弧数 */
  VexType vexs[MAX_VEX] ;/* 顶点向量 */
  AdjType adj[MAX_VEX][MAX_VEX];
}MGraph; /* 图的结构定义 */
```

利用上述定义的数据结构，可以方便地实现图的各种操作。

邻接矩阵的实现

图的顶点定位

图的顶点定位操作实际上是确定一个顶点在vexs数组中的位置(下标) , 其过程完全等同于在顺序存储的线性表中查找一个数据元素。

算法实现 :

```
int LocateVex(MGraph *G , VexType *vp)
{
    int k ;
    for (k=0 ; k<G->vexnum ; k++)
        if (G->vexs[k]==*vp) return(k) ;
    return(-1) ; /* 图中无此顶点 */
}
```

邻接矩阵的实现

向图中增加顶点

向图中增加一个顶点的操作，类似在顺序存储的线性表的末尾增加一个数据元素。

```
int AddVertex(MGraph *G , VexType *vp)
{ int k , j ;
  if (G->vexnum>=MAX_VEX)
    { printf("Vertex Overflow !\n") ; return(-1) ; }
  if (LocateVex(G , vp)!=-1)
    { printf("Vertex has existed !\n") ; return(-1) ; }
  k=G->vexnum ; G->vexs[G->vexnum++]=*vp ;
  if (G->kind==DG || G->kind==AG)
    for (j=0 ; j<G->vexnum ; j++)
      G->adj[j][k].ArcVal=G->adj[k][j].ArcVal=0 ;
      /* 是不带权的有向图或无向图 */
  else
    for (j=0 ; j<G->vexnum ; j++)
      { G->adj[j][k].ArcVal=INFINITY ;
        G->adj[k][j].ArcVal=INFINITY ;
        /* 是带权的有向图或无向图 */
      }
  return(k) ;
}
```

邻接矩阵的实现

向图中增加一条弧

根据给定的弧或边所依附的顶点，修改邻接矩阵中所对应的数组元素。

```
int AddArc(MGraph *G , ArcType *arc)
{ int k , j;
  k=LocateVex(G , &arc->vex1) ;
  j= LocateVex(G , &arc->vex2) ;
  if (k== -1 || j== -1)
    { printf("Arc's Vertex do not existed !\n");
      return(-1);
    }
  if (G->kind==DG || G->kind==WDG)
    {   G->adj[k][j].ArcVal=arc->ArcVal;
        G->adj[k][j].ArcInfo=arc->ArcInfo ;
        /* 是有向图或带权的有向图*/
    }
  else
    {   G->adj[k][j].ArcVal=arc->ArcVal ;
        G->adj[j][k].ArcVal=arc->ArcVal ;
        G->adj[k][j].ArcInfo=arc->ArcInfo ;
        G->adj[j][k].ArcInfo=arc->ArcInfo ;
        /* 是无向图或带权的无向图,需对称赋值 */
    }
  return(1);
}
```

图的存储表示--邻接表 (Adjacency List)

- 当图中的边数较少时，相邻矩阵就会出现大量的零元素，存储这些零元素将耗费大量的存储空间。对于稀疏图，可以采用邻接表存储法。
- 邻接表(adjacency list)表示法是一种链式存储结构，由一个顺序存储的顶点表和n个链接存储的边表组成。
 - 顶点表目有两个域：顶点数据域和指向此顶点边表指针域
 - 边表把依附于同一个顶点vi的边（即相邻矩阵中同一行的非0元素）组织成一个单链表。边表中的每一个表目都代表一条边，由两个主要的域组成：
 - 与顶点vi邻接的另一顶点的序号
 - 指向边表中下一个边表目的指针

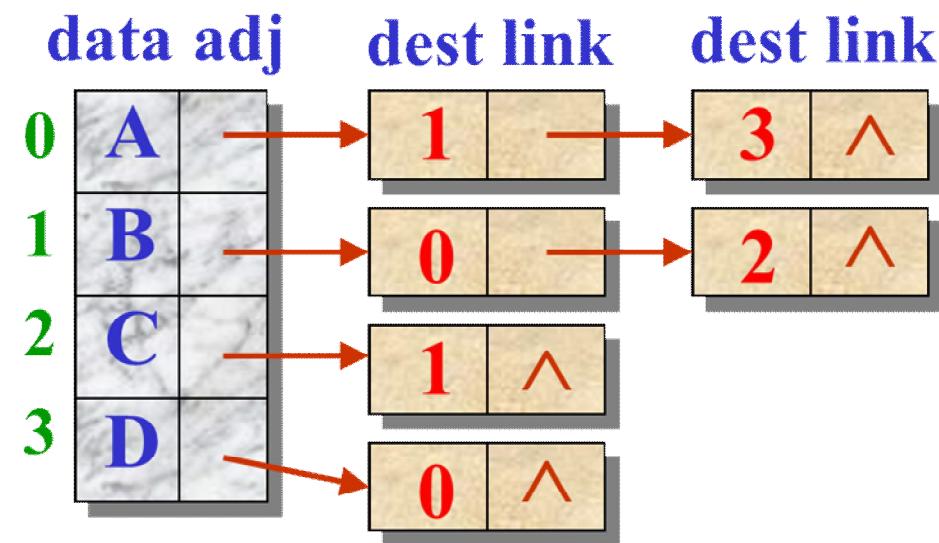
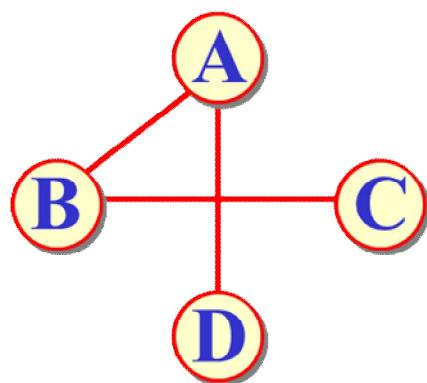
| 顶点结点 | 边（或弧）结点 |
|------|----------|
| data | firstarc |

| | |
|--------|---------|
| adjvex | nextarc |
|--------|---------|

图的存储表示--邻接表 (Adjacency List)

- 邻接表 (Adjacency List)

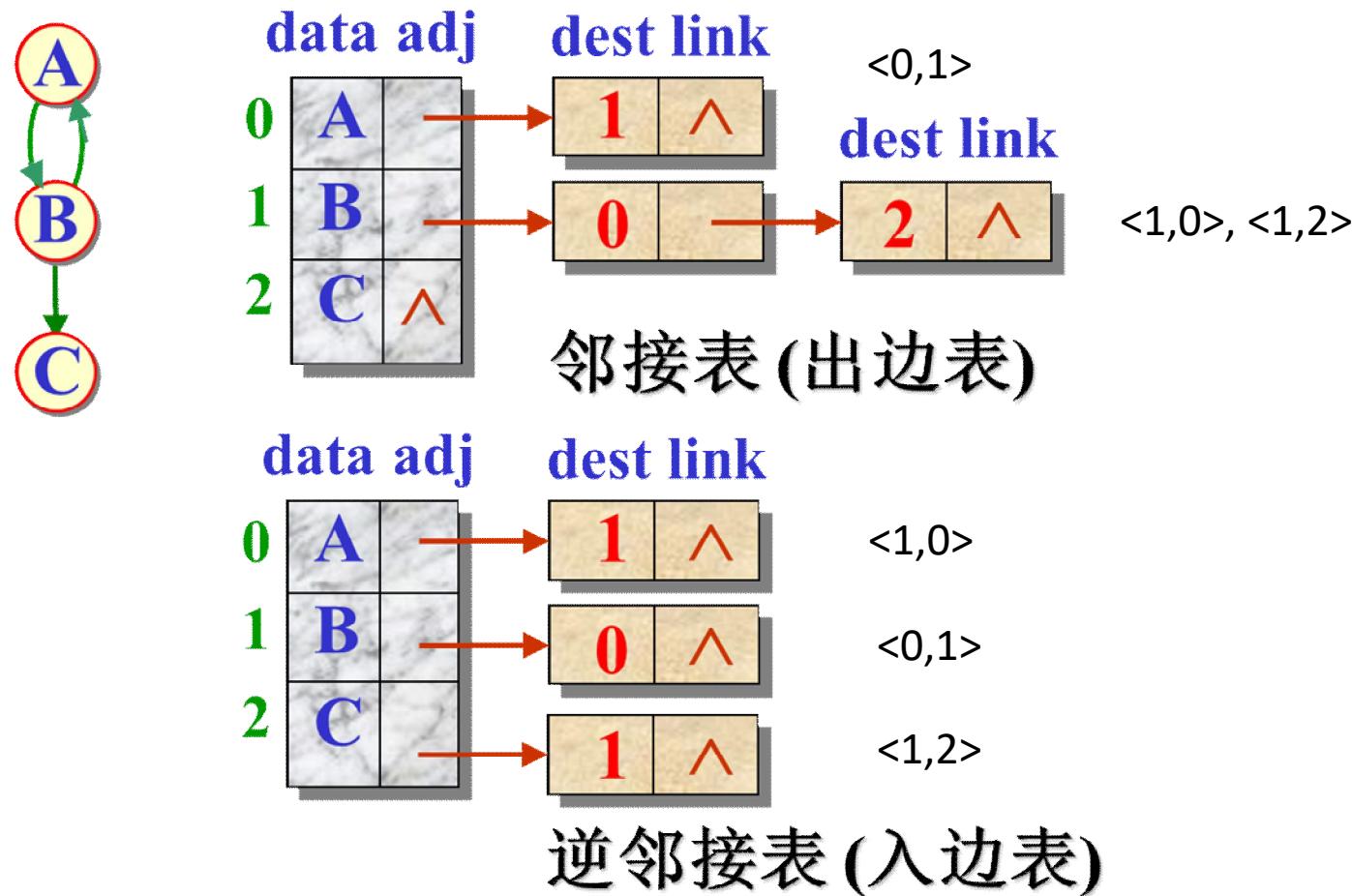
无向图的邻接表



同一个顶点发出的边链接在同一个边链表中，每一个链结点代表一条边(边结点)，结点中有另一顶点的下标 **dest** 和指针 **link**。

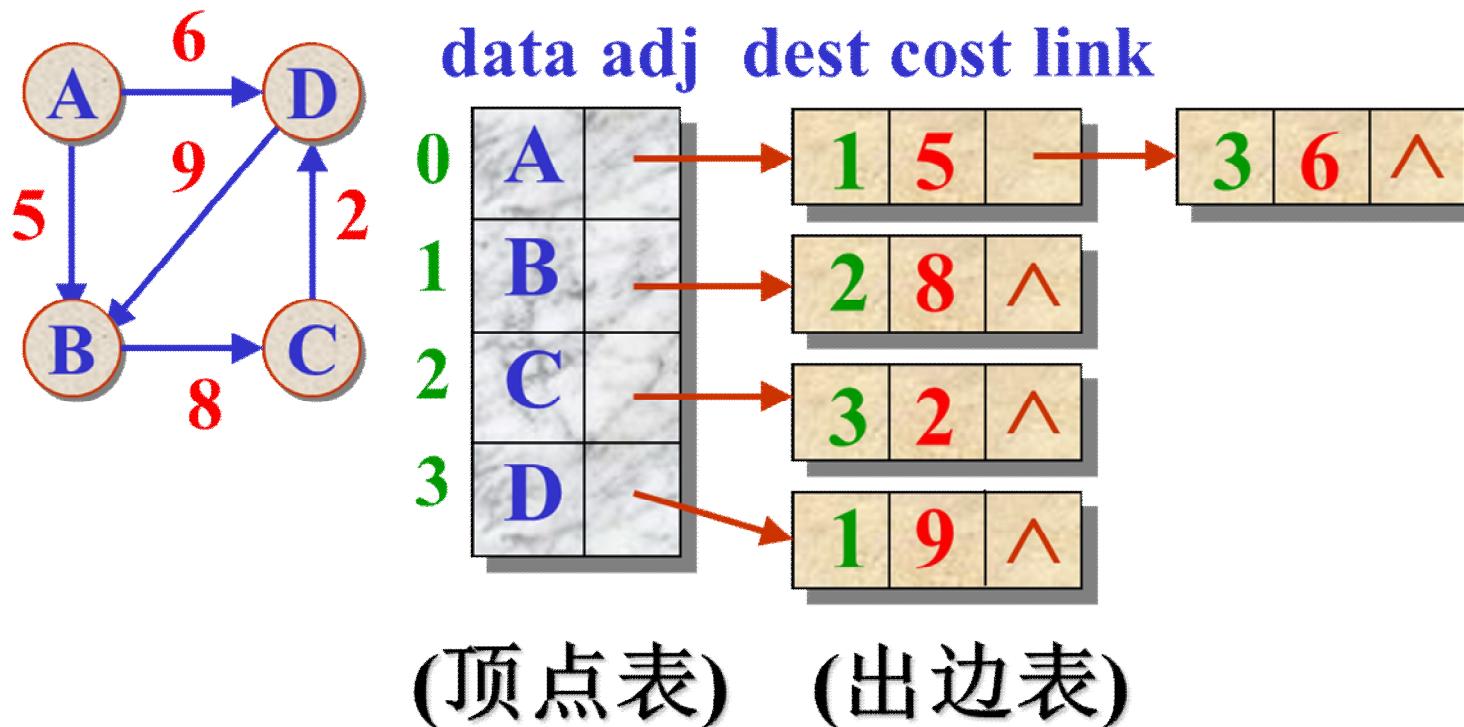
图的存储表示--邻接表 (Adjacency List)

- 有向图的邻接表和逆邻接表



图的存储表示--邻接表 (Adjacency List)

- 网络 (带权图) 的邻接表



图的存储表示--邻接表 (Adjacency List)

- **邻接表法的特点**
- 设图中有 n 个顶点， e 条边，则用邻接表表示无向图时，需要 n 个顶点结点， $2e$ 个边结点；用邻接表表示有向图时，若不考虑逆邻接表，只需 n 个顶点结点， e 个边结点。
- 表头向量中每个分量就是一个单链表的头结点，分量个数就是图中的顶点数目；
- 在边或弧稀疏的条件下，用邻接表表示比用邻接矩阵表示节省存储空间；
- 在无向图，顶点 V_i 的度是第 i 个链表的结点数；
- 对有向图可以建立正邻接表或逆邻接表。正邻接表是以顶点 V_i 为出度(即为弧的起点)而建立的邻接表；逆邻接表是以顶点 V_i 为入度(即为弧的终点)而建立的邻接表；
- 在邻接表的边链表中，各个边结点的链入顺序任意，视边结点输入次序而定，往往按照顶点编号从小到大排列。

邻接表的实现

• 结点及其类型定义

```
#define MAX_VEX 30 /* 最大顶点数 */
typedef int InfoType;
typedef enum {DG, AG, WDG,WAG} GraphKind ;
typedef struct LinkNode
{
    int adjvex ; // 邻接点在头结点数组中的位置(下标)
    InfoType info ; // 与边或弧相关的信息, 如权值
    struct LinkNode *nextarc ; // 指向下一个表结点
}LinkNode ; /* 表结点类型定义 */
```

```
typedef struct VexNode
{
    VexType data; // 顶点信息
    int indegree ; // 顶点的度, 有向图是入度或出度或没有
    LinkNode *firstarc ; // 指向第一个表结点
}VexNode ; /* 顶点结点类型定义 */
```

```
typedef struct
{
    /* 图的种类标志 */
    GraphKind kind ;
    int vexnum ;
    VexNode AdjList[MAX_VEX] ;
}ALGraph ; /* 图的结构定义 */
```

图的存储表示

- 邻接多重表 (Adjacency Multilist)
 - 在邻接多重表中, 每一条边只有一个边结点。为有关边的处理提供了方便。
 - 无向图的情形
 - ◆ 边结点的结构



其中, mark 是记录是否处理过的标记; vertex1和vertex2是该边两顶点位置。path1域是链接指针, 指向另一条依附顶点vertex1的边 ; path2 是指向另一条依附顶点vertex2的边链接指针。

图的存储表示

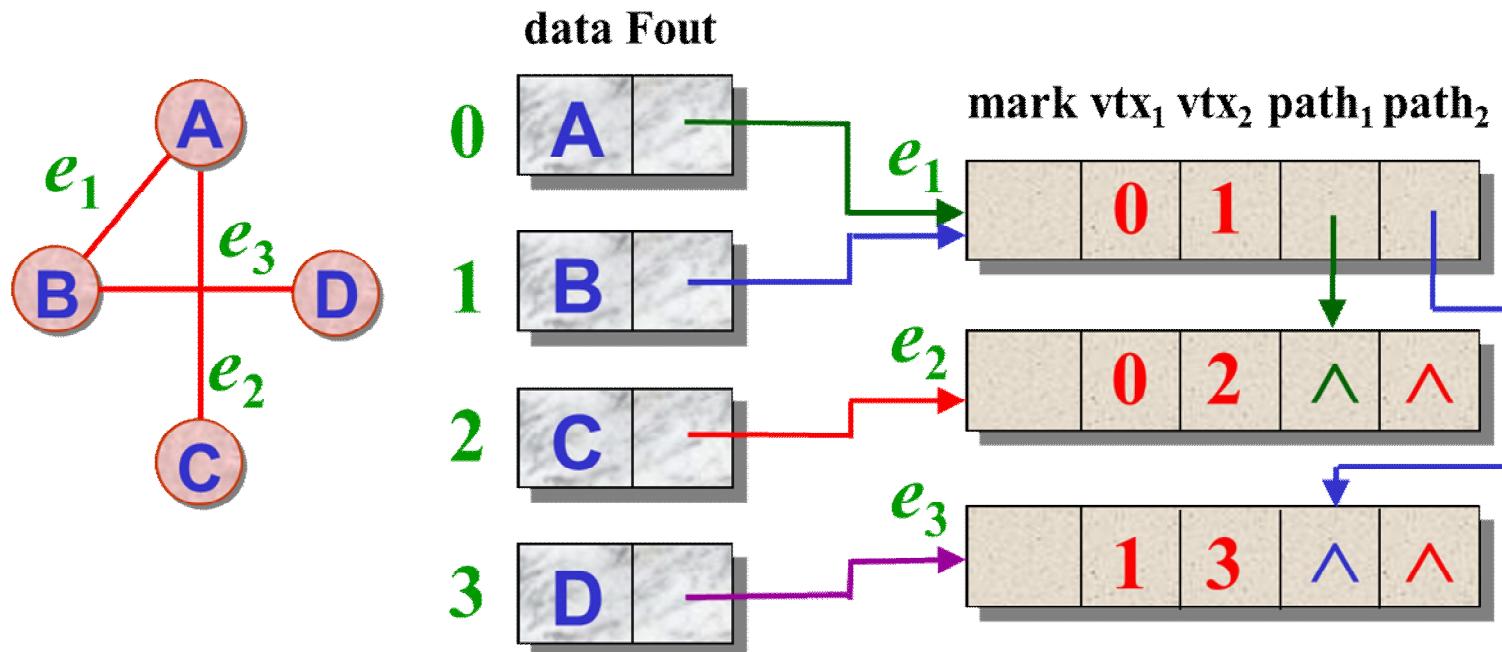
- 需要时还可设置一个存放与该边相关的权值的域 **cost**。
- 顶点结点的结构



- 存储顶点信息的结点表**以顺序表方式组织**，每一个顶点结点有两个数据成员：其中，**data** 存放与该顶点相关的信息，**Firstout** 是指示第一条依附该顶点的边的指针。
- 在邻接多重表中，所有依附同一个顶点的边都链接在同一个单链表中。

图的存储表示

- 从顶点*i*出发, 可以循链找到所有依附于该顶点的边, 也可以找到它的所有邻接顶点。
- 邻接多重表的结构**



图的存储表示--十字链表

- 有向图的情形
- 在用邻接表表示有向图时, 有时需要同时使用邻接表和逆邻接表。用有向图的邻接多重表(十字链表)可把两个表结合起来表示。
 - ◆ 边结点的结构



其中，mark是处理标记；vertex1和vertex2指明该有向边始顶点和终顶点的位置。path1是指向始顶点与该边相同的下一条边的指针；path2是指向终顶点与该边相同的下一条边的指针。需要时还可有权值域cost。

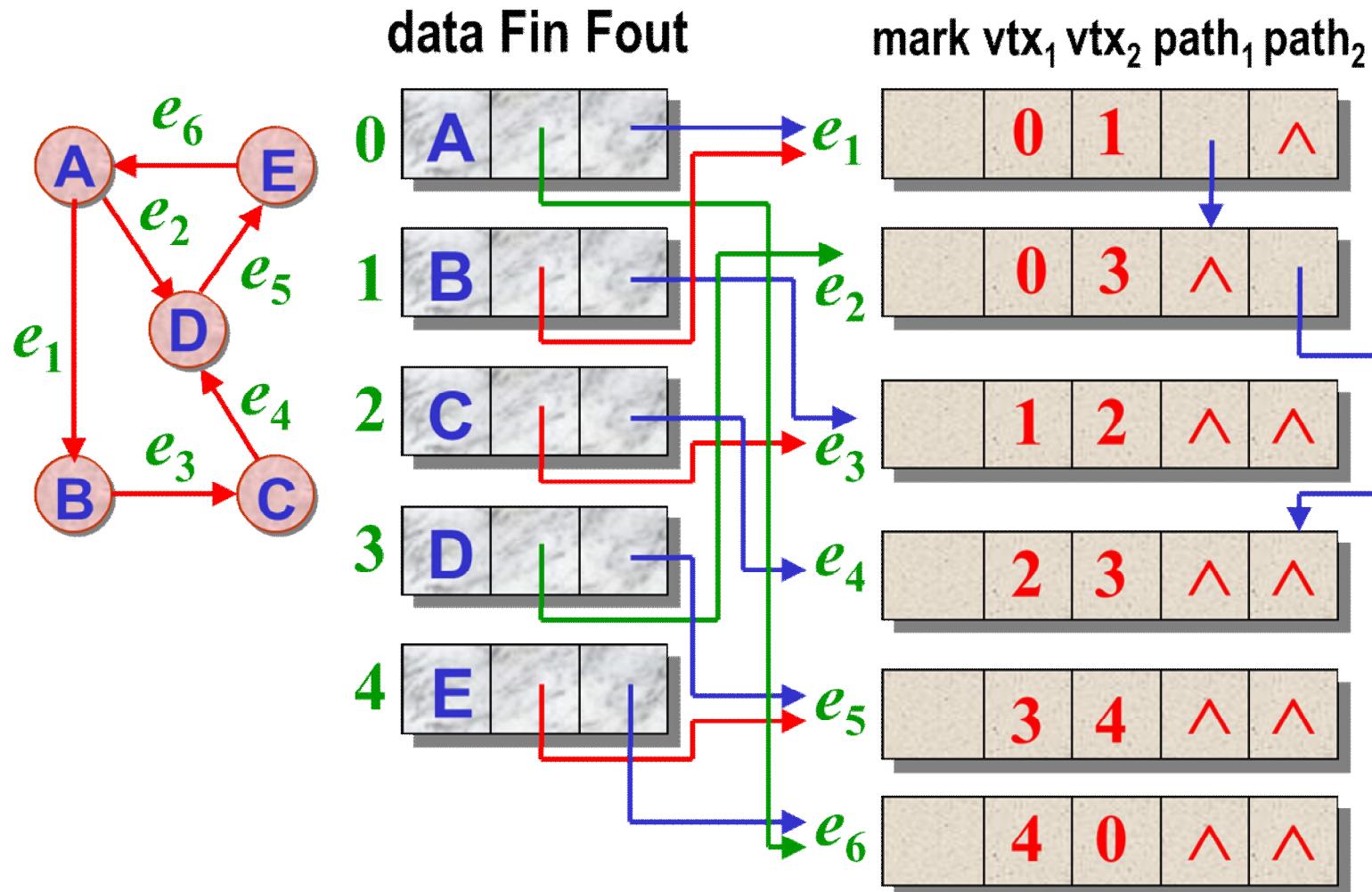
图的存储表示--十字链表

- 顶点结点的结构



每个顶点有一个结点，它相当于出边表和入边表的表头结点：
其中，数据成员**data**存放与该顶点相关的信息，指针**Firstin**指示以该顶点为终顶点的入边表的第一条边，**Firstout**指示以该顶点为始顶点的出边表的第一条边。

图的存储表示--十字链表



图的存储表示--十字链表

- 结点类型定义

```
#define INFINITY MAX_VAL /* 最大值∞ */  
#define MAX_VEX 30 // 最大顶点数  
typedef struct ArcNode  
{ int tailvex , headvex ; // 尾结点和头结  
点在图中的位置  
    InfoType info ; // 与弧相关的信  
息, 如权值  
    struct ArcNode *hlink , *tlink ;  
}ArcNode ; /* 弧结点类型定义 */
```

```
typedef struct VexNode  
{ VexType data; // 顶点信息  
    ArcNode *firstin , *firstout ;  
}VexNode ; /* 顶点结点类型定义 */  
  
typedef struct  
{ int vexnum ;  
    VexNode xlist[MAX_VEX] ;  
}OLGraph ; /* 图的类型定义 */
```

从这种存储结构图可以看出，从一个顶点结点的firstout出发，沿表结点的path1指针构成了正邻接表的链表结构，而从一个顶点结点的firstin出发，沿表结点的path2指针构成了逆邻接表的链表结构。

图的遍历与连通性

- 从已给的连通图中某一顶点出发，沿着一些边访遍图中所有的顶点，且使每个顶点仅被访问一次，就叫做**图的遍历** (Graph Traversal)。
- 图中可能存在回路，且图的任一顶点都可能与其他顶点相通，在访问完某个顶点之后可能会沿着某些边又回到了曾经访问过的顶点。
- 为了避免重复访问，可设置一个标志顶点是否被访问过的辅助数组 **visited []**。

图的遍历与连通性

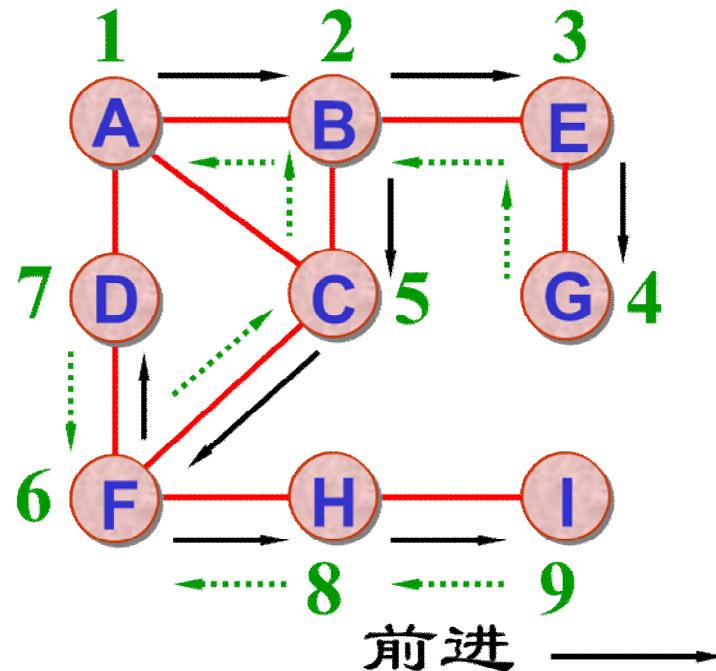
- 辅助数组 `visited []` 的初始状态为 0, 在图的遍历过程中, 一旦某一个顶点 i 被访问, 就立即让 `visited [i]` 为 1, 防止它被多次访问。
- 图的遍历的分类:
 - ◆ 深度优先搜索
DFS (Depth First Search)
 - ◆ 广度优先搜索
BFS (Breadth First Search)

深度优先搜索DFS (Depth First Search)

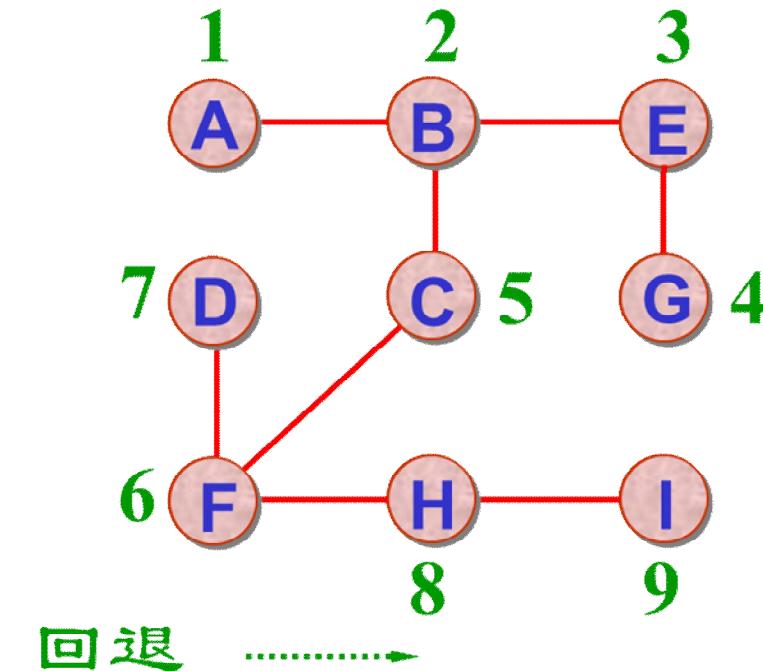
- DFS 在访问图中某一起始顶点 v 后, 由 v 出发, 访问它的任一邻接顶点 w_1 ; 再从 w_1 出发, 访问与 w_1 邻接但还没有访问过的顶点 w_2 ; 然后再从 w_2 出发, 进行类似的访问, … 如此进行下去, 直至到达所有的邻接顶点都被访问过的顶点 u 为止。
- 接着, 退回一步, 退到前一次刚访问过的顶点, 看是否还有其它没有被访问的邻接顶点。如果有, 则访问此顶点, 之后再从此顶点出发, 进行与前述类似的访问; 如果没有, 就再退回一步进行搜索。
- 重复上述过程, 直到连通图中所有顶点都被访问过为止。

图的遍历与连通性

- 深度优先搜索的示例



深度优先搜索过程



深度优先生成树

深度优先搜索DFS (Depth First Search)

- 由算法思想知，这是一个递归过程。因此，先设计一个从某个顶点(编号)为 v_0 开始深度优先搜索的函数，便于调用。
- 在遍历整个图时，可以对图中的每一个未访问的顶点执行所定义的函数。

```
template<class Type>
void DFSTraverse (Graph<Type>& G) {
    int n = G.NumberOfVertices( );
    static int * visited = new int [n];
    for ( int i = 0; i < n; i++ )      // 访问数组
        visited [i] = 0;           // visited 初始化
    DFS (G, 0, visited);      // 开始搜索
    delete [ ] visited;         // 释放 visited
}
```

深度优先搜索DFS (Depth First Search)

```
template<class Type>
void DFS ( Graph<Type>& G, int v,
int visited [ ] ) {
    cout << G.GetValue (v) << ' '; // 访问顶点 v
    visited[v] = 1;                  // 顶点 v 作访问标记
    int w = G.GetFirstNeighbor (v);
    while ( w != -1 ) {             // 若邻接顶点 w 存在
        if ( !visited[w] ) DFS ( G, w, visited );
        // 若顶点 w 未访问过, 递归访问顶点 w
        w = G.GetNextNeighbor ( v, w );
        // 取顶点 v 排在 w 后的下一个邻接顶点
    }
}
```

什么是递归？

◀ Back to Safari 19:52 ⚡ 63% 🔋



回答

...

清华TUNA协会的TUNA是什么缩写？



Justin Wong

+ 关注

现在是 Tsinghua University TUNA
Association 的强行递归缩写。

创建于 2016-01-12

著作权归作者所有



17



感谢作者



加入收藏



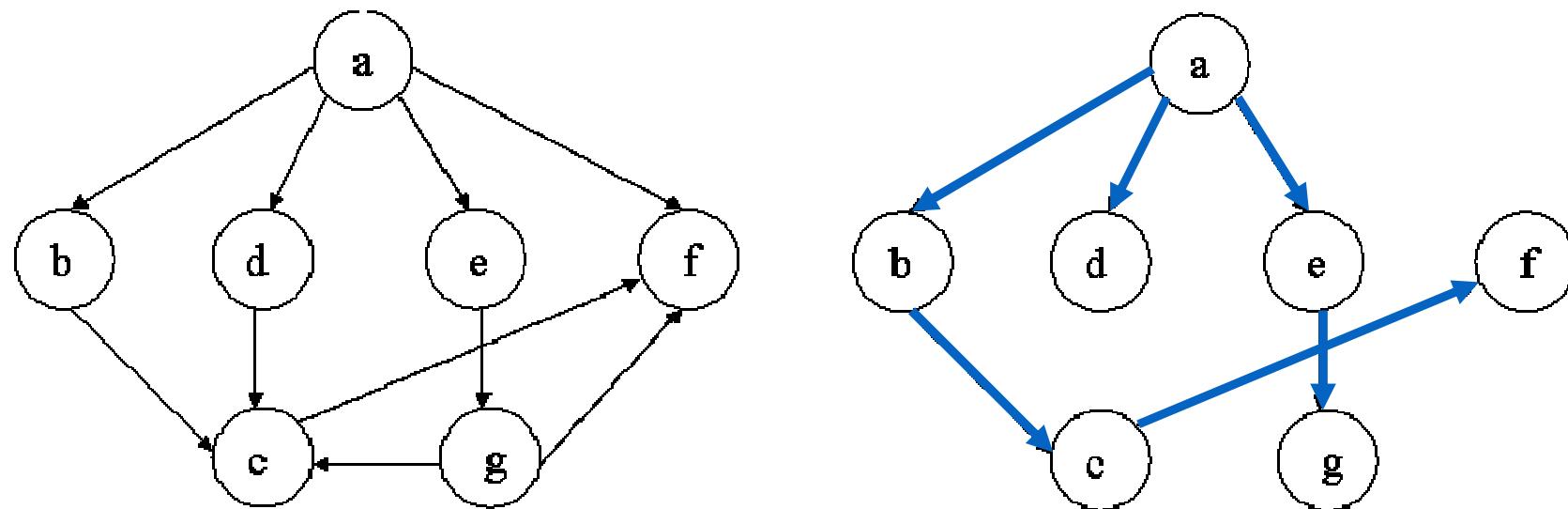
评论 (2)

深度优先搜索DFS (Depth First Search)

- 算法分析
- 遍历时，对图的每个顶点至多调用一次DFS函数。其实质就是对每个顶点查找邻接顶点的过程，取决于存储结构。当图有 e 条边，其时间复杂度为 $O(e)$ ，总时间复杂度为 $O(n+e)$ 。

深度优先搜索DFS (Depth First Search)

- 给出深度优先搜索的顺序，并绘制出对应的遍历图



深度优先搜索的顺序是a, b, c, f, d, e, g

广度优先搜索BFS (Breadth First Search)

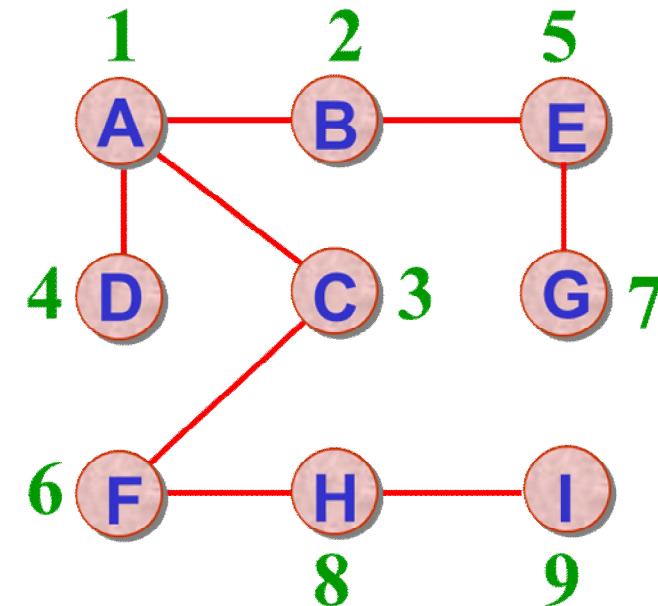
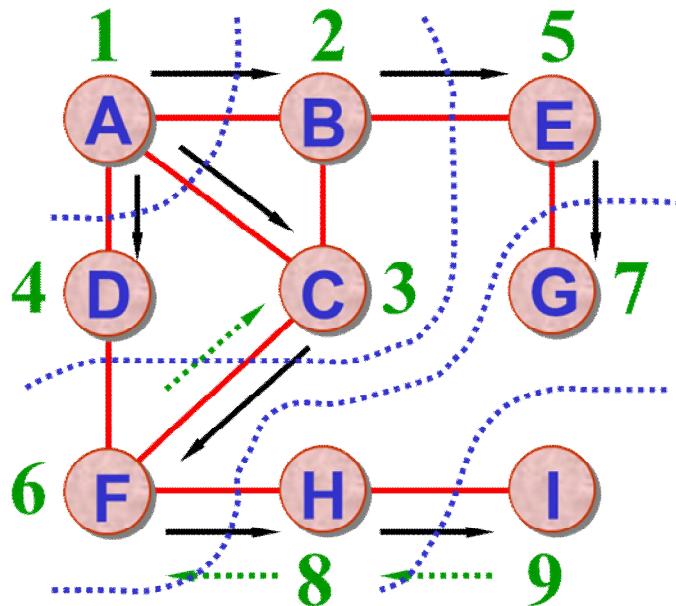
- BFS在访问了起始顶点 v 之后, 由 v 出发, 依次访问 v 的各个未被访问过的邻接顶点 w_1, w_2, \dots, w_t , 然后再顺序访问 w_1, w_2, \dots, w_t 的所有还未被访问过的邻接顶点。再从这些访问过的顶点出发, 再访问它们的所有还未被访问过的邻接顶点, ... 如此做下去, 直到图中所有顶点都被访问到为止。
- 广度优先搜索是一种分层的搜索过程, 每向前走一步可能访问一批顶点, 不像深度优先搜索那样有往回退的情况。因此, 广度优先搜索不是一个递归的过程。

广度优先搜索BFS (Breadth First Search)

- 为了实现逐层访问, 算法中使用了一个队列, 以记忆正在访问的这一层和上一层的顶点, 以便于向下一层访问。
- 为避免重复访问, 需要一个辅助数组 `visited []`, 给被访问过的顶点加标记。

广度优先搜索BFS (Breadth First Search)

- 广度优先搜索的示例



广度优先搜索BFS (Breadth First Search)

```
template<class Type>
void BFS ( Graph <Type>& G, int v ) {
    int n = G.NumberOfVertices( );
    int * visited = new int[n];
    for ( int i = 0; i < n; i++ )
        visited[i] = 0;
    cout << G.GetValue (v) << ' ';
    visited[v] = 1;
    Queue<int> q;
    // 进队列
    q.Enqueue (v);
    ...
    delete [ ] visited;
}

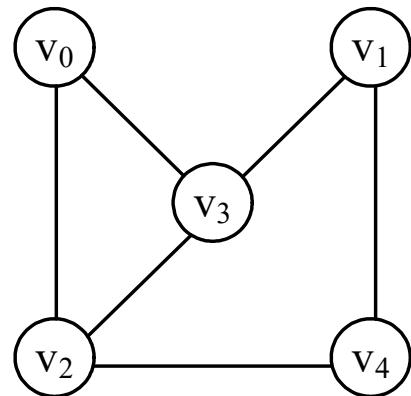
while ( !q.IsEmpty ( ) ) { //队空搜索结束
    q.GetFront(v); q.DeQueue ( );
    int w = G.GetFirstNeighbor (v);
    while ( w != -1 ) { //若邻接顶点 w 存在
        if ( !visited[w] ) { //未访问过
            cout << G.GetValue (w) << ' ';
            visited[w] = 1; q.Enqueue (w);
        }
        w = G.GetNextNeighbor (v, w);
    } //重复检测 v 的所有邻接
} //外层循环，判队列空否
```

广度优先搜索BFS (Breadth First Search)

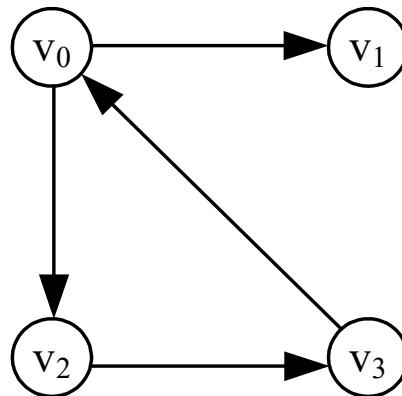
- 用广度优先搜索算法遍历图与深度优先搜索算法遍历图的唯一区别是邻接点搜索次序不同，因此，广度优先搜索算法遍历图的总时间复杂度为 $O(n+e)$ 。
- 图的遍历可以系统地访问图中的每个顶点，因此，图的遍历算法是图的最基本、最重要的算法，许多有关图的操作都是在图的遍历基础之上加以变化来实现的。

课题练习

- 画出下图G1和G2的邻接表，以及G2的逆邻接表



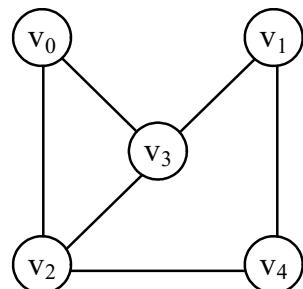
(a) 无向图G1



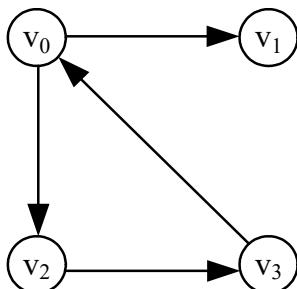
(b) 有向图G2

课题练习-答案

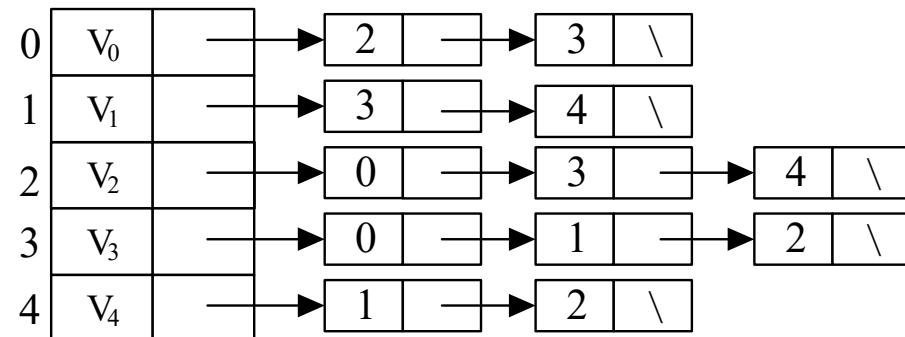
- 画出下图G1和G2的邻接表，以及G2的逆邻接表



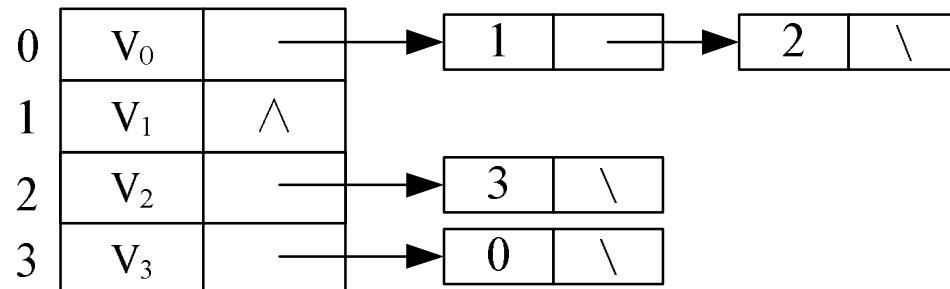
(a) 无向图G₁



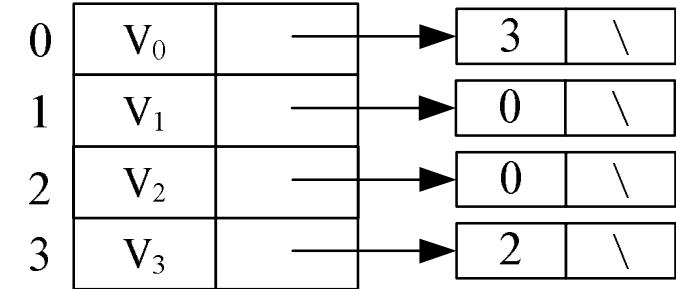
(b) 有向图G₂



无向图G₁的邻接表表示



(a) 有向图G₂的邻接表



(b) 有向图G₂的逆邻接表

有向图G2的邻接表和逆邻接表表示

谢谢！

