

第七章 图

本章介绍另一种非线性数据结构 —— 图

图：是一种多对多的结构关系，每个元素可以有零个或多个直接前趋；零个或多个直接后继；

第七章 图

- 7.1 图的概念
- 7.2 图的存储结构
- 7.3 图的遍历
- 7.4 生成树
- 7.5 最短路径
- 7.6 拓扑排序

第七章 图

学习要点

1. 熟悉图的各种存储结构及其构造算法，了解实际问题的求解效率与采用何种存储结构和算法有密切联系；
2. 熟练掌握图的两种遍历：深度优先遍历和广度优先遍历的算法。在学习中应注意图的遍历算法与树的遍历算法之间的类似和差异。树的先根遍历是一种深度优先搜索策略，树的层次遍历是一种广度优先搜索策略
3. 理解课件中讨论的各种图的算法；

第七章 图



7.1 图的概念

- 一 图的概念
- 二 图的应用
- 三 图的基本操作
- 四 图的基本术语

一 图的概念

图的定义

图G由两个集合构成，记作 $G=\langle V, E \rangle$ 其中V是顶点的非空有限集合，E是边的有限集合，其中边是顶点的无序对或有序对集合。

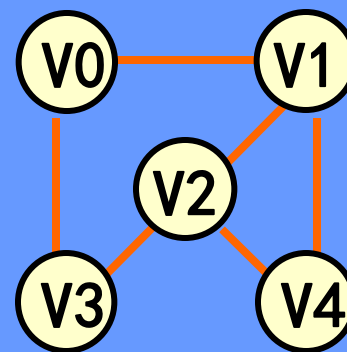
例

$$G1=\langle V1, E1 \rangle$$

$$V1=\{v_0, v_1, v_2, v_3, v_4\}$$

$$E1=\{(v_0, v_1), (v_0, v_3), (v_1, v_2), (v_1, v_4), (v_2, v_3), (v_2, v_4)\}$$

无序对 (v_i, v_j) ：
用连接顶点 v_i 、 v_j 的线段
表示，称为无向边；

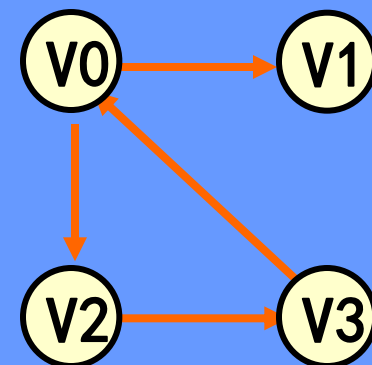


$$G2 = \langle V2, E2 \rangle$$

$$V2 = \{v_0, v_1, v_2, v_3\}$$

$$E2 = \{ \langle v_0, v_1 \rangle, \langle v_0, v_2 \rangle, \langle v_2, v_3 \rangle, \langle v_3, v_0 \rangle \}$$

有序对 $\langle v_i, v_j \rangle$:
用以为 v_i 起点、以 v_j 为终点的有向线段表示, 称为有向边或弧;



G2 图示

无向图: 在图G中, 若所有边是无向边, 则称G为无向图;

有向图: 在图G中, 若所有边是有向边, 则称G为有向图;

混和图: 在图G中, 即有无向边也有有向边, 则称G为混合图;

二 图的应用举例

例1 交通图（公路、铁路）

顶点：地点

边：连接地点的公路

交通图中的有单行道双行道，分别用有向边、无向边表示；

例2 电路图

顶点：元件

边：连接元件之间的线路

例3 通讯线路图

顶点：地点

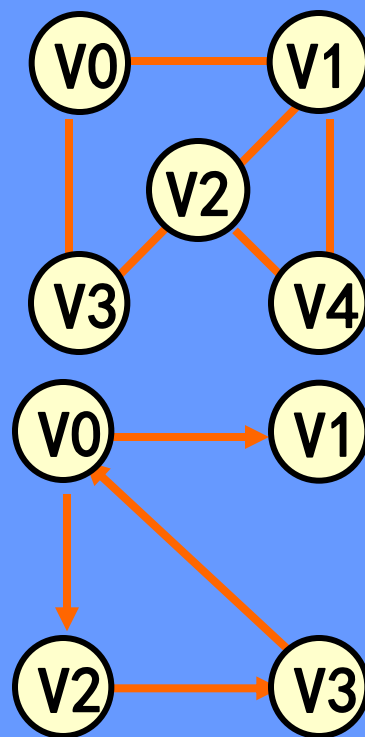
边：地点间的连线

例4 各种流程图

如产品的生产流程图

顶点：工序

边：各道工序之间的顺序关系



三 图的基本术语

1 邻接点及关联边

邻接点：边的两个顶点

关联边：若边 $e = (v, u)$ ，则称顶点 v 、 u 关连边 e

2 顶点的度、入度、出度

顶点 V 的度 = 与 V 相关联的边的数目

在有向图中：

顶点 V 的出度 = 以 V 为起点有向边数

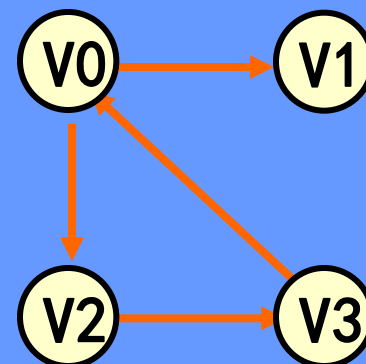
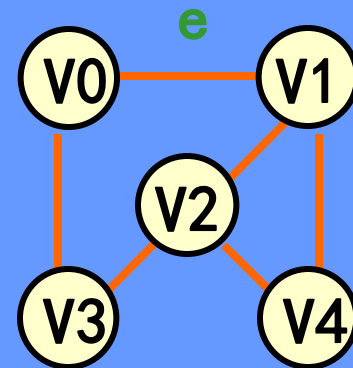
顶点 V 的入度 = 以 V 为终点有向边数

顶点 V 的度 = V 的出度 + V 的入度

设图 G 的顶点数为 n ，边数为 e

图的所有顶点的度数和 = $2 * e$

（每条边对图的所有顶点的度数和“贡献”2度）



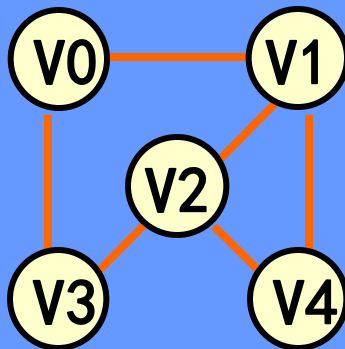
3 路径、回路

无向图 $G = (V, E)$ 中的顶点序列 v_1, v_2, \dots, v_k , 若 $(v_i, v_{i+1}) \in E$ ($i=1, 2, \dots, k-1$), $v = v_1$, $u = v_k$, 则称该序列是从顶点 v 到顶点 u 的路径; 若 $v=u$, 则称该序列为回路;

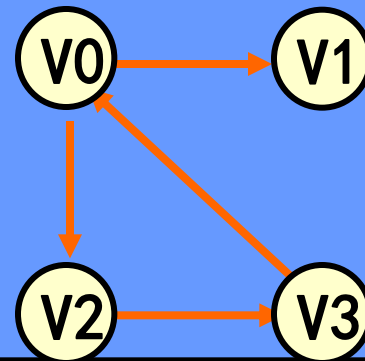
有向图 $D = (V, E)$ 中的顶点序列 v_1, v_2, \dots, v_k , 若 $\langle v_i, v_{i+1} \rangle \in E$ ($i=1, 2, \dots, k-1$), $v = v_1$, $u = v_k$, 则称该序列是从顶点 v 到顶点 u 的路径; 若 $v=u$, 则称该序列为回路;

例

在图1中, V_0, V_1, V_2, V_3 是 V_0 到 V_3 的路径; V_0, V_1, V_2, V_3, V_0 是回路; 在图2中, V_0, V_2, V_3 是 V_0 到 V_3 的路径; V_0, V_2, V_3, V_0 是回路;



无向图G1



有向图G2

3 简单路径和简单回路

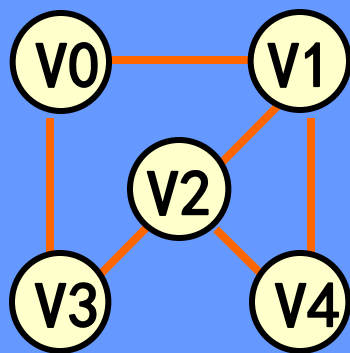
在一条路径中, 若除起点和终点外, 所有顶点各不相同, 则称该路径为简单路径;

由简单路径组成的回路称为简单回路;

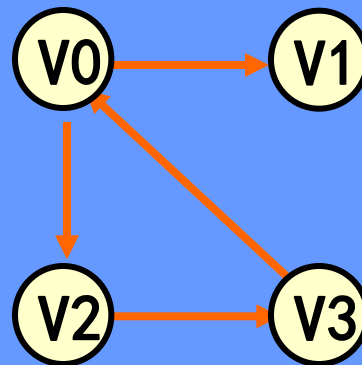
例

在图1中, V_0, V_1, V_2, V_3 是简单路径; V_0, V_1, V_2, V_4, V_1 不是简单路径; 在图2中, V_0, V_2, V_3, V_0 是简单回路;

无向图G1



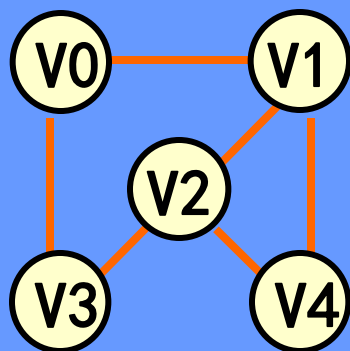
有向图G2



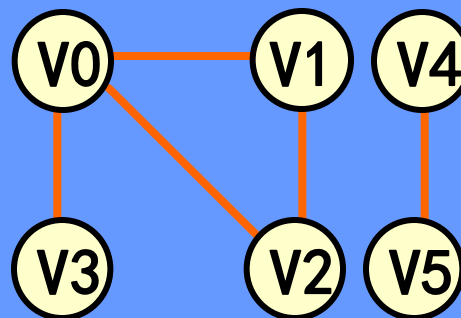
4 连通图（强连通图）

在无（有）向图 $G = \langle V, E \rangle$ 中，若对任何两个顶点 v 、 u 都存在从 v 到 u 的路径，则称 G 是连通图（强连通图）

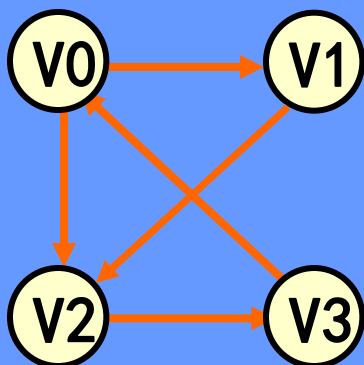
连通图



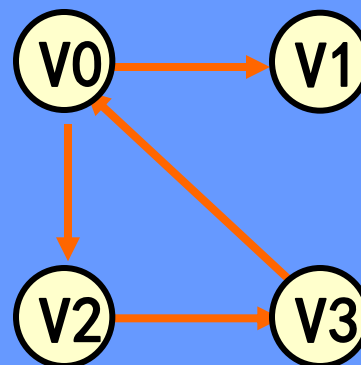
非连通图



强连通图



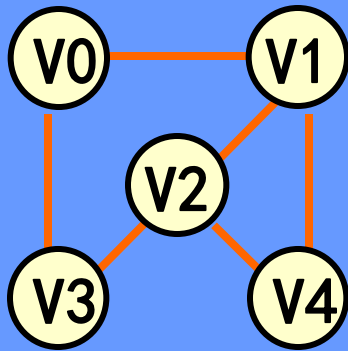
非强连通图



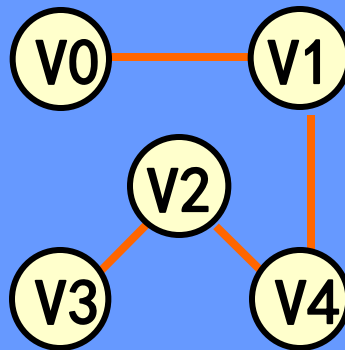
5 子图

设有两个图 $G=(V, E)$ 、 $G_1=(V_1, E_1)$ ，若 $V_1 \subseteq V$ ， $E_1 \subseteq E$ ， E_1 关联的顶点都在 V_1 中，则称 G_1 是 G 的子图；

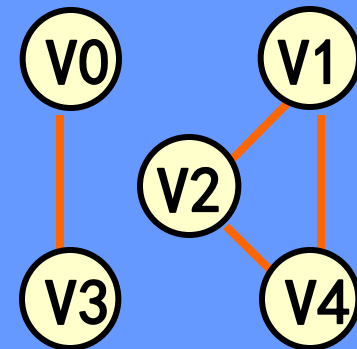
例 (b)、(c) 是 (a) 的子图



(a)



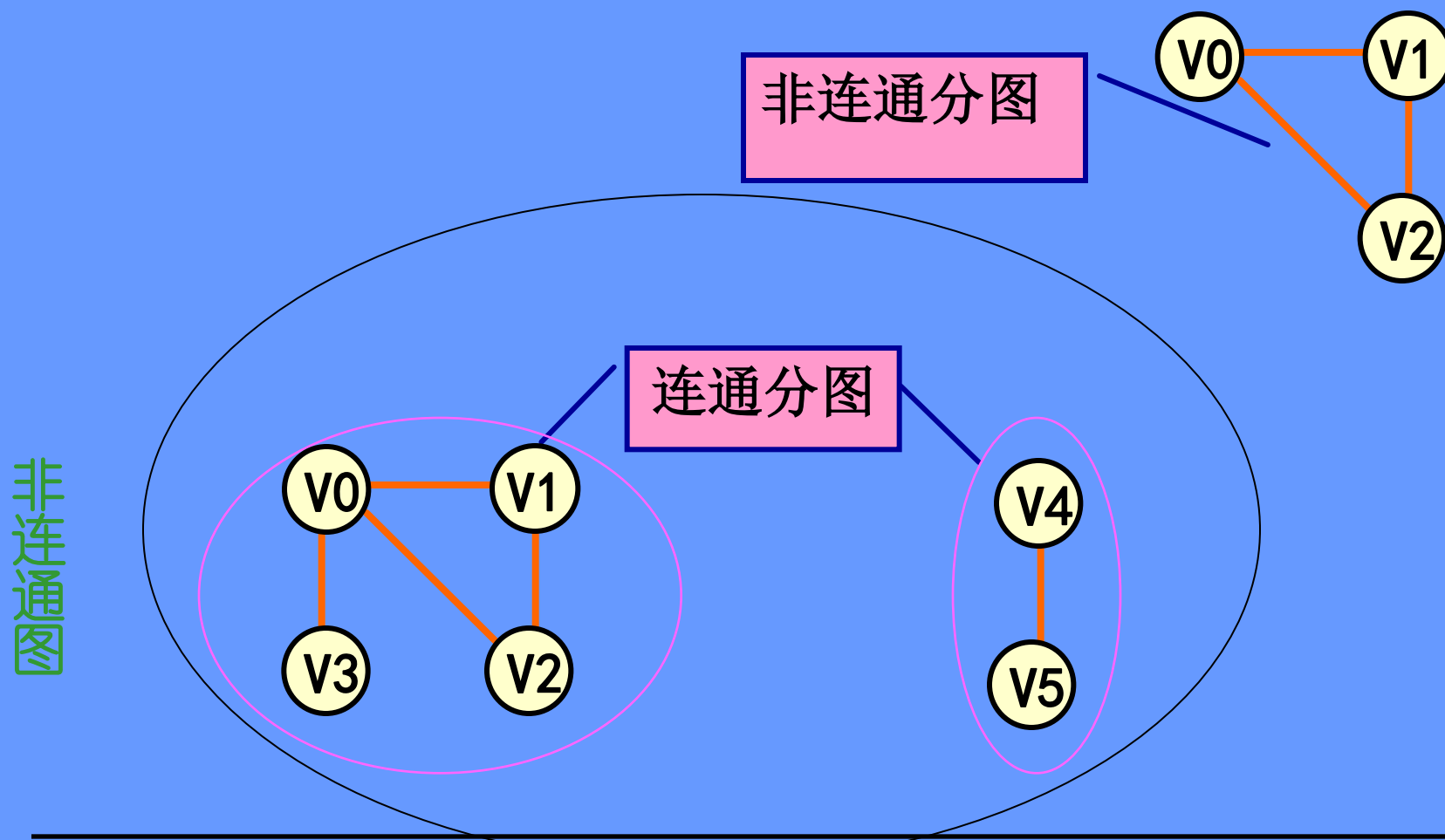
(b)



(c)

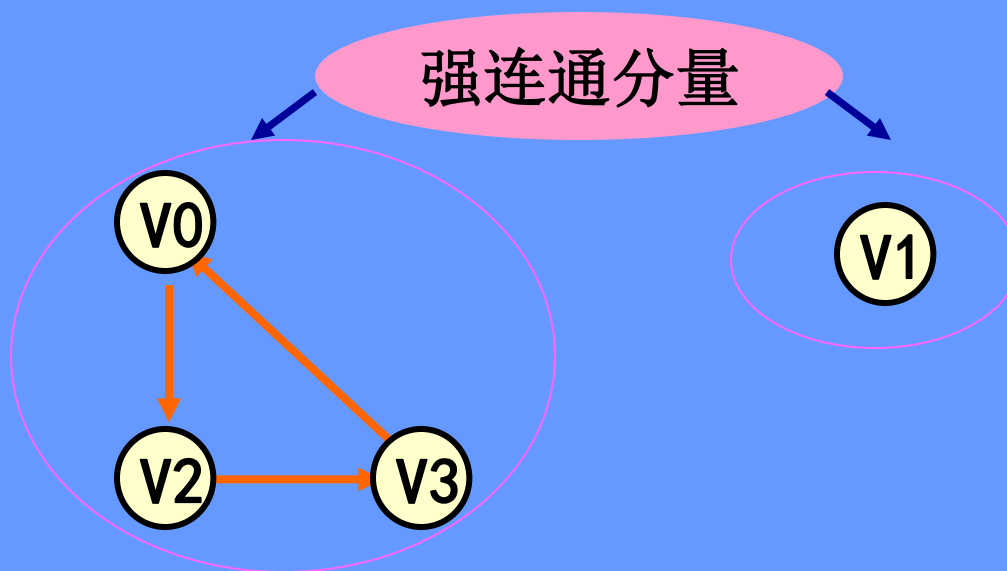
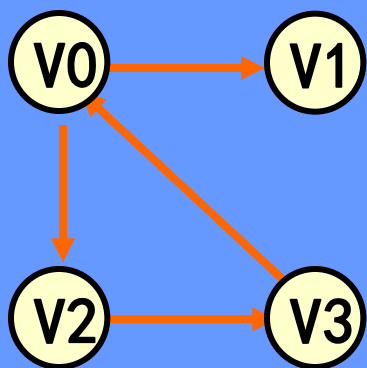
无向图 G 的极大连通子图称为 G 的连通分量

极大连通子图意思是：该子图是 G 连通子图，将 G 的任何不在该子图中的顶点加入，子图不再连通；



有向图D的极大强连通子图称为D的强连通分量

极大强连通子图意思是：该子图是D强连通子图，将D的任何不在该子图中的顶点加入，子图不再是强连通的；



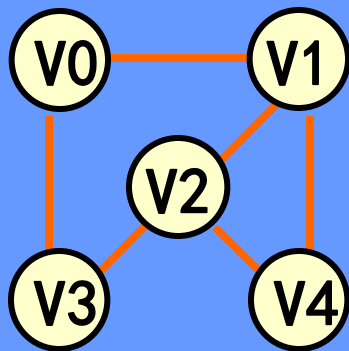
7 生成树

包含无向图 G 所有顶点的极小连通子图称为 G 的生成树

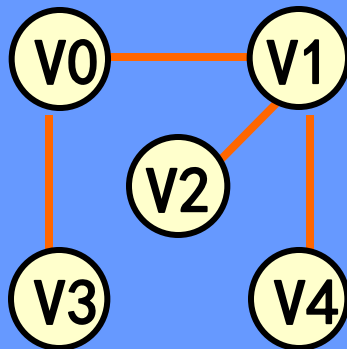
极小连通子图意思是：该子图是 G 的连通子图，在该子图中删除任何一条边，子图不再连通，

若 T 是 G 的生成树当且仅当 T 满足如下条件

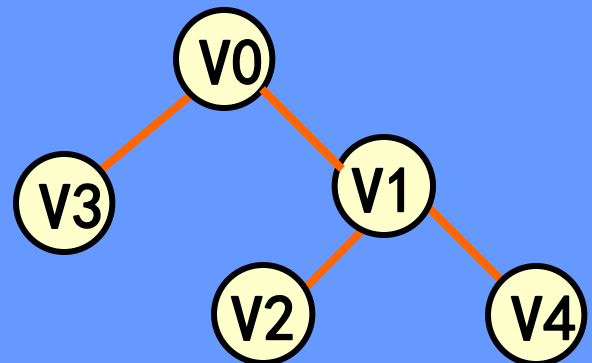
- T 是 G 的连通子图
- T 包含 G 的所有顶点
- T 中无回路



连通图 $G1$



$G1$ 的生成树

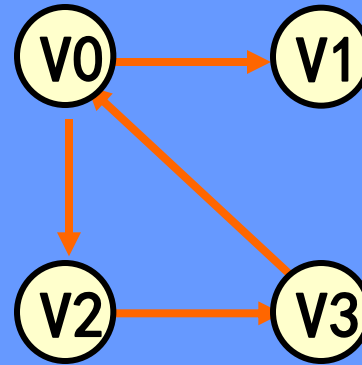
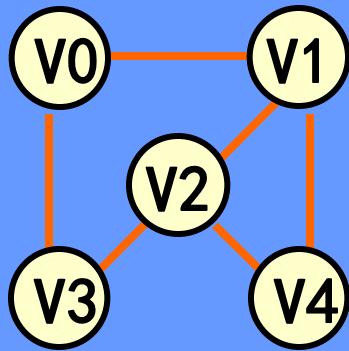


第七章 图



7.2 图的存储结构

- 一 数组表示法
- 二 邻接表



图的存储结构至少要保存两类信息:

- 1) 顶点的数据
- 2) 顶点间的关系

如何表示顶点间的关系?

顶点的编号

约定:

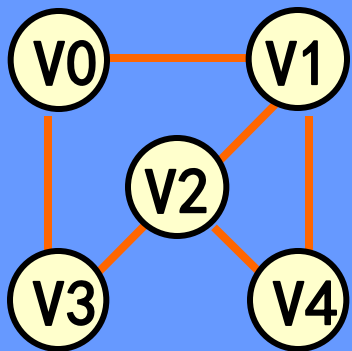
$G = \langle V, E \rangle$ 是图, $V = \{v_0, v_1, v_2, \dots, v_{n-1}\}$, 设顶点的
角标为它的编号

在数组表示法中,

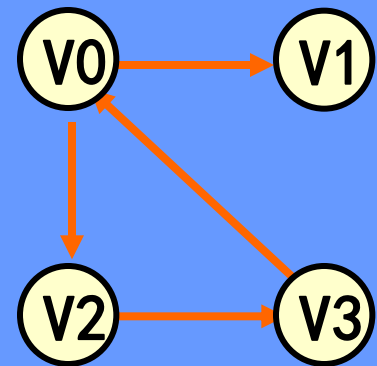
一 数组表示法(邻接矩阵) 用邻接矩阵表示顶点间的关系

邻接矩阵: G 的邻接矩阵是满足如下条件的 n 阶矩阵:

$$A[i][j] = \begin{cases} 1 & \text{若 } (v_i, v_{i+1}) \in E \text{ 或 } \langle v_i, v_{i+1} \rangle \in E \\ 0 & \text{否则} \end{cases}$$



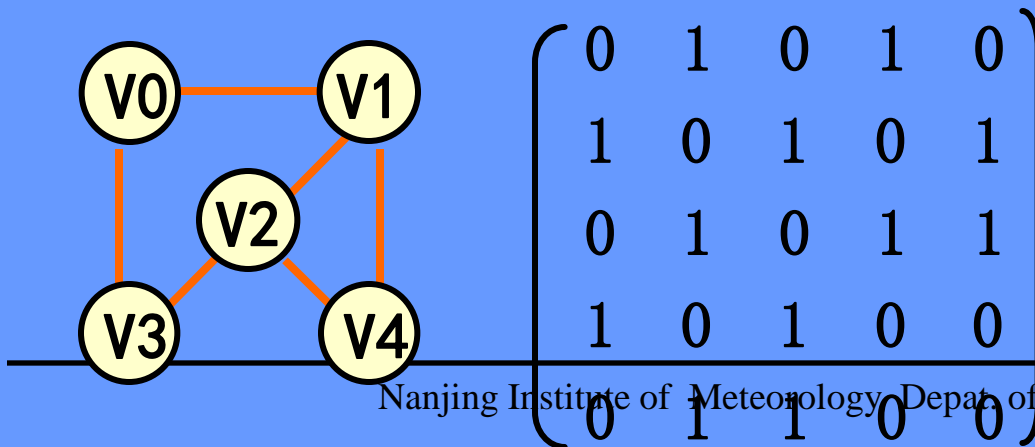
$$\begin{pmatrix} 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \end{pmatrix}$$



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

无向图数组表示法特点:

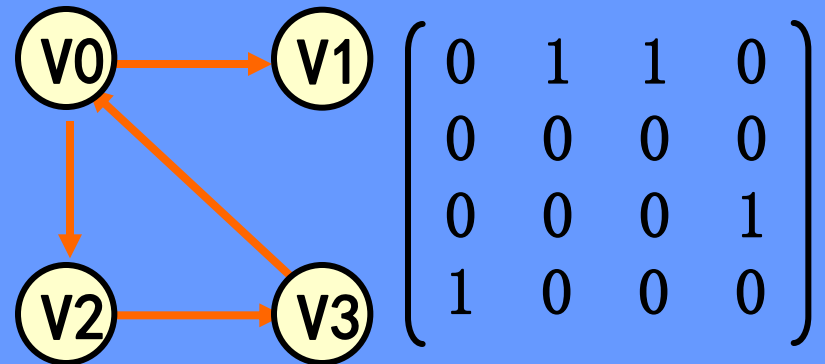
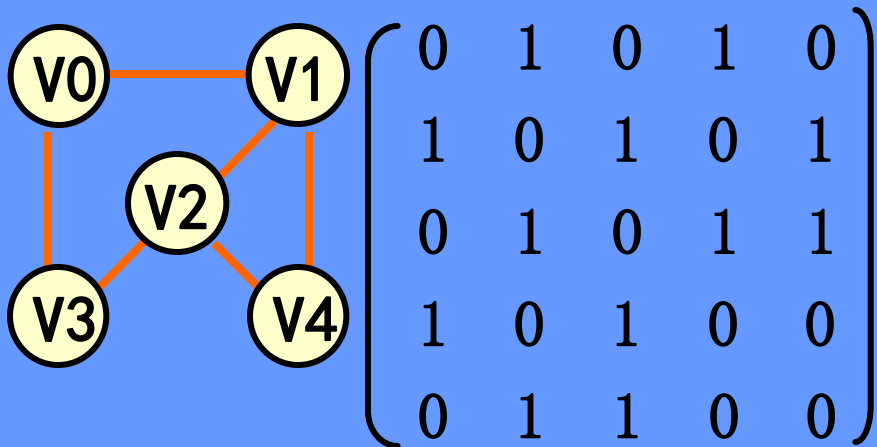
- 1) 无向图邻接矩阵是对称矩阵, 同一条边表示了两次;
 - 2) 顶点 v 的度: 等于二维数组对应行 (或列) 中1的个数;
 - 3) 判断两顶点 v 、 u 是否为邻接点: 只需判二维数组对应分量是否为1;
 - 4) 顶点不变, 在图中增加、删除边: 只需对二维数组对应分量赋值1或清0;
 - 5) 设图的顶点数为 n , 存储图用一维数组, 数组元素有 m 个 ($m \geq n$), 则 G 占用存储空间: $m+n^2$; G 占用存储空间只与它的顶点数有关, 与边数无关; 适用于边稠密的图;
- 对有向图的数组表示法可做类似的讨论



数组表示法的空间代价
只与图的顶点数有关

图的基本操作:

- 1) 求无向图某顶点 v_i 的度(或有向图中 v_i 的出度)。A[i][0]到A[i][n-1]中的非0个数, 即数组A中第i 行的非0 元素的个数;
- 2) 求有向图某顶点 v_i 的入度。: A[0][i]到A[n-1][i] 中的非0 个数, 即数组A中第i 列的非0 元素的个数;
- 3) 检测图中的总边数。扫描整个数组A, 统计出数组中非0元素的个数。无向图的总边数为非0元素个数的一半, 而有向图的总弧数为非0元素个数;



二 邻接表

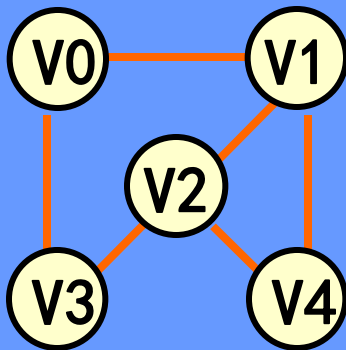
邻接表是图的链式存储结构

1 无向图的邻接表

顶点：通常按编号顺序将顶点数据存储在一维数组
关联同一顶点的边：用线性链表存储

该结点表示边
($V_i V_j$)，其中的1是 V_j
在一维数组中的位置

例



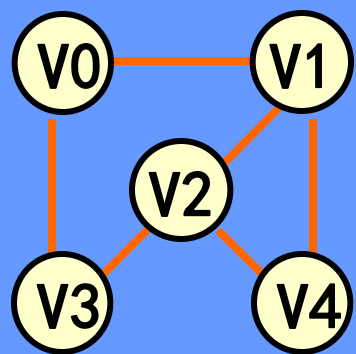
| 下标 | 编号 | link |
|-----|----|-------------|
| 0 | V0 | 1 → 3 ^ |
| 1 | V1 | 0 → 2 → 4 ^ |
| 2 | V2 | 1 → 3 → 4 ^ |
| 3 | V3 | 0 → 2 ^ |
| 4 | V4 | 1 → 2 ^ |
| | | |
| m-1 | | |

图的邻接表类型定义

```
struct node    //边（弧）结点的类型定义
{ int  vertex; //边（弧）的另一顶点的在数组中的位置
  struct node *link; //指向下一条边（弧）结点的指针
}; typedef struct NODE;
NODE adjlist[MAX]; // 邻接点链表的头指针所对应的数组
```

无向图的邻接表的特点

- 1) 在G邻接表中，同一条边对应两个结点；
- 2) 顶点v的度：等于v对应线性链表的长度；
- 3) 判定两顶点v，u是否邻接：要看v对应线性链表中有无对应的结点
- 4) 在G中增减边：要在两个单链表插入、删除结点；
- 5) 设存储顶点的一维数组大小为m($m \geq$ 图的顶点数n)，图的边数为e，G占用存储空间为： $m+2*e$ 。G占用存储空间与 G 的顶点数、边数均有关；适用于边稀疏的图



| | | | | | | | | |
|---|----|---|---|---|---|---|---|---|
| 0 | V0 | → | 1 | → | 3 | ∧ | | |
| 1 | V1 | → | 0 | → | 2 | → | 4 | ∧ |
| 2 | V2 | → | 1 | → | 3 | → | 4 | ∧ |
| 3 | V3 | → | 0 | → | 2 | ∧ | | |
| 4 | V4 | → | 1 | → | 2 | ∧ | | |
| | | | | | | | | |
| | | | | | | | | |
| | | | | | | | | |

邻接表的空间代价
与图的边及顶点数均有关

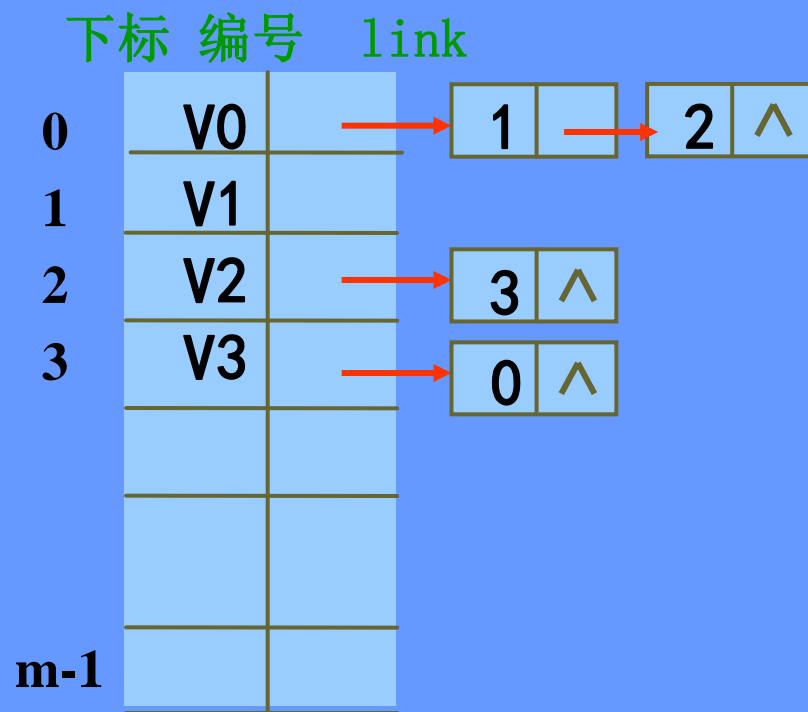
2 有向图的邻接表和逆邻接表

1) 有向图的邻接表

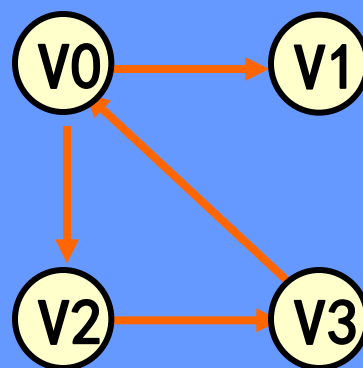
顶点：用一维数组存储（按编号顺序）

以同一顶点为起点的弧：用线性链表存储

例



类似于无向图的邻接表，
所不同的是：
以同一顶点为起点的弧：
用线性链表存储

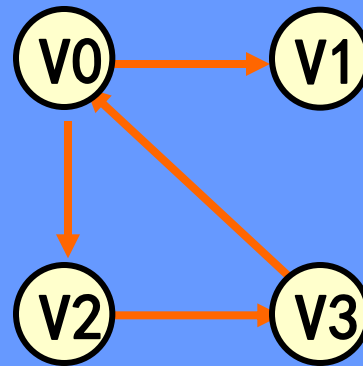
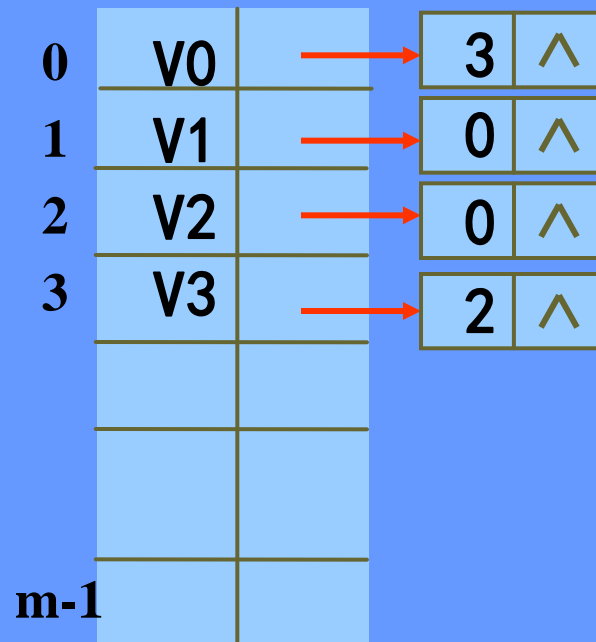


2) 有向图的逆邻接表

顶点：用一维数组存储（按编号顺序）

以同一顶点为**终点**的弧：用线性链表存储

例



类似于有向图的邻接表，
所不同的是：
以同一顶点为**终点**弧：
用线性链表存储

建立邻接表的算法

建立无向邻接表

```
int create(NODE *adjlist[ ] )  
{ NODE *p;  
  int num, i, v1, v2;  
  scanf("%d\n", &num); //读入结点数  
  for(i=0; i<num; i++) //初始化空关系图  
  { adjlist[i].link=NULL; adjlist[i].vertex=i; }
```

```
for(;;)
```

```
{ scanf(“%d to %d\n”, &v1, &v2); //读入一条边
```

```
if (v1<0 || v2<0) break; // 数据输入的终止条件
```

```
p=(NODE *)malloc(sizeof(NODE));
```

```
p->vertex=v2;
```

```
p->link=adjlist[v1].link; adjlist[v1].link=p; //插入在链表首部
```

```
p=(NODE *)malloc(sizeof(NODE));
```

```
p->vertex=v1;
```

```
p->link=adjlist[v2].link; adjlist[v2].link=p;
```

```
} return(num); // 返回图的结点数
```

```
}
```

在不同的存储结构下，实现各种操作的效率可能是不同的。所以在求解实际问题时，要根据求解问题所需操作，选择合适的存储结构。

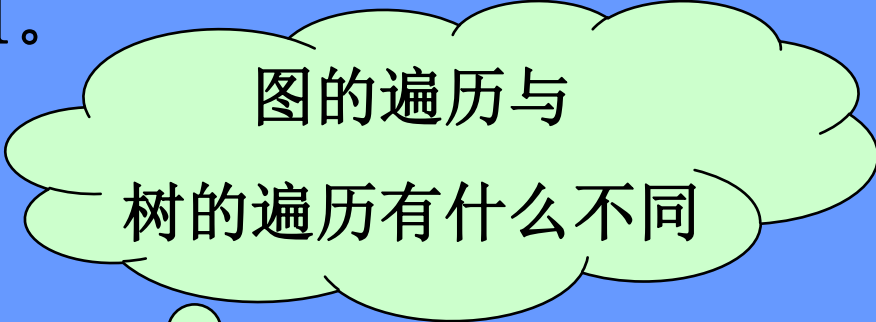


第七章 图


§ 7.3 图的遍历

- 一 深度优先遍历
- 二 广度优先遍历

图的遍历：从图的某顶点出发，访问图中所有顶点，并且每个顶点仅访问一次。在图中，访问部分顶点后，可能又沿着其他边回到已被访问过的顶点。为保证每一个顶点只被访问一次，必须对顶点进行标记，一般用一个辅助数组 `visit[MAX]` 作为对顶点的标记，当顶点 v_i 未被访问，`visit[i]` 值为0；当 v_i 已被访问，则 `visit[i]` 值为1。



图的遍历与
树的遍历有什么不同



有两种遍历方法（它们对无向图，有向图都适用）

深度优先遍历

广度优先遍历

一 深度优先遍历

从图中某顶点 v 出发:

- 1) 访问顶点 v ;
- 2) 依次从 v 的未被访问的邻接点出发, 继续对图进行深度优先遍历;

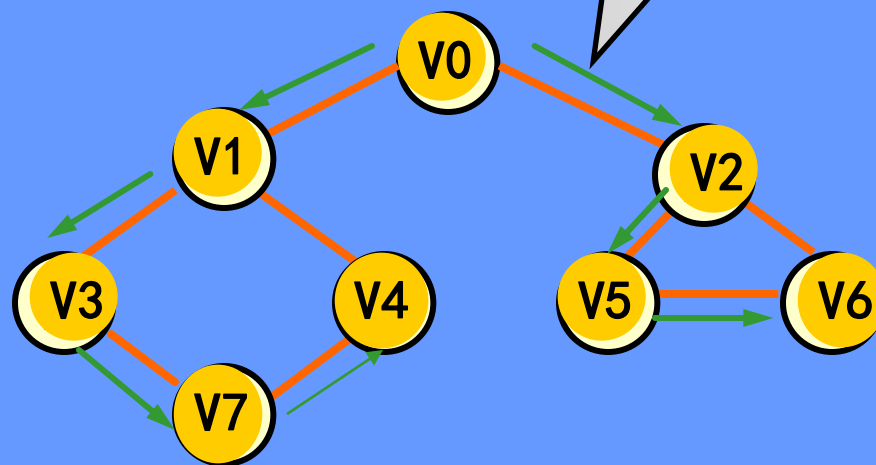
例 求图G以 V_0 起点的深度优先序列:

$V_0, V_1, V_3, V_7, V_4, V_2, V_5, V_6,$

$V_0, V_1, V_4, V_7, V_3, V_2, V_5, V_6$

这是序列 (1)
在遍历过程中
所经过的路径

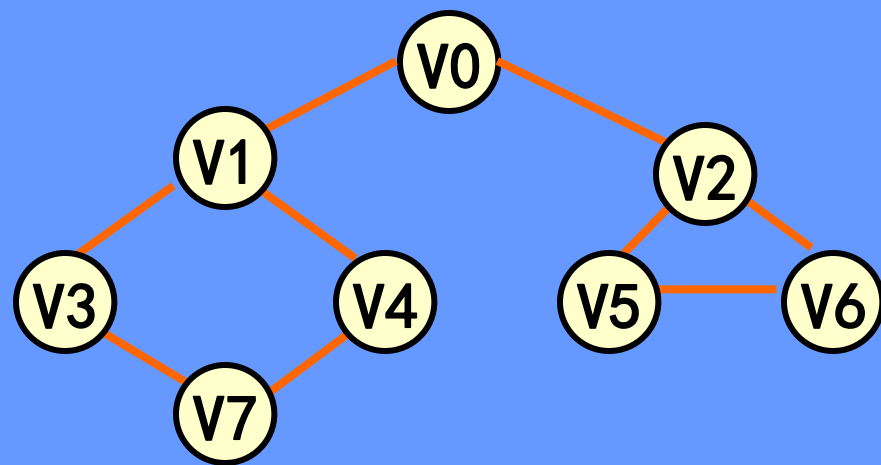
由于没有规定
访问邻接点的顺序,
深度优先序列不是唯一的



深度优先遍历

从图中某顶点 v 出发:

- (1) 访问顶点 v ;
- (2) 依次从 v 的未被访问的邻接点出发, 对图进行深度优先遍历;



先序遍历 (DLR)

若二叉树非空

- (1) 访问根结点;
- (2) 先序遍历左子树;
- (3) 先序遍历右子树;

如果将图顶点的邻接点
看作二叉树结点的左、右孩子
深度优先遍历与先序遍历
是不是很类似?

深度优先遍历算法

结点定义

```
typedef struct node
```

```
{ int vertex; // 边（弧）的另一顶点在数组中的位置
  struct node *link; // 指向下一条边（弧）结点的指针
}; typedef struct NODE;
```

NODE adjlist[MAX]; // 邻接点链表的头指针所对应的数组

| visit | |
|-------|---|
| 0 | 0 |
| 1 | 0 |
| 2 | 0 |
| 3 | 0 |
| 4 | 0 |
| | |
| m-1 | |

辅助数组

```
int visit[MAX];    //顶点标志数组, 全局变量
NODE * ptr[MAX];   //顶点链表指针数组
```

深度优先遍历算法

调用深度优先遍历算法的主函数

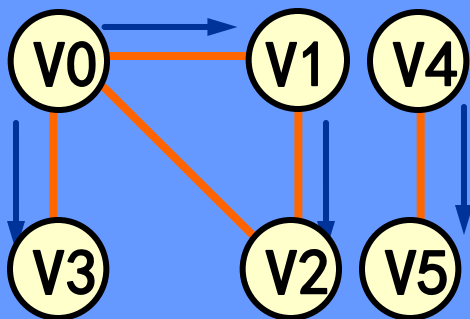
```
main( )  
{ NODE adjlist[ MAX];  
  int num;  
  num=create(adjlist ); /* 建立图G 的邻接表 */  
  depthfirst(adjlist, num); /* 调用对图G 深度优先搜索算法*/  
}
```

图G 的深度优先遍历算法

```

void depthfirst( NODE adjlist[], int num)
{ int i;
  for (i=0; i<num;i++)
  { ptr[i]=adjlist[i].link; //记住每个顶点链表的第一个结点的地址
    visit[i]=0;             //给每个结点一个未访问标记
  }
  for (i=0; i<num;i++)
    if (visit[i] == 0) dfs(i); //调用以顶点vi 为出发点的深度优先遍历图G 的算法
}

```

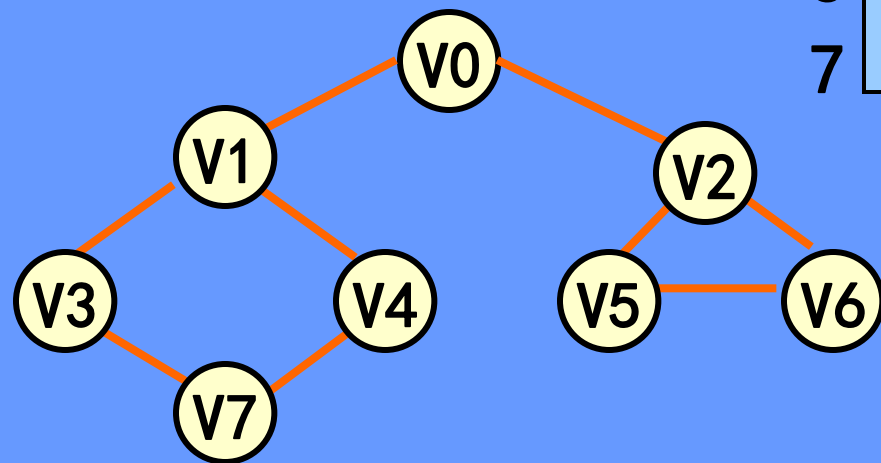


深度优先遍历

从第v个顶点出发，递归地深度优先遍历图G。

```
void dfs( int v)
{ int w;
  printf( “%d, “, v); visit[v]=1;  // 访问此结点
  while (ptr[v]! =NULL)
  { w= ptr[v]->vertex; 取结点的邻接顶点w
    if ( visit[w]= =0; ) dfs(w);
    ptr[v]=ptr[v]->link;  // 记住顶点v 的邻接顶点位置，
                           该邻接点在w之后
  }
}
```

§ 7.3 图的遍历



| | | | | | | | | |
|---|----|--|---|---|---|---|---|-------|
| 0 | V0 | | → | 1 | → | 2 | ^ | |
| 1 | V1 | | → | 3 | → | 4 | | → 0 ^ |
| 2 | V2 | | → | 5 | → | 6 | | → 0 ^ |
| 3 | V3 | | → | 7 | → | 1 | ^ | |
| 4 | V4 | | → | 7 | → | 1 | ^ | |
| 5 | V5 | | → | 6 | → | 2 | ^ | |
| 6 | V6 | | → | 2 | → | 5 | ^ | |
| 7 | V7 | | → | 3 | → | 4 | ^ | |

深度优先遍历算法

```
void dfs( int v)
```

```
{ int w;
```

```
    printf( “%d, “, v); visit[v]=1; // 访问此结点
```

```
    while (ptr[v]! =NULL)
```

```
    { w= ptr[v]->vertex; 取结点的邻接顶点w
```

```
        if ( visit[w]= =0; ) dfs(w);
```

```
        ptr[v]=ptr[v]->link; // 记住顶点v 的邻接点位置,
```

```
    } // 该邻接点在w之后
```

```
}
```

先序遍历递归算法

```
void prev (NODE *root)
```

```
{ if (root!=NULL)
```

```
{ printf(“%d,”, root->data); // 访问此结点
```

```
    prev(root->lch); // 访问孩子结点
```

```
    prev(root->rch);
```

```
}
```

```
}
```

比较

如果将图顶点的邻接点
看作二叉树结点的左、右孩子
深度优先遍历算法与
先序遍历算法
的结构是不是很像？

二 广度优先遍历 (类似于树的按层遍历)

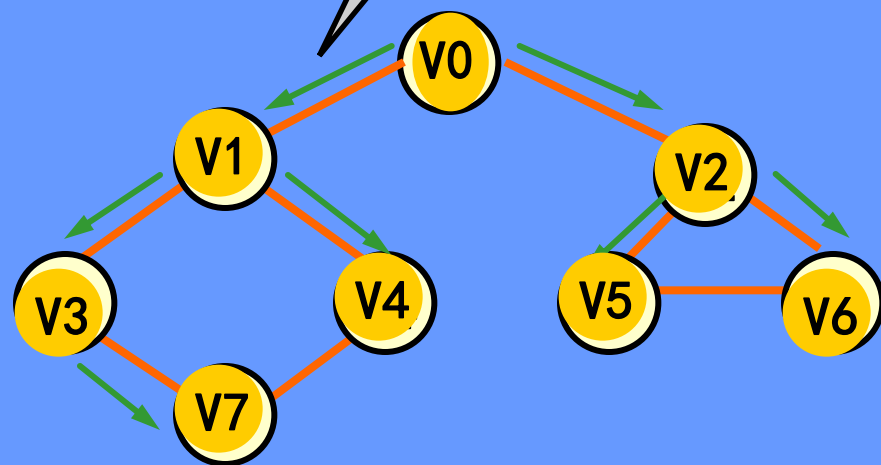
图中某未访问过的顶点 v_i 出发:

- 1) 访问顶点 v_i ;
- 2) 访问 v_i 的所有未被访问的邻接点 w_1, w_2, \dots ;
- 3) 依次从这些邻接点出发, 访问它们的所有未被访问的邻接点, 依此类推, 直到图中所有访问过的顶点的邻接点都被访问;

例 求图G 的以 V_0 起点的的广度优先序列

$V_0, V_1, V_2, V_3, V_4, V_5, V_6, V_7$

由于没有规定
访问邻接点的顺序,
广度优先序列不是唯一的

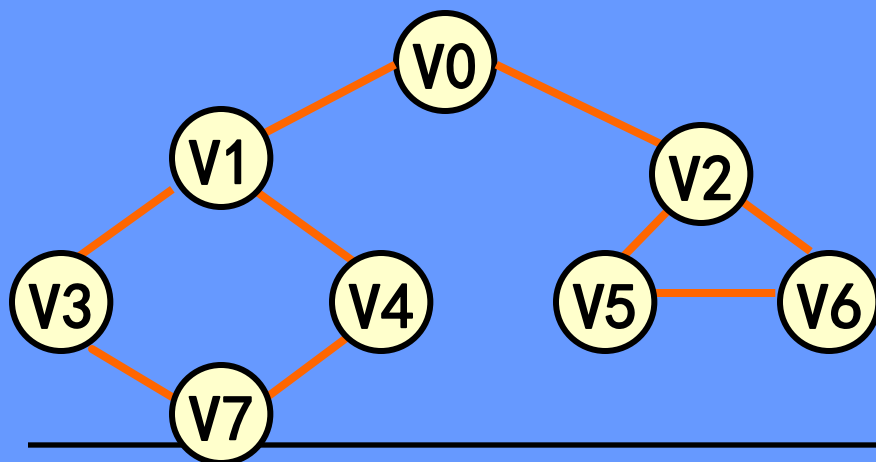


广度优先遍历算法

从图中某顶点 v_i 出发:

- 1) 访问顶点 v_i ; (容易实现)
- 2) 访问 v_i 的所有未被访问的邻接点 w_1, w_2, \dots, w_k ; (容易实现)
- 3) 依次从这些邻接点 (在步骤 2) 访问的顶点) 出发, 访问它们的所有未被访问的邻接点; 依此类推, 直到图中所有访问过的顶点的邻接点都被访问;

为实现 3), 需要保存在步骤(2)中访问的顶点, 而且访问这些顶点邻接点的顺序为: 先保存的顶点, 其邻接点先被访问。

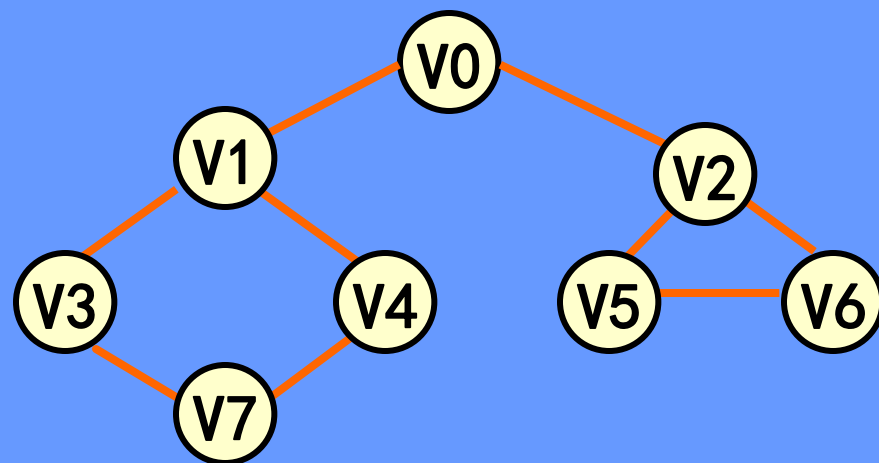
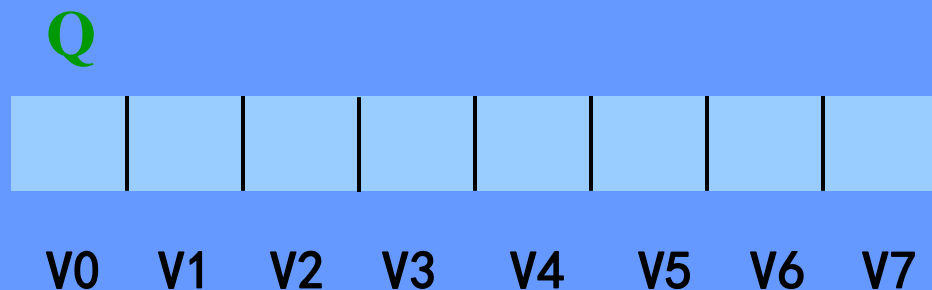


在广度优先遍历算法中,
需设置一队列Q,
保存已访问的顶点,
并控制遍历顶点的顺序。

广度优先遍历

从图中某顶点 v 出发：

- 1) 访问顶点 v ；
- 2) 访问 v 的所有未被访问的邻接点 w_1, w_2, \dots, w_k ；
- 3) 依次从这些邻接点出发，访问它们的所有未被访问的邻接点；
依此类推，直到图中所有访问过的顶点的邻接点都被访问；



```
int visit[MAX];
int queue[MAX], front=0, rear=0;
main( )
{ int num;
  num=create(adjlist);
  breadfirst(num); // 调用对图G 广度遍历图算法
}
```

在广度优先遍历算法中，
需设置一队列 Q，
保存已访问的顶点，
并控制遍历顶点的顺序

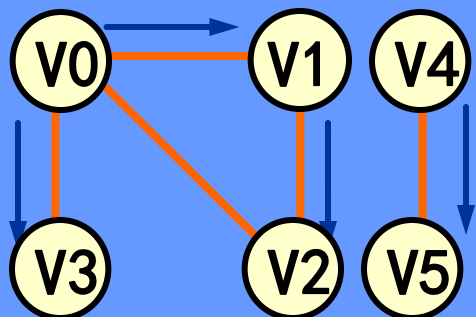
```
void breadfirst(int num)
{ int;
  for (i=0;i<num;i++) visit[i]=0; //给所有顶点一个未被访问标记
  for (i=0;i<num;i++)
    if(visit[i]==0) bft(i); // 调用以顶点vi 为出发点的是广度优先
                             遍历图G的算法
}
```

```

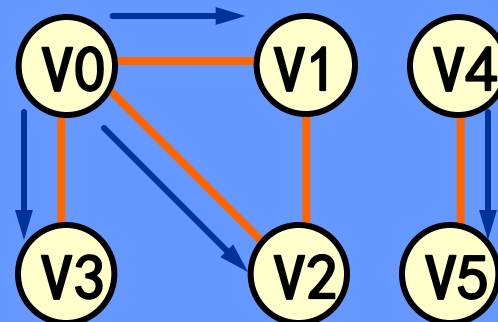
void bft(int v) //从v出发，广度优先遍历图G。
{
    int w; NODE *p;
    visit[v]=1; printf(“%d, ”, v);
    enter(v); // enter(v)为入队列函数
    while((v=leave())!=NULL); // leave()为出队列函数
    {
        p=adjlist[v].link; //p指向出队列顶点v的第一个邻接点
        while(p!=NULL)
        {
            w = p->vertex; //遍历v所指的整个链表
            if(visit[w]==0) //如果w 未被访问过
            {
                printf(“%d, ”, w); // 访问 w
                visit[w]=1; enter(w); //访问后，w 入队
            }
            p=p->link;
        }
    }
}

```

两种遍历的比较



深度优先遍历

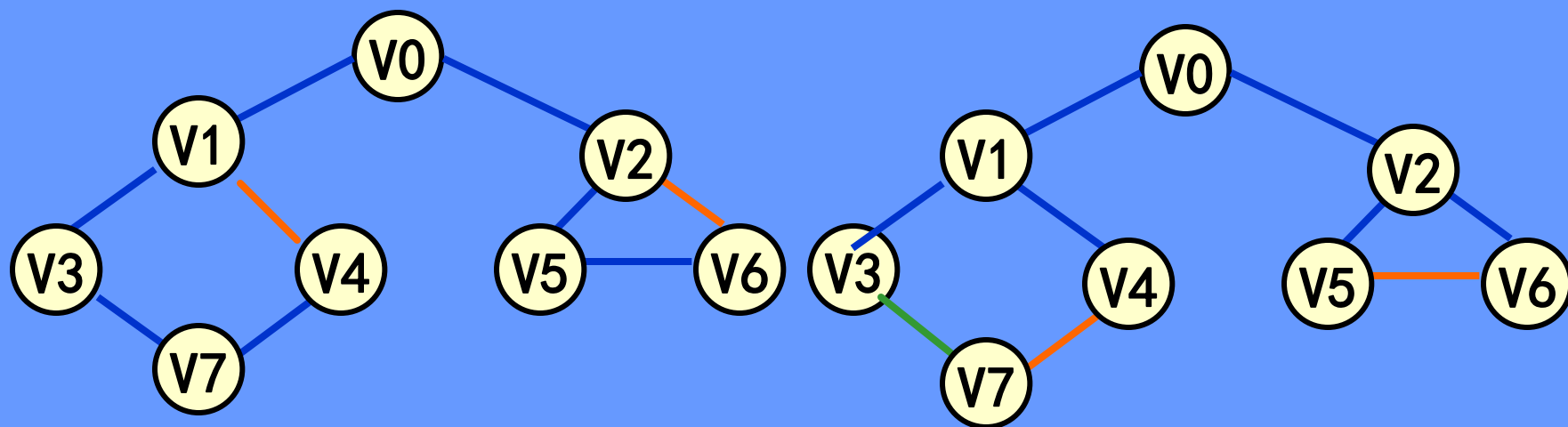


广度优先遍历

§ 7.4 生成树

1 无向图的生成树

生成树是一个连通图 G 的一个极小的连通子图。包含图 G 的所有顶点，但只有 $n-1$ 条边，并且是连通的。生成树可由遍历过程中所经过的边组成。深度优先遍历得到的生成树称为深度优先生成树；广度优先遍历得到的生成树称为广度优先生成树。



深度优先生成树

广度优先生成树

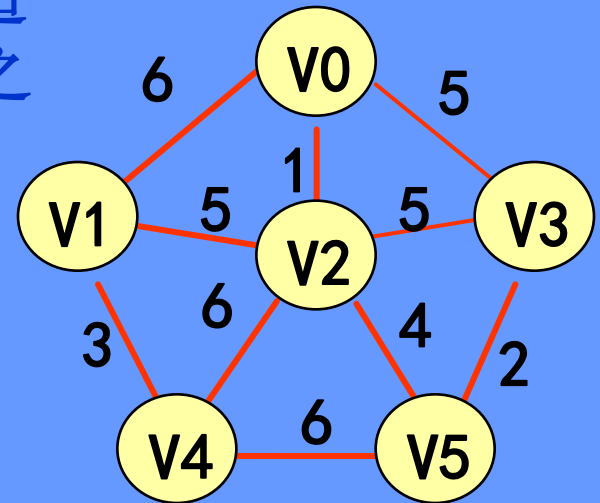
7.4.2 最小生成树（最小支撑树）

若有一个连通的无向图 G ，有 n 个顶点，并且它的边是有权值的。在 G 上构造生成树 G' ，最小生成树 $n-1$ 条边的权值之和最小的 G' 。

例

要在 n 个城市间建立交通网，要考虑的问题如何在保证 n 点连通的前提下最节省经费？

求解：连通6个城市且代价最小的交通线路？



如何求连通图的最小生成树？



二 普鲁姆算法

➤ 普鲁姆算法基本步骤

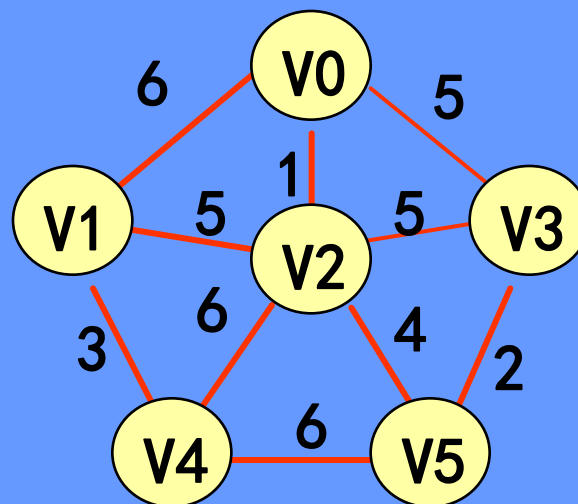
设 $G=(V, E)$ 为一个具有 n 个顶点的带权的连通网络, $T=(U, TE)$ 为构造的生成树。

(1) 初始时, $U=\{u_0\}$, $TE=\phi$;

(2) 在所有 $u \in U$ 、 $v \in V-U$ 的边 (u,v) 中选择一条权值最小的边, 不妨设为 (u,v) ;

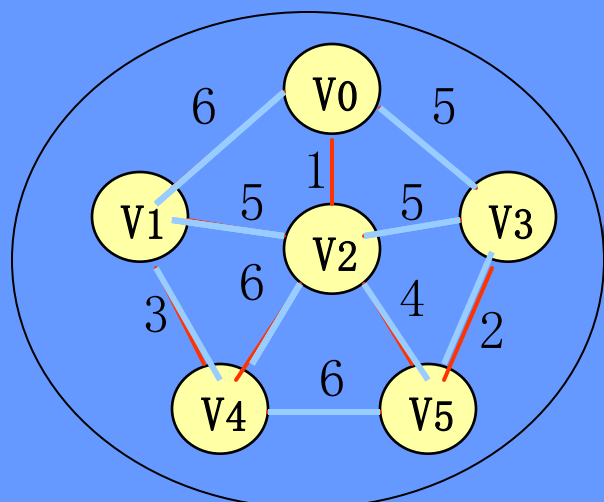
(3) (u,v) 加入 TE , 同时将 u 加入 U ;

(4) 重复(2)、(3), 直到 $U=V$ 为止;

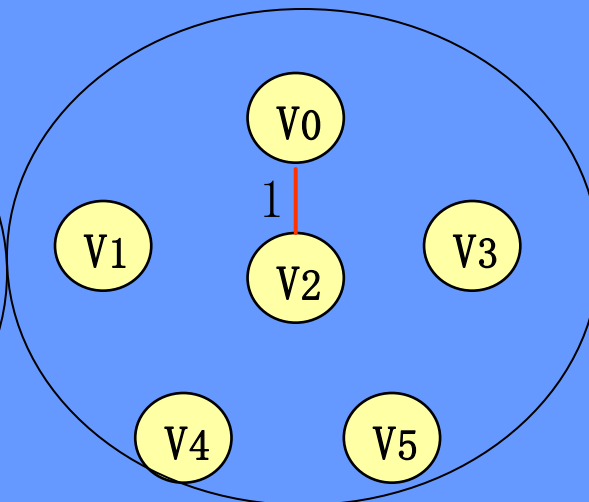


§ 7.4 最小的生成树

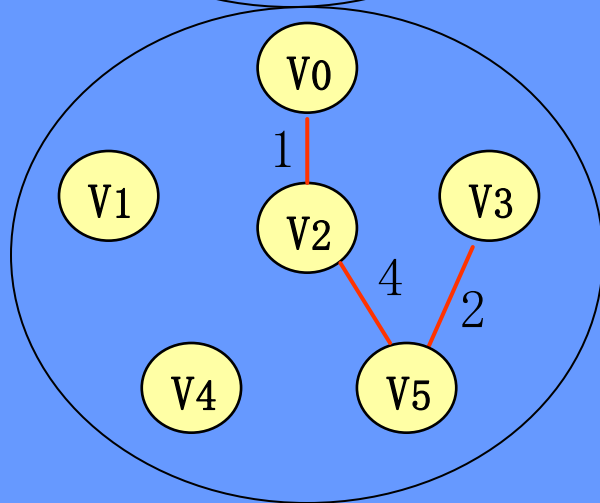
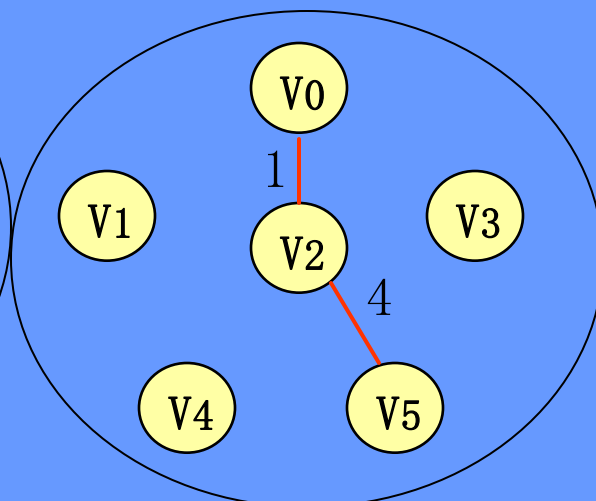
$U = \{ V_0 \}$



$U = \{ V_0, V_2 \}$

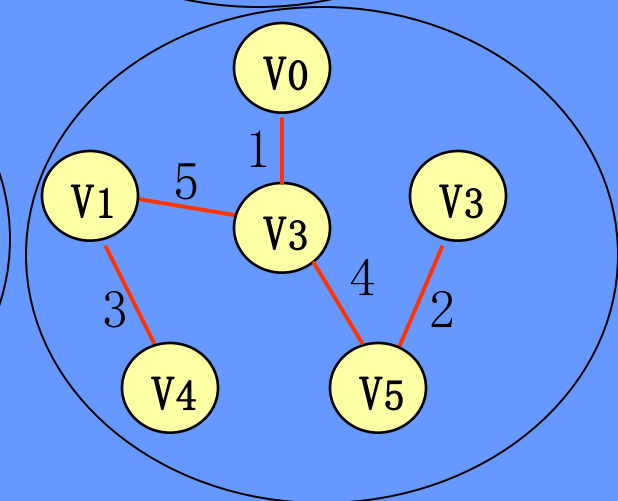
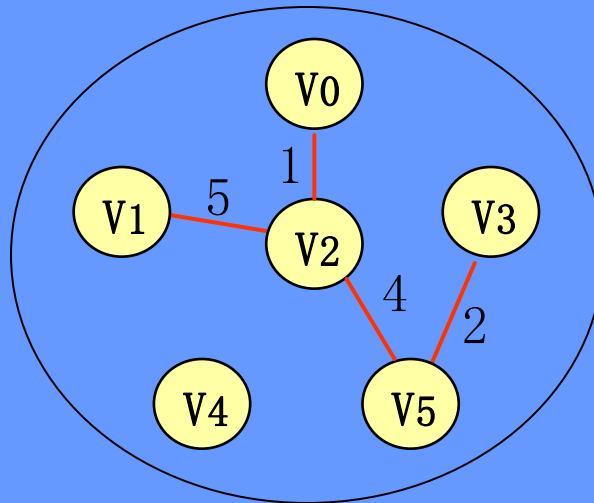


$U = \{ V_0, V_2, V_5 \}$



$U = \{ V_0, V_2, V_5, V_3 \}$

$U = \{ V_0, V_2, V_5, V_3, V_1 \}$



$U = \{ V_0, V_2, V_5, V_3, V_1, V_4 \}$

➤ 普鲁姆算法涉及的数据和操作:

数据: 无向连通网络

操作: 选择权值最小的边, 不妨设为 (u,v)
 (u,v) 加入 TE , u 加入 U

➤ 有关数据的存储结构

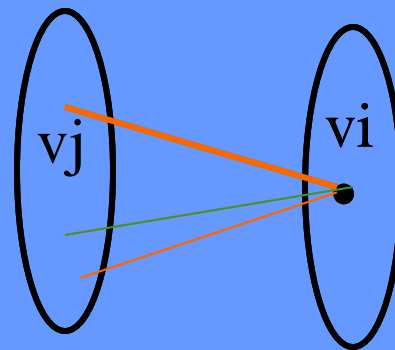
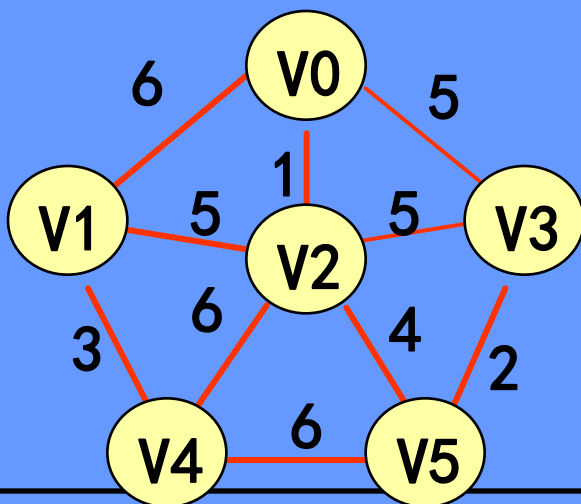
无向连通网络: G

为选择权值最小的边: 置一个一维数组: $closest[]$,

对 $i \in V-U$

$closest[i] = j$ ($j \in U$) (j, i) 是一条边, 且是 i 到 U 中各顶点 “权最小边”

$Lowcost[i]$: 用来保存连接 i 到 U 中各顶点 “权最小边” 的权。



对 $i \in V-U$

$\text{closest}[i] = j$ ($j \in U$) ((j, i) 是一条边, 且是 i 到 U 中各顶点 “权最小边”)

$\text{Lowcost}[i]$: 用来保存连接 i 到 U 中各顶点 “权最小边” 的权。

例

$U = \{v_0\}$

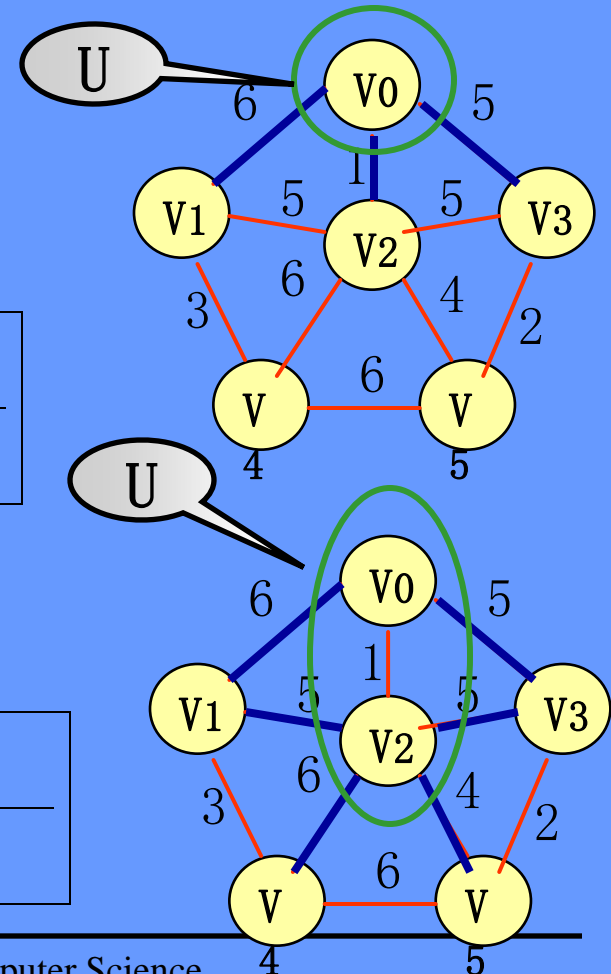
$V-U = \{v_1, v_2, v_3, v_4, v_5\}$

| | | | | | | | |
|---------------------|---|---|---|---|-----|-----|--|
| i | 0 | 1 | 2 | 3 | 4 | 5 | |
| $\text{closest}[i]$ | 0 | 0 | 0 | 0 | 0 | 0 | |
| $\text{lowcost}[i]$ | 0 | 6 | 1 | 5 | max | max | |

$U = \{v_0, v_2\}$

$V-U = \{v_1, v_3, v_4, v_5\}$

| | | | | | | | |
|---------------------|---|---|---|---|---|---|--|
| i | 0 | 1 | 2 | 3 | 4 | 5 | |
| $\text{closest}[i]$ | 0 | 2 | 0 | 0 | 2 | 2 | |
| $\text{lowcost}[i]$ | 0 | 5 | 0 | 5 | 6 | 4 | |



| i | 0 | 1 | 2 | 3 | 4 | 5 | U |
|--------------------|--------|--------|--------|--------|----------|----------|--------------------|
| closest lowcost | 0 0 | 0 6 | 0 1 | 0 5 | 0 max | 0 max | {0} |
| closest lowcost | 0 0 | 2 5 | 0 0 | 0 5 | 2 6 | 2 4 | {0, 2} |
| closest lowcost | 0 0 | 2 5 | 0 0 | 5 2 | 2 6 | 2 0 | {0, 2, 5} |
| closest lowcost | 0 0 | 2 5 | 0 0 | 5 0 | 2 6 | 2 0 | {0, 2, 5, 3} |
| closest lowcost | 0 0 | 2 0 | 0 0 | 5 0 | 1 3 | 2 0 | {0, 2, 5, 3, 1} |
| closest lowcost | 0 0 | 2 0 | 0 0 | 5 0 | 1 0 | 2 0 | {0, 2, 5, 3, 1, 4} |

普里姆算法

图采用邻接矩阵表示，邻接矩阵所对应的二维数组是 $\text{cost}[\text{MAX}][\text{MAX}]$ ，则

- (1) 初始化($U=\{0\}$), $\text{closest}[i]=0$; $\text{lowcost}[i]=\text{cost}[0][i]$; $\text{lowcost}[0]=0$;
($i=1, 2, 3, \dots, n-1$;))
- (2) 每次扫描数组 $\text{lowcost}[i]$ ，找出值最小且不为0的 $\text{lowcost}[k]$ ，得到最小生成树的一条边 ($\text{closest}[k], k$), 将其输出。
- (3) 令 $\text{lowcost}[k]=0$, 将 k 并入 U 中
- (4) 修改数组 $\text{closest}[i]$ 和 $\text{lowcost}[i]$ ($\text{lowcost}[i] \neq 0$ 及 $i \in V-U$)
- (5) 重复 (2) (3) (4) , 直到 $U=V$ (或循环 $n-1$ 次) 结束

用普里姆算法

```

viud PRIM( int cost[][N], int start_v )
{ int closest[N], lowcost[N], i, j, k, min;
  for (i=0; j<N; i++)
    { lowcost[i]=cost[start_v][i];
      closest[i]=start_v;
    }
  for (i=1; i<N; i++) // 循环n-1次, 每次求出最小生成树的一条边
    { min=MAX;
      for (j=0; j<N; j++)
        if (lowcost[j]!=0 && lowcost[j]<min)
          {min= lowcost[j];k=j;} // 找出值最小的 lowcost[k]
      printf(“%d,%d:%d\n”, closest[k],k, lowcost[k]);
      lowcost[k]=0; // 将 k 并入U 中
      for (j=0; j<N; j++) // 修改 lowcost[ ] 和closest[ ]
        if (cost[k][j]!=0 && cost[k][j]< lowcost[j])
          {lowcost[j]=cost[k][j];closest[j]=k;}
    }
}

```

三 克鲁斯卡尔算法

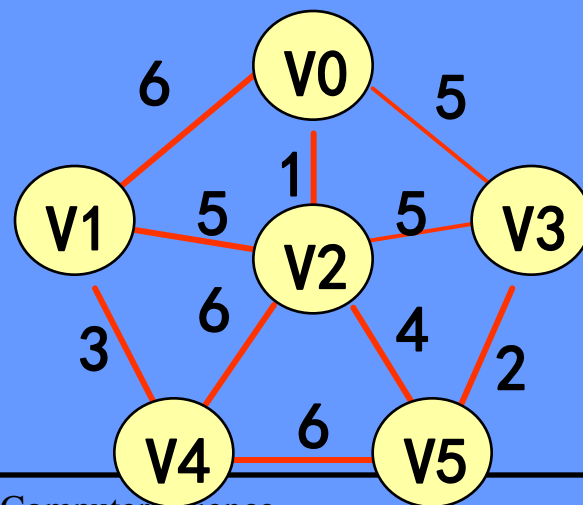
➤ 基本步骤

设 $G=(V, E)$ 为一个具有 n 个顶点的带权的连通网络，最小生成树的初始状态为有 n 个顶点但无边的非连通图 $T=(V, \phi)$ 。

(1) 将 E 中的边按权值的递增顺序排序，选择权值最小的边，若与相关的顶点落在 T 中不同的连通分量上，则将其加入 T 中，否则，将其弃舍。

(2) 循环至所有顶点在同一的连通分量上。

如何识别一条边所相关的顶点是否落在同一个连通分量上？可将一个连通分量的所有顶点看成一个集合，当从 E 中取出一条边 (x_i, x_j) 时，若 x_i, x_j 在同一集合 u 中，则将该边弃舍；否则，则将该边加入到 T 中，并将 x_j 所在的集合 v 并入集合 u 中。



需要引入辅助数组 **sets[]**

(1) 初始化：图 **G** 中的 **n** 个顶点，构成 **n** 个连通分量，顶点 **xi** 对应的连通分量用集合 **i** 表示，所以 **sets[i]=i**，即表示第 **i** 个顶点在集合 **i** 中。

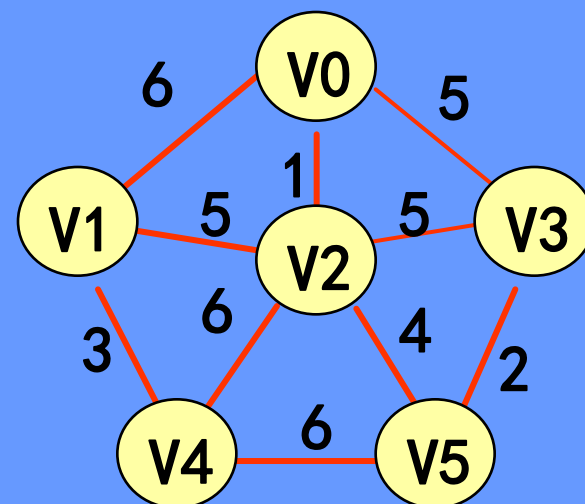
(2) 依次取出 **E** 中的边（按边权值递增顺序），设取出的边为（**xi, xj**）。

(3) 判断：若 **sets[i]= sets[j]**，则表示 **xi** 和 **xj** 在同一集合中，返回（2）；否则，转到（4）。

(4) 将边（**xi, xj**）并入 **T**，同时将 **xj** 所在的集合 **v**（与 **xj** 连通的顶点）并入 **xi** 所在的集合 **u**（与 **xi** 连通的顶点），即：由 **v=sets[j]** 和 **u =sets[i]**，扫描辅助数组 **sets[]**，若 **sets[k]=v**，则令 **sets[k]=u**。返回（2）。

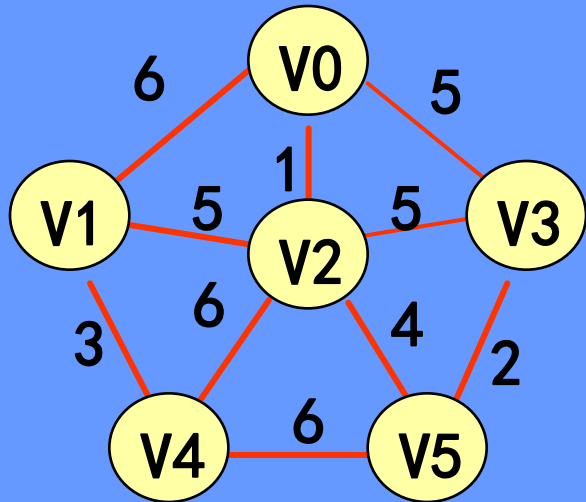
(5) 重复（2）、（3）、（4），取出 **n-1** 条边。

| | | | | | | |
|------|---|---|---|---|---|---|
| 下标 | 0 | 1 | 2 | 3 | 4 | 5 |
| sets | 0 | 1 | 2 | 3 | 4 | 5 |



下标 0 1 2 3 4 5

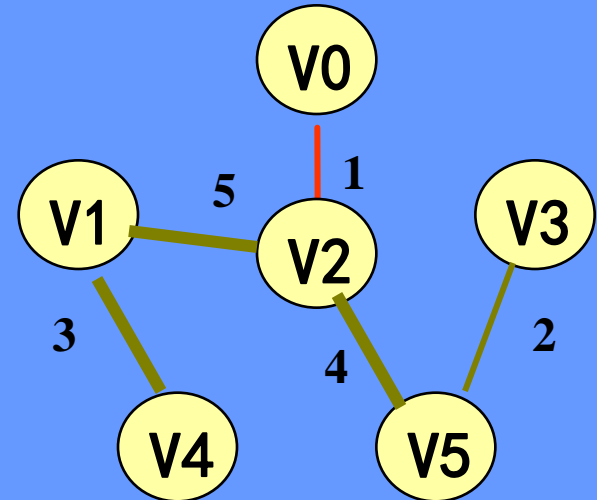
sets 0 | 1 | 2 | 3 | 4 | 5 |



。

下标 0 1 2 3 4 5

sets 1 | 1 | 1 | 1 | 1 | 1 |



克鲁斯卡尔算法中，数据结构定义为：

```
struct node
```

```
{ int begin, end; // 边的相关顶点编号  
  int cost; };    //边的权值
```

```
typedef struct node EDGE;
```

```
EDGE edges[MAX]; //存放边的数组
```

```
int num;          //数组中实际存放的边数
```

```
int kruskal (EDGE edges[ ], int num)
```

```
{ int sets[N], t, i, j, k, u, v;
```

```
  for(i=0; i<N; i++) sets[i]=i; //初始化
```

```
  k=0; t=0;
```

克鲁斯卡尔算法:

```

while ((t<N)&&(k<num))
{
    i=edges[k].begin; j=edges[k].end;k++; //按顺序取出一条边
    u=sets[i]; v=sets[j];    // xi 在集合 u 中, xj在集合 v 中
    if (u!=v)                // xi, xj 不在同一集合中
    {
        printf(“%d, %d: %d\n”, u, v, edges[k].cost);
        t++;
        for (i=0; i<N; i++)
            if(sets[i] != v) set[i]=u; // xi在集合的v并入在集合的u
    }
}
if (t == N-1) return(1);
else return(0);
}

```

中

§ 7.5 最短路径

一 问题的提出

交通咨询系统、通讯网、计算机网络常要寻找两结点间最短路径

交通咨询系统：A 到 B 最短路径

计算机网 发送Email节省费用 A到B沿最短路径传送

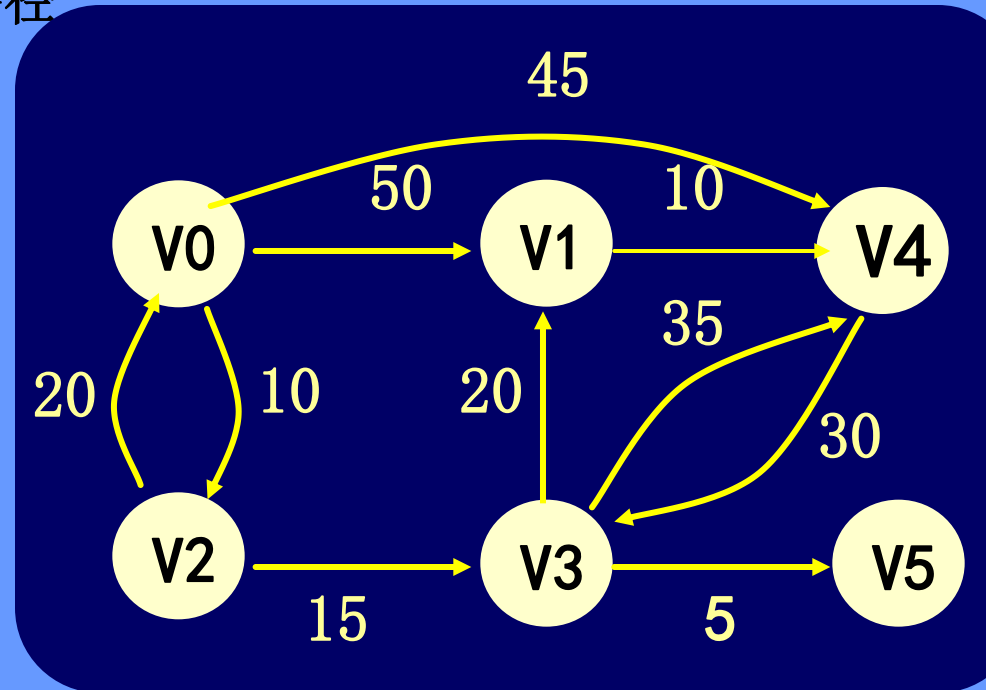
路径长度：路径上边数

路径上边的权值之和

最短路径：两结点间权值之和最小的路径

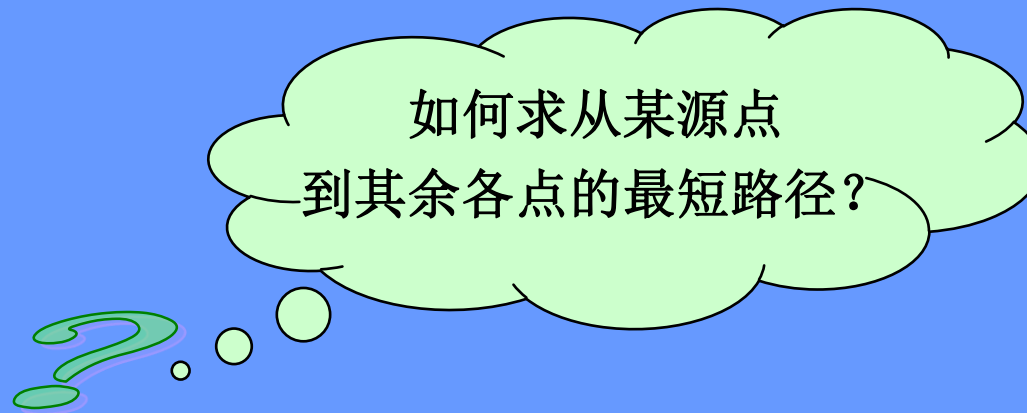
例：求V0到V4最短路径

| | |
|----------------|----|
| V0到V4 路径：V0 V4 | 45 |
| V0 V1 V4 | 60 |
| V0 V2 V3 V4 | 60 |
| V0 V2 V3 V1 V4 | 55 |



二 从某源点到其余各点的最短路径

带权值的有向图的单源最短路径问题



1 迪杰斯特拉算法 (Dijkstra)

1) Dijkstra算法的基本思想

按路径长度递增顺序求最短路径算法。与求最小生成树的普里姆算法类似

2) Dijkstra 算法的基本步骤

设 V_0 是起始源点, $U =$ 已求得最短路径终点集合。 $V-U =$ 未确定最短路径的顶点的集合

初始时 $U = \{V_0\}$

1) 长度最短的最短路径是边数为1的长度最小的路径。

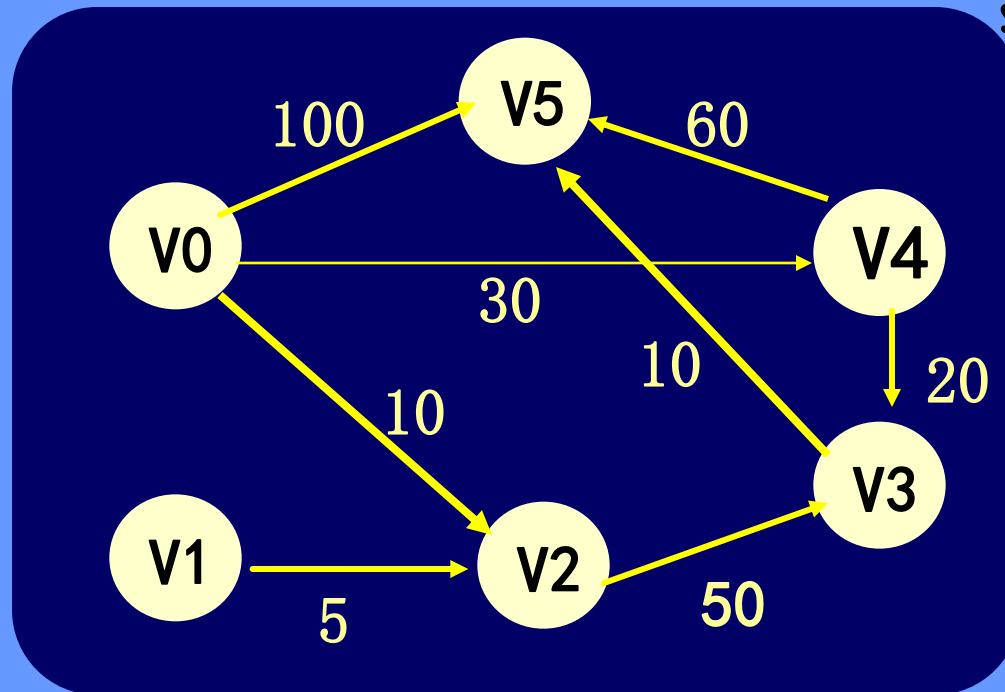
2) 下一条长度最短的路径:

① $V_i \in V - U$, 先求出 V_0 到 V_i 中间只经 U 中结点的最短路径;

② 上述最短路径中长度最小者即为下一条长度最短的路径;

③ 将所求最短路径的终点加入 U 中;

3) 重复2) 直到求出所有的最短路径



| 始点 | 终点 | 最短路径 | 路径长度 |
|----|----|------------------|------|
| V0 | V1 | 无 | |
| | V2 | (V0, V2) | 10 |
| | V3 | (V0, V4, V3) | 50 |
| | V4 | (V0, V4) | 30 |
| | V5 | (V0, V4, V3, V5) | 60 |

➤迪杰斯特拉算法涉及的数据和操作:

➤有关数据的存储结构

※ 有向连通网络: G , 采用带权邻接矩阵 $cost[][]$ 存储

➤具体步骤:

(1) 初始 $U=\{v_0\}$, 用辅助数组 $dist[N]$ 。对已经找到最短路径终点的顶点 $v_i (i \in U)$, v_i 所对应的数组分量 $dist[i]$ 的值为负数; 对从 v_0 出发, 尚未确定为最短路径终点的顶点 $v_j (j \in V - U)$, v_j 所对应的数组分量 $dist[j]$ 的值为 W_{v_j} , 而 W_{v_j} 为从 v_0 出发, 考虑途经已确定为终点的顶点, 到达 $v_j (j \in V - U)$ 的最短路径。初始时, 对 $j \in V - U$, 有 $dist[j]=cost[v][j]$; 而对 $U=\{v\}$, 则有 $dist[v]=-cost[v][v]$ 。

(2) 扫描 $dist[]$ 数组, 找出非0、非负且最小的 $dist[j] (j \in V - U)$, 即从 v_0 出发到 $v_j (j \in V - U)$ 的路径是最短的。

(3) v_j 并入 U , 则 $dist[j]=-dist[j]$ 。

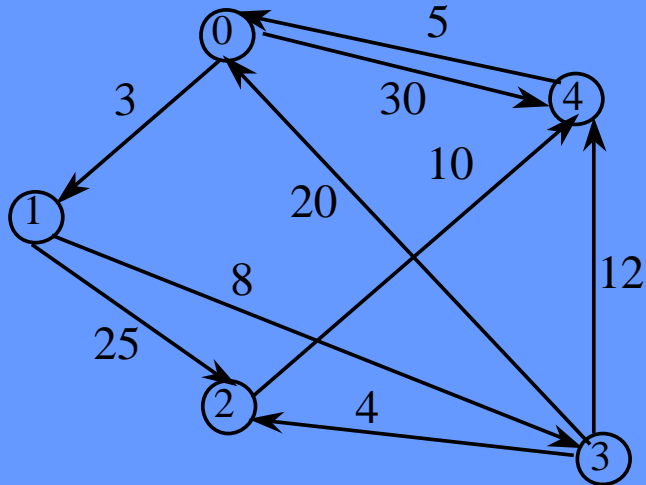
(4) 调整 $dist[k] (k \in V - U)$, 考虑从 v_0 出发, 途经 v_j 到达 v_k 是否更短。比较: 若 $-dist[j]+cost[j][k]<dist[k]$ 则 $dist[k]=-dist[j]+cost[j][k]$

(5) 重复 (2) (3) (4)。共 $n-1$ 次。

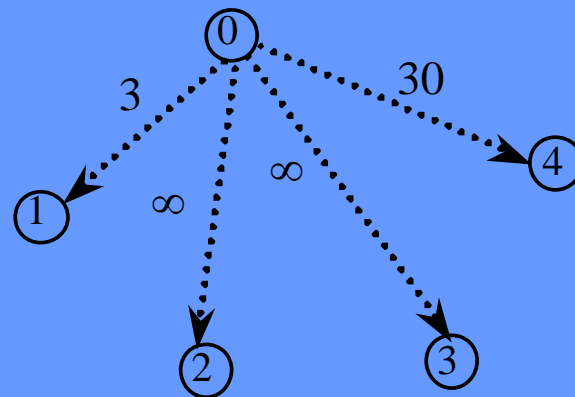
§ 7.6 最短路径

| 求解步骤 辅助数组 | i=1 | i=2 | i=3 | i=4 | i=5 |
|------------------|-----------------|--------------------|--------------------|------------------------|-------------------------|
| dist[1] 路径终点1 | max | max | max | max | max |
| dist[2] 路径终点2 | 10 (V0, V2) | -10 (V0, V2) | -10 (V0, v2) | -10 (V0, V2) | -10 (V0, V2) |
| dist[3] 路径终点3 | max | 60 (V0, V2, V3) | 50 (V0, v4, v3) | -50 (V0, V4, v3) | -50 (V0, V4, v3) |
| dist[4] 路径终点4 | 30 (v0, v4) | 30 (V0, V4) | -30 (V0, V4) | -30 (V0, V4) | -30 (V0, V4) |
| dist[5] 路径终点5 | 100 (V0, V5) | 100 (V0, V5) | 90 (V0, V4, v5) | 60 (V0, v4, v3, V5) | -60 (V0, v4, v3, V5) |
| 最短路径 的终点 | V2 | V4 | V3 | V5 | |
| U: {V} | {V0, V2} | {V0, V2, V4} | {V0, V2, V4, V3} | {V0, V2, V4, V3, V5} | |

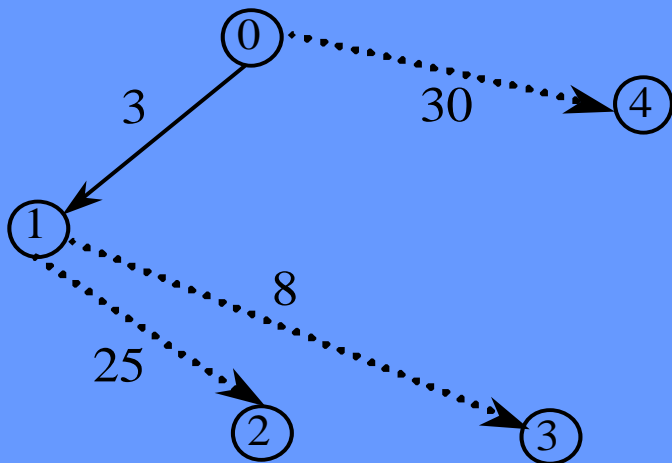
迪杰斯特拉算法的求解过程



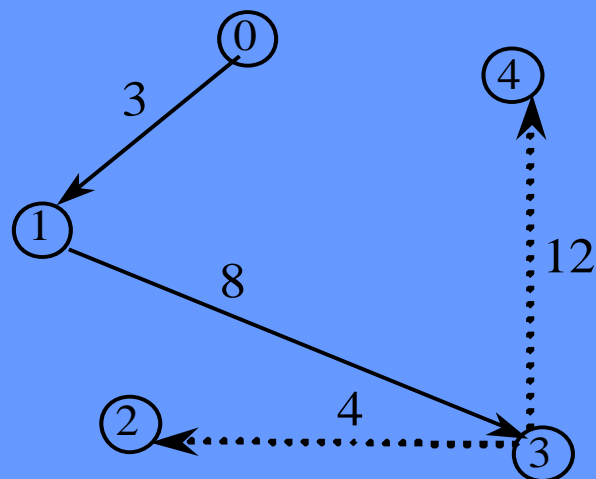
(a) 一个有向网点



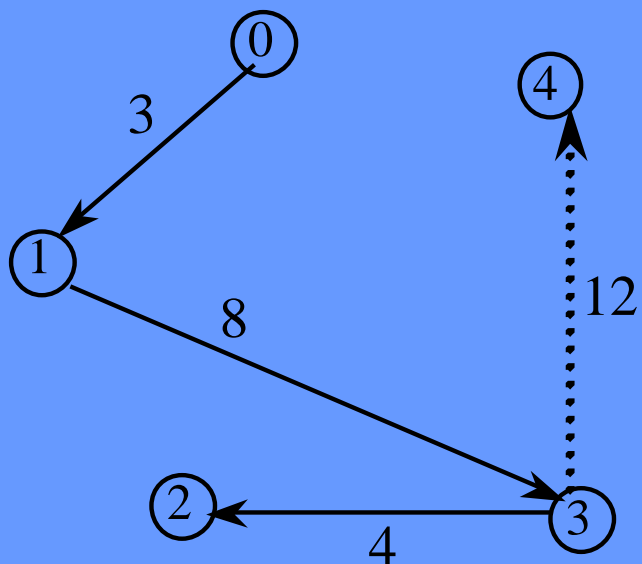
(b) 源点 0 到其它顶点的初始距离



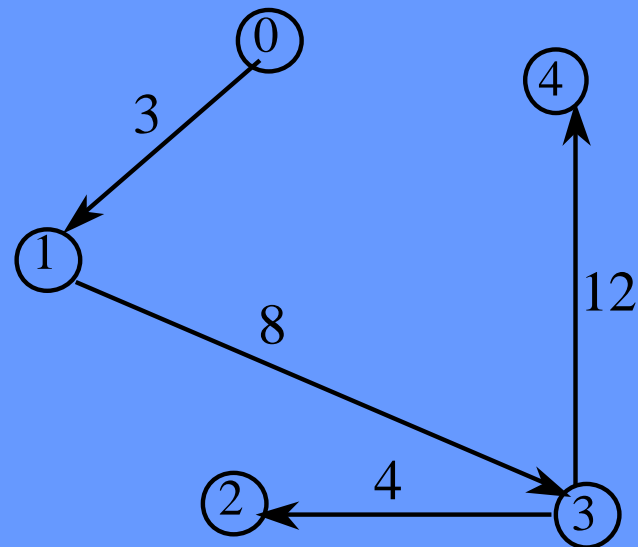
(c) 第一次求得的结果



(d) 第二次求得的结果



(e) 第三次求得的结果



(f) 第四次求得的结果

图 7-27 迪杰斯特拉算法求最短路径过程及结果

Dijkstra算法

```

void dijkstra( int cost[][N], int v ){
{ int dist[N],i,j,w
  for (i=0; i<N; i++) dist[i]=cost[v][i]; //初始化
  dist[v]=-dist[v];
  for( i=0; i<N; i++)
  { j=mincost(dist); //找非0、非负且最小的dist[j]
    if( j==0);break;
    dist[j]=-dist[j]; // vj并入U中
    for(k=0;k<N;k++) // 调整dist[k]
      if(dist[j]>0) // vk是尚未到达的终点
        if(-dist[j]+cost[j][k]<dist[k])
          dist[k]=-dist[j]+cost[j][k]; //途经vj到达vk的距离更短
    }
  for ( i=0; i<N; i++)
    if(dist[j]<0)printf(“%d, %d: %d\n”,v,i,-dist[i]);
}
}

```

```
int mincost( int dist[ ])
// 在V-U 集合中找顶点j, dist[j]是dist[ ]中的最小值
{ int i, min, j;
  min=MAX;j=0;
  for(i=0;i<N;i++)
    if(dist[i]>=0&& dist[i]<min)
      {min=dist[i];j=i; }
  return(j);
}
```

7.5.2所有顶点对之间的最短路径

1. 顶点对之间的最短路径概念

所有顶点对之间的最短路径是指：对于给定的有向网 $G=(V,E)$,要对 G 中任意一对顶点有序对 u 、 v ($u \neq v$),找出 u 到 v 的最短距离和 v 到 u 的最短距离。

解决此问题的一个有效方法是:轮流以每一个顶点为源点,重复执行迪杰斯特拉算法 n 次,即可求得每一对顶点之间的最短路径,总的时间复杂度为 $O(n^3)$ 。

下面将介绍用弗洛伊德(Floyd)算法来实现此功能,时间复杂度仍为 $O(n^3)$,但该方法比调用 n 次迪杰斯特拉方法更直观一些。

2. 弗洛伊德算法的基本思想

弗洛伊德算法仍然使用前面定义的图的邻接矩阵 $\text{cost}[N][N]$ 来存储带权有向图。算法的基本思想是：

设置一个 $N \times N$ 的矩阵 $A[N][N]$ ，其中除对角线的元素都等于0外，其它元素 $A[i][j]$ 的值表示顶点 i 到顶点 j 的最短路径长度，运算步骤为：

开始时，以任意两个顶点之间的有向边的权值作为路径长度，没有有向边时，路径长度为 ∞ ，此时， $A[i][j] = \text{cost}[i][j]$ ，

以后逐步尝试在原路径中加入其它顶点作为中间顶点，如果增加中间顶点后，得到的路径比原来的路径长度减少了，则以此新路径代替原路径，修改矩阵元素。具体做法为：

第一步，让所有边上加入中间顶点0，取 $A[i][j]$ 与 $A[i][0] + A[0][j]$ 中较小的值作 $A[i][j]$ 的值。

第二步，让所有边上加入中间顶点1，取 $A[i][j]$ 与 $A[i][1]+A[1][j]$ 中较小的值，完成后得到 $A[i][j]$...，如此进行下去，当第n步完成后，得到 $A[i][j]$ ，即为我们所求结果， $A[i][j]$ 表示顶点i到顶点j的最短距离。

因此，弗洛伊德算法可以描述为：

$$A^{(0)}[i][j] = \text{cost}[i][j]; \quad // \text{cost 为图的邻接矩阵}$$

$$A^{(k)}[i][j] = \min\{A^{(k-1)}[i][j], A^{(k-1)}[i][k] + A^{(k-1)}[k][j]\}$$

其中 $k=0,1,2,\dots, n-1$

3. 弗洛伊德算法实现

```
Void floyd (int cost[ ][N] )
```

```
{ int a[N][N], i, j, k;
```

```
  for ( i=0; i<N; i++)
```

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

```
      a[i][j]=cost[i][j]; //初始化
```

```
  for ( k=0; k<N; k++) //考虑途经 vk(k=0,1,..n-1)
```

```
    for ( i =0; i<N;i++)
```

```
      for (j=0; j<N; j++) //图中每对顶点
```

```
        if (a[i][k]+a[k][j]<a[i][j])
```

```
          a[i][j]=a[i][k]+a[k][j];
```

```
for (i=0;i<N; i++)           //输出路径长度及路径
    for (j=0; j<N; j++)
        printf( “(%d, %d):%d”,i, j, a[i][j]);
}
```

算法的时间复杂度为 $O(N^3)$, N 为图的顶点数。

第七章 图



§ 7.6 有向无环图的应用

一 AOV网、AOE网

二 拓扑排序

三 关键路径

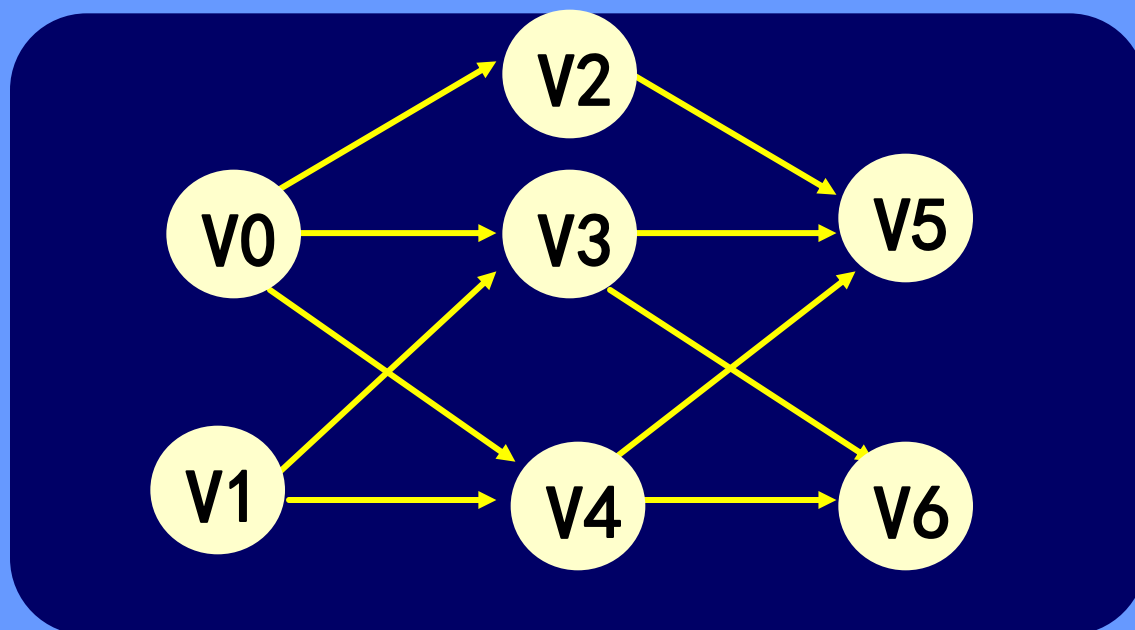
有向无环图：没有回路的有向图

应用：工程流程、生产过程中各道工序的流程、程序流程、课程的流程

AOV网 (activity on vertex net)

用顶点表示活动，边表示活动的顺序关系的有向图称为AOV网

例 某工程可分为7个子工程，若用顶点表示子工程（也称活动），用弧表示子工程间的顺序关系。工程流程可用如下AOV网表示



二 AOV网与拓扑排序

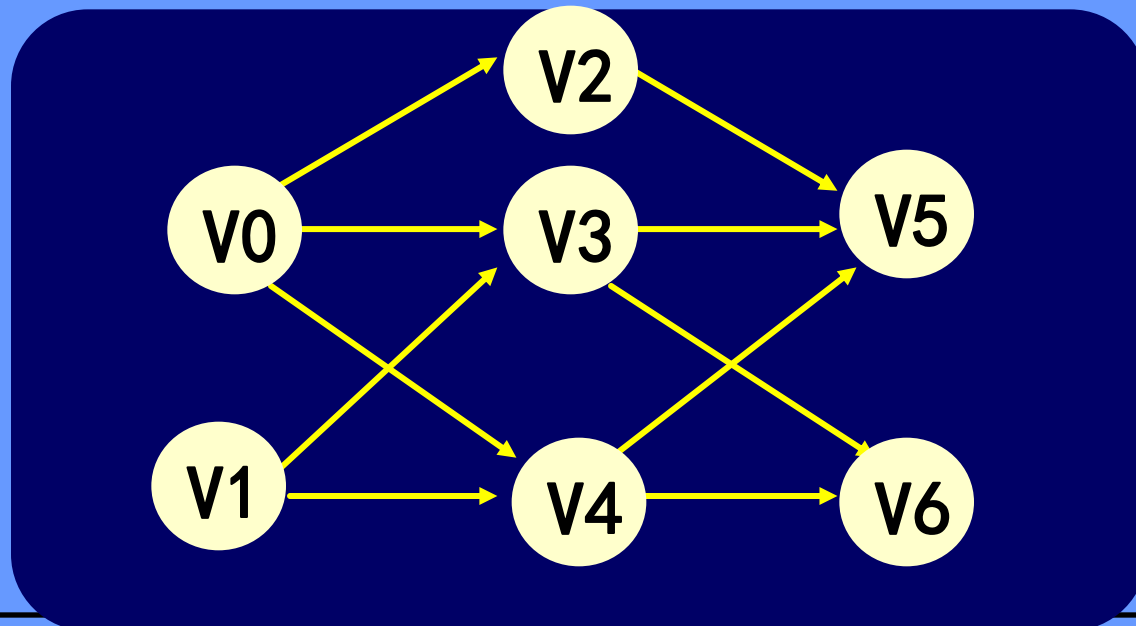
1 拓扑排序

对工程问题，人们至少关心如下两类问题：

- 1) 工程能否顺序进行，即工程流程是否“合理”
- 2) 完成整项工程至少需要多少时间，哪些子工程是影响工程进度的关键子工程

为求解工程流程是否“合理”，通常用AOV网的有向图表示工程流程

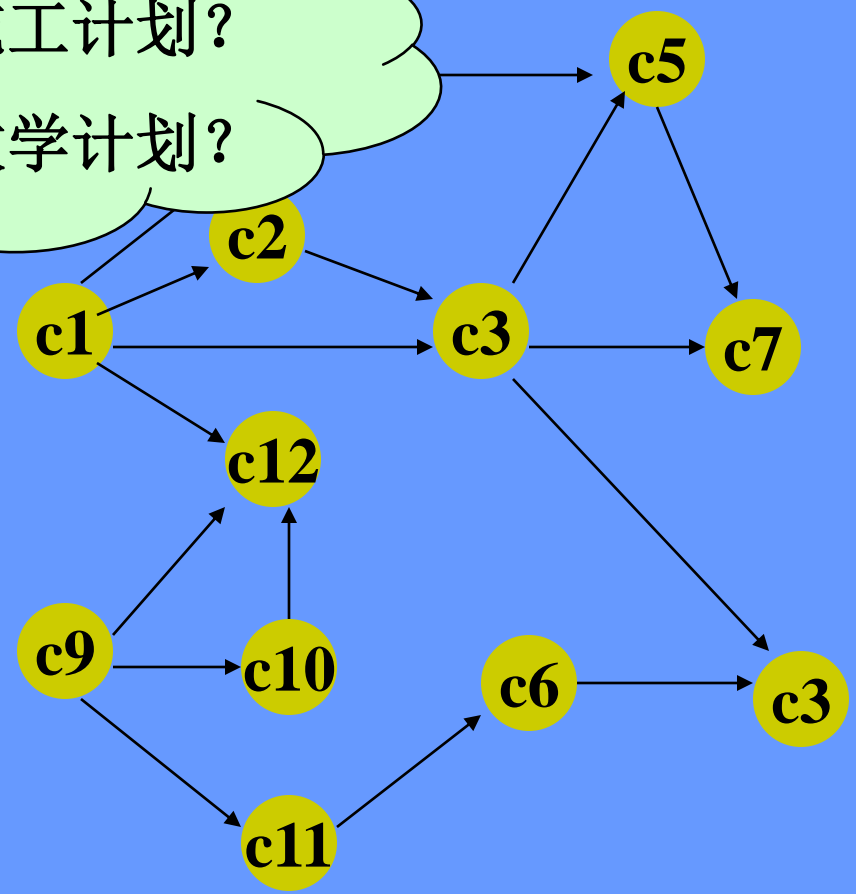
例1 某工程可分为V0、V1、V2、V3、V4、V5、V6 7个子工程， 工程流程可用如下AOV网表示。其中顶点：表示子工程（也称活动）， 弧：表示子工程间的顺序关系。



某校计算机专业课程流程可AOV网表示。其中顶点：表示课程（也称活动），弧：表示课程间的先修关系；

| 课程编号 | 课程名称 | |
|------|-------|--------------|
| c1 | 程序设计 | |
| c2 | 离散数学 | c1 |
| c3 | 数据结构 | c1, c2 |
| c4 | 汇编语言 | c1 |
| c5 | 算法分析 | c3, c4 |
| c6 | 计算机体系 | c11 |
| c7 | 编译方法 | c5, c3 |
| C8 | 操作系统 | c3, c6 |
| c9 | 高等数学 | 无 |
| c10 | 线性代数 | c9 |
| c11 | 电子电路 | c9 |
| c12 | 数值分析 | c9, c10, c11 |

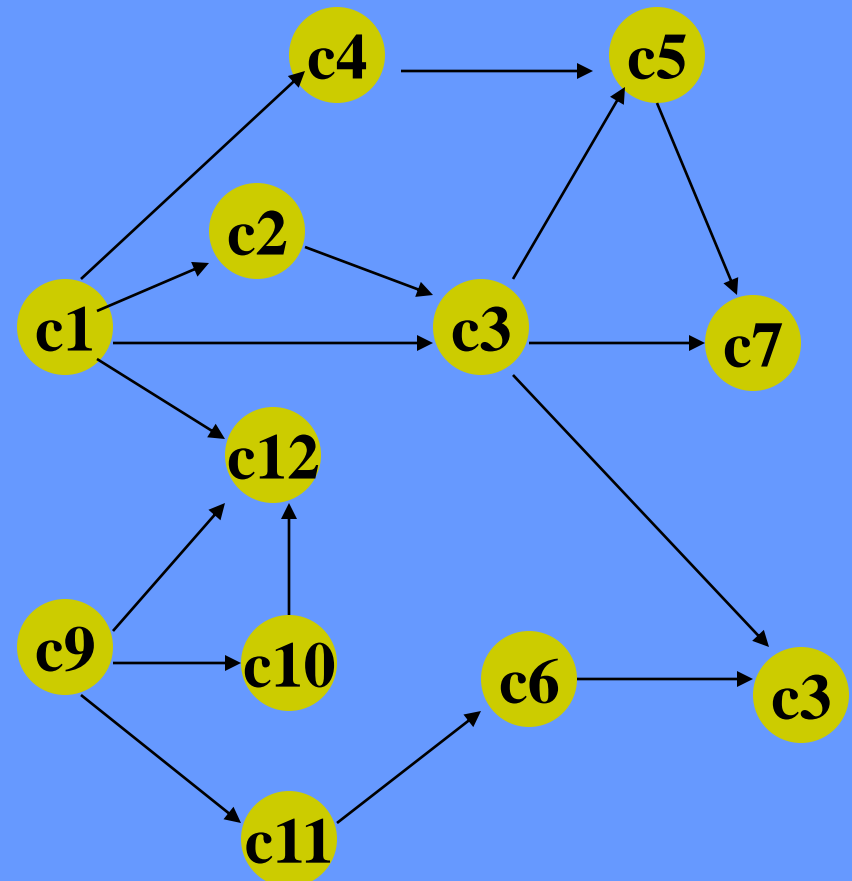
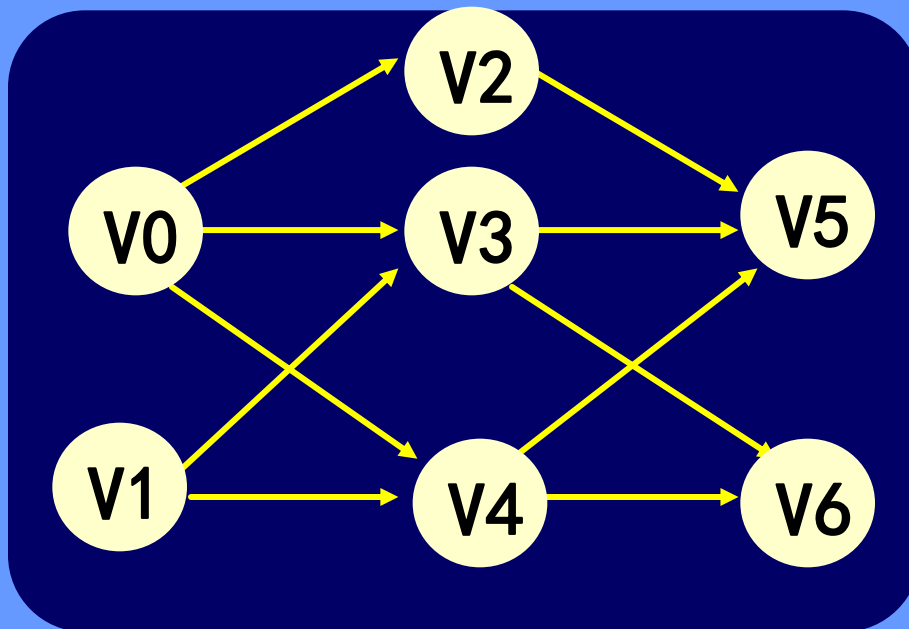
如何安排施工计划？
如何安排教学计划？



一个可行的施工计划为: $V_0, V_1, V_2, V_4, V_3, V_5, V_6,$

一个可行的学习计划为: $C_1, C_9, C_4, C_2, C_{10}, C_{11}, C_{12}, C_3, C_6, C_5, C_7, C_8$

可行的计划的特点: 若在流程图中顶点 v 是顶点 u 的前趋, 则在计划序列中顶点 v 也是 u 的前趋。

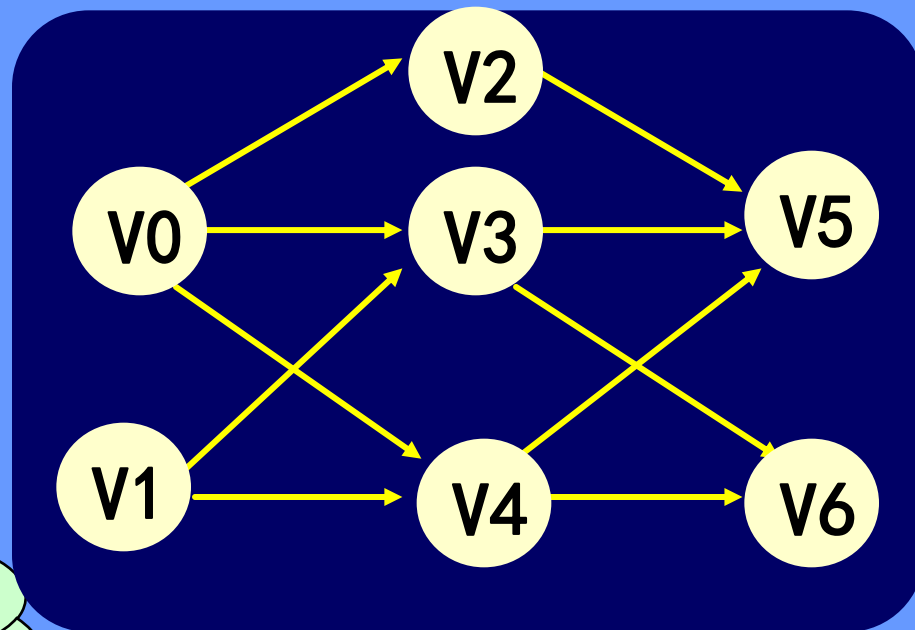


拓扑序列：有向图D的一个顶点序列称作一个拓扑序列，如果该序列中任两顶点 v 、 u ，若在D中 v 是 u 前趋，则在序列中 v 也是 u 前趋。

拓扑排序：就是将有关图中顶点排成拓扑序列。

拓扑排序的应用

- ☞ 安排施工计划（如上）
- ☞ 判断工程流程的是否合理

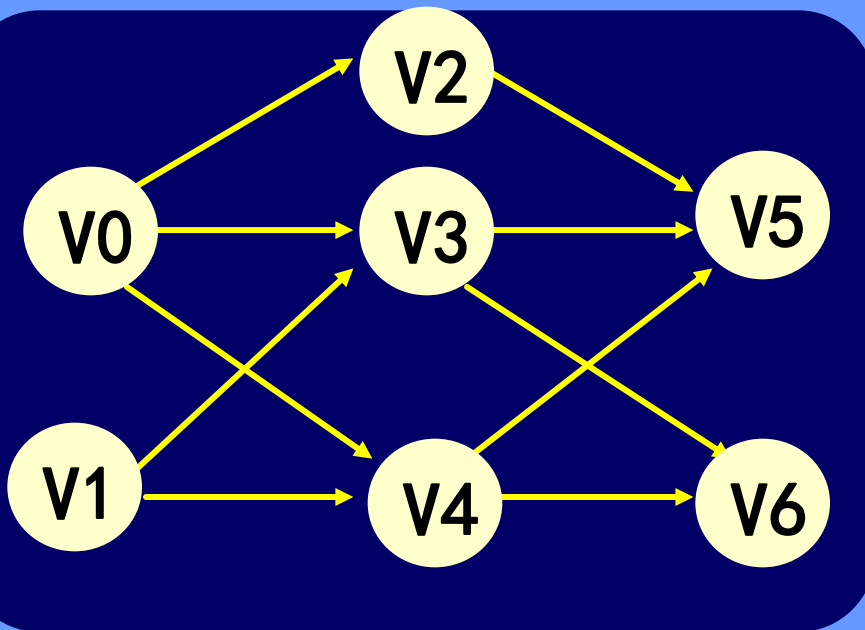


如何判断AOV网（有向图）
是否存在有向回路？

AOV网（有向图）
不存在有向回路
当且仅当能对AOV网
进行拓扑排序

2 拓扑排序方法:

- 1) 在有向图选一无前趋的顶点 v , 输出;
- 2) 从有向图中删除 v 及以 v 为尾的弧;
- 3) 重复1、2、直接全部输出全部顶点或有向图中不存在无前趋顶点; 例:
 $V_0, V_1, V_2, V_3, V_4, V_5, V_6$



如何在计算机上实现
对有向图的拓扑排序?



3 拓扑排序算法

1) 拓扑排序方法的另一种描述(等价描述)

- (1) 选择一入度为 0 顶点 v , 输出;
- (2) 将 v 邻接到的顶点的入度减1;
- (3) 重复1、2 直到输出全部顶点或有向图没有入度为0的顶点;

2) 拓扑排序涉及的数据和操作:

数据: 有向图, 顶点的入度;

操作: (1) 选择一入度为 0 顶点 v , 输出;
(2) 将 v 邻接到的顶点 u 的入度减1;

```
struct node  
{ int degree;  
  struct node *link;  
};typedef struct node NODE;
```

3) 有关数据的存储结构

① 有向图(AOV网): **G**② 顶点的入度: **int degree**③ 整数栈**int stack[]**: 用于存储入度为0的顶点的编号

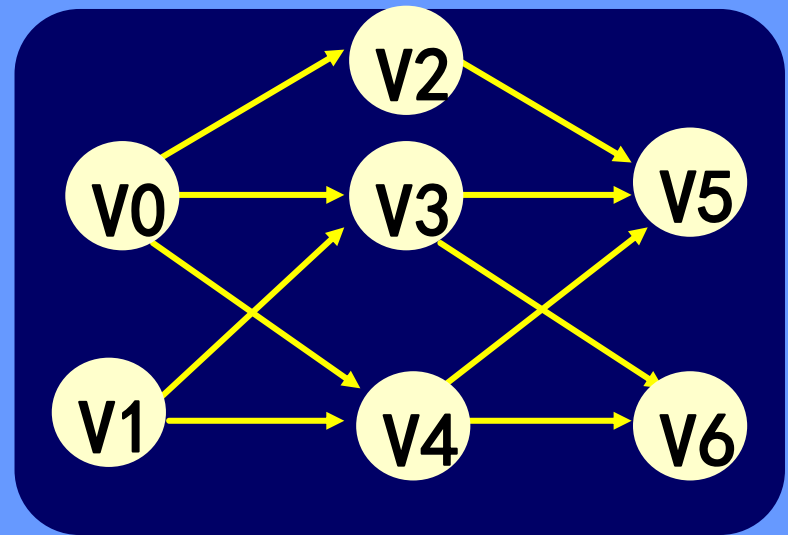
下标 入度 link

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 0 | 0 | → | 2 | → | 3 | → | 4 | ^ |
| 1 | 0 | → | 3 | → | 4 | ^ | | |
| 2 | 1 | → | 5 | ^ | | | | |
| 3 | 2 | → | 5 | → | 6 | ^ | | |
| 4 | 2 | → | 5 | → | 6 | ^ | | |
| 5 | 3 | ^ | | | | | | |
| 6 | 2 | ^ | | | | | | |

top

base

| |
|---|
| 6 |
| 5 |
| 4 |
| 3 |
| 2 |
| 1 |
| 0 |



初始时, 只有v0, v1
两顶点的入度为0

4) 建立AOV网的邻接表算法

```
int create( NODE adjlist[ ] )
{ NODE *p;
  int num, i, v1,v2;
  scanf( "%d\n", &num);

  for (i=0; i<num; i++) //初始化空关系图
  { adjlist[i].link=NULL; adjlist[i].degree=0; }

  for(; ;) { scanf("%d to %d\n", &v1, &v2); //读入一条弧

    if (v1<0 || v2<0) break; // 数据输入的终止条件

    p=(NODE *)malloc(sizeof(NODE));

    p->vertex=v2;

    p->link=adjlist[v1].link; adjlist[v1].link=p; //插入在链表首部

    adjlist[v2].degree++; //统计每个顶点的入度

  } return(num); // 返回图的结点数
```

}

5) 拓扑排序算法

```
int toposort(NODE adjlist[ ], int num)
//有向图G采用邻接表存储结构。
//若G无回路，则输出G的顶点的一个拓扑序列并返回1，否则0。
{ int stack[MAX]; //存放入度为0的顶点的栈
  int i, top, out, k; NODE *p;
  int num1=0;top=0;
  for(i=0; i<num; i++)
    if (adjlist[i].vertex==0) //入度为0顶点的编号进栈
      { stack[top]=i; top++;}
```

```

while( top>0)
{ top--; out=stack[top];
  printf( “%d,”,out); num1++;
  p=adjlist[out].link;
  while ( p!=NULL)
  { k=p->vertex;
    adjlist[k].vertex- -; //顶点入度减1,
    if (adjlist[k].vertex==0) //若入度减为0, 则入栈
      {stack[top]=k;top++;}
      p=p->link;
    }
  }
}

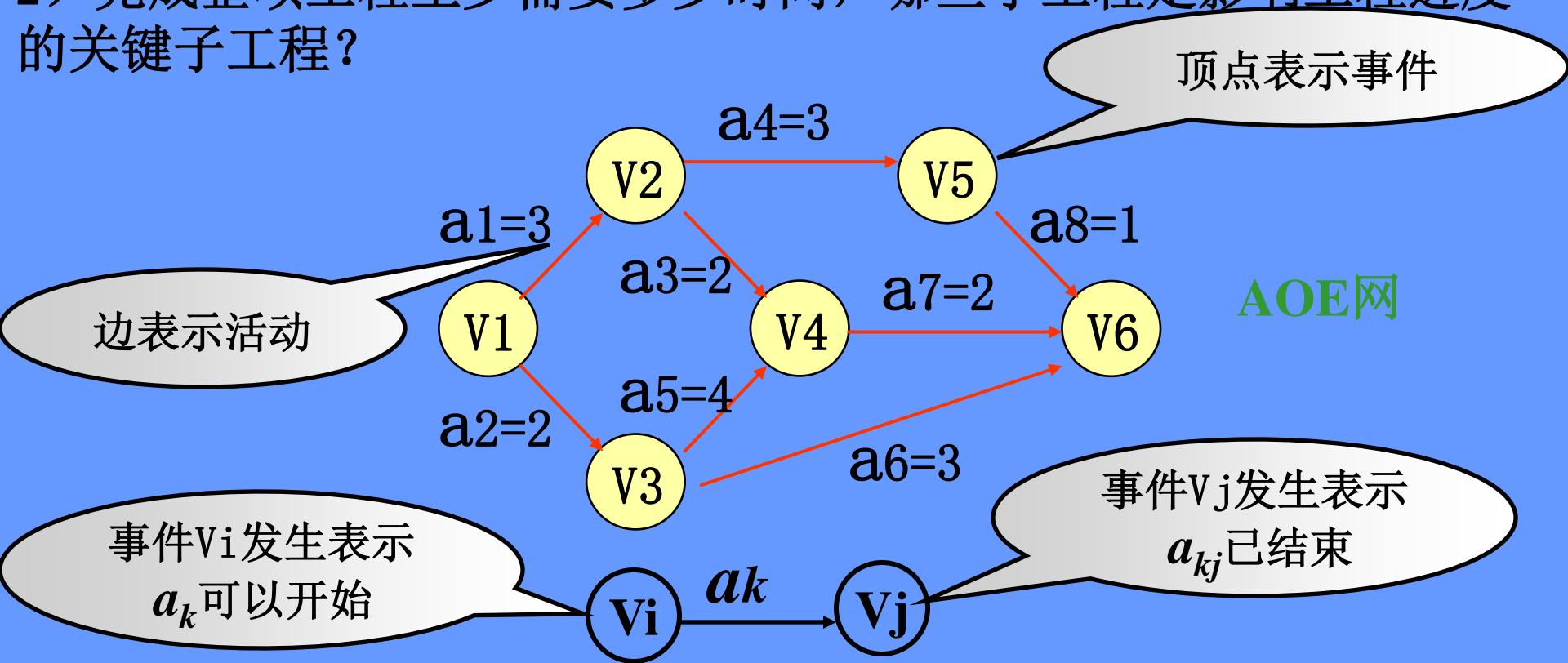
if (num1<num)  return( 0); //该有向图有回路
else return (1);
}

```

三 AOE网与关键路径

对工程人们关心两类问题：

- 1) 工程能否顺序进行，即工程流程是否“合理”
- 2) 完成整项工程至少需要多少时间，哪些子工程是影响工程进度的关键子工程？



为计算完成整个工程至少需要多少时间，需将每一子工程所需的时间作为权值赋给AOE网的各边。

AOV网，AOE网，用都能表示工程各子工程的流程，一个用顶点表示活动，一个用边表示活动，但AOV网侧重表示活动的前后次序，AOE网除表示活动先后次序，还表示了活动的持续时间等，因此可利用AOE网解决工程所需最短时间及哪些子工程拖延会影响整个工程按时完成等问题。

实际应用中采用那一种图，取决于要求解的问题。

§ 7.6 有向无环图的应用

V1 开始事件, V6 结束事件

事件 v_j 的最早发生时间 $ve[j]$:

事件 v_j 的最迟发生时间 $vl[i]$:

指在不推迟整个工期的前提下,

事件 v_j 的最晚发生时间

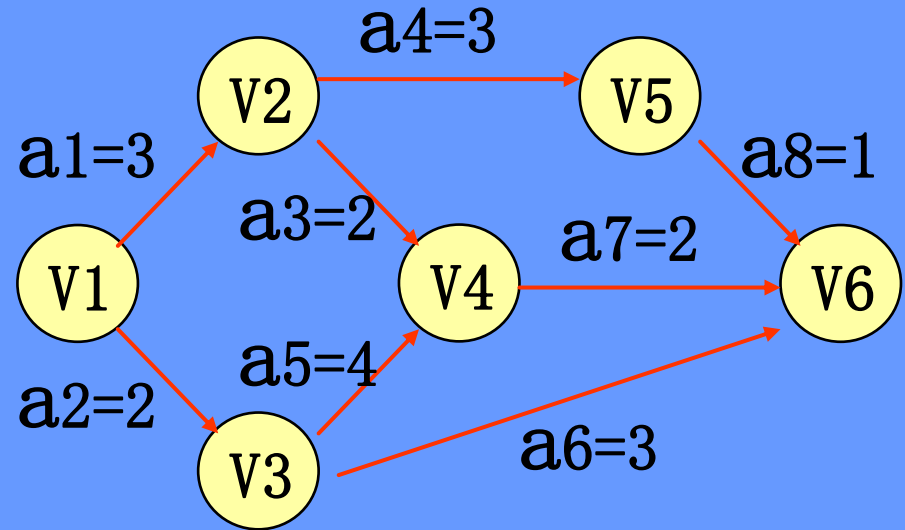
活动 a_k 的最早开始时间 $e[k]$

活动 a_k 的最晚开始时间 $l[i]$:

指在不推迟整个工期的前提下,

事件 a_{k_j} 的最晚发生时间

活动 a_k 的延迟时间 $diff[k]$



| 事件 V_j | 最早发生时间 | 最晚发生时间 |
|----------|--------|--------|
| V1 | 0 | 0 |
| V2 | 3 | 4 |
| V3 | 2 | 2 |
| V4 | 6 | 6 |
| V5 | 6 | 7 |
| V6 | 8 | 8 |

| 活动 a_k | 最早发生时间 | 最晚发生时间 |
|----------|--------|--------|
| a1 | 0 | 1 |
| a2 | 0 | 0 |
| a3 | 3 | 4 |
| a4 | 3 | 4 |
| a5 | 2 | 2 |
| a6 | 2 | 5 |
| a7 | 6 | 6 |
| a8 | 6 | 7 |

1 事件 v_j 的最早发生时间 $ve[j]$

$$ve[1]=0$$

$$ve[j]=\text{Max}\{ve(i)+dut(<i,j>)\} \quad ve[j]=\text{从源点到顶点 } v_j \text{ 的最长路径的长度}$$

2 事件 v_j 的最迟发生时间 $vl[i]$

$ve[j]$ =从顶点 v_j 终点到的最长路径的长度。

$$vl[n]=ve[n]$$

$$vl[j]=\text{Min}\{vl(k)-dut(<j,k>)\} \quad vl[i]=vl[n]-\text{从顶点 } v_j \text{ 终点到的最长路径的长度}$$

3 活动 a_k 的最早开始时间 $e[k]$

设 $a_k=<i,j>$, $e[k]=ve[i]$

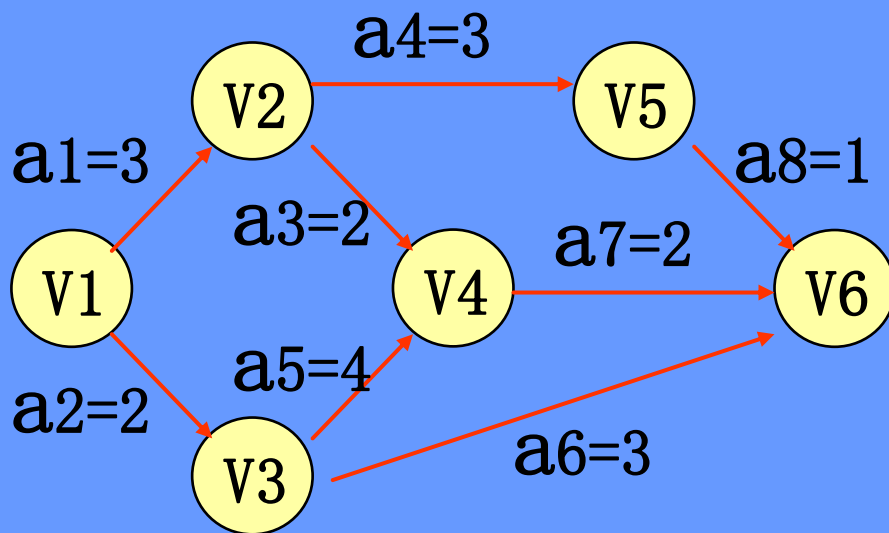
4 活动 a_k 的最晚开始时间 $l[i]$

$$l[k]=vl[j]-dur(<i,j>)$$

5 活动 a_k 的延迟时间 $diff[k]$

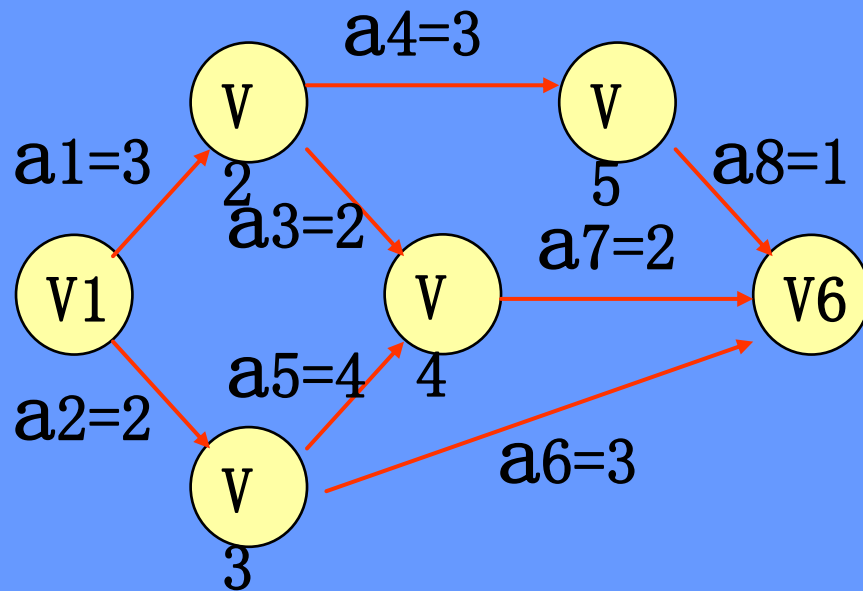
$$diff[k]=l[k]-e[k]$$

把活动 a_i 的最晚开始时间 $l[i]$ 与最早开始时间 $e[i]$ 之差定义为活动 a_i 的延迟时间



6 关键活动对于活动 a_i ,

若有 $l[i]-e[i]=0$ 为关键活动，由关键活动组成的路径就是关键路径。



| 事件 V_j | 最早发生时间 | 最晚发生时间 |
|----------|--------|--------|
| V1 | 0 | 0 |
| V2 | 3 | 3 |
| V3 | 2 | 2 |
| V4 | 4 | 4 |
| V5 | 6 | 6 |
| V6 | 9 | 9 |

| 活动 a_k | 最早发生时间 | 最晚发生时间 |
|----------|--------|--------|
| a_1 | 0 | 1 |
| a_2 | 0 | 0 |
| a_3 | 3 | 4 |
| a_4 | 3 | 4 |
| a_5 | 2 | 2 |
| a_6 | 2 | 5 |
| a_7 | 6 | 6 |
| a_8 | 6 | 7 |

第七章 结束