数据结构与算法

第九章-I 图的基本概念

裴文杰

计算机科学与技术学院 教授

什么是三观不同?
你敬畏天理，他崇拜权威，这是世界观不同;
你站在良知的一边，他站在赢者的一遍，这是价值观不同;
你努力是为了理想的生活，他努力是为了做人上人，这是人生观不同

——毛姆

在繁华中自律，
在落魄中自愈，
谋生的路上不抛弃良知，
谋爱的路上不放弃尊严。

第九章 图

9.1 图的定义和术语

9.2 图的存储结构

9.3 图的遍历

9.4 有向无环图的应用

9.5 最短路径

本章知识点

- 图的类型定义
- 图的存储表示
- 图的深度优先搜索遍历
- 图的广度优先搜索遍历
- 无向网的最小生成树
- 最短路径
- 拓扑排序
- 关键路径

本章难点

- 最小生成树的算法
- 拓扑排序的算法;
- 关键路径算法;
- 求最短路径的Dijkstra算法和Floyed算

第九章 图

9.1 图的定义和术语（集合与图论）

9.2 图的存储结构

9.3 图的遍历

9.4 有向无环图的应用

9.5 最短路径

9.1 图的定义和术语

- 图的抽象数据类型定义

ADT Graph {

数据对象V： 具有相同特征的数据元素的集合，称为**顶点**集。

数据关系R： R={VR} VR={<v,w> | v,w∈V且 <v,w> 表示从v到w的**弧**}

基本操作

} ADT Graph

9.1 图的定义和术语

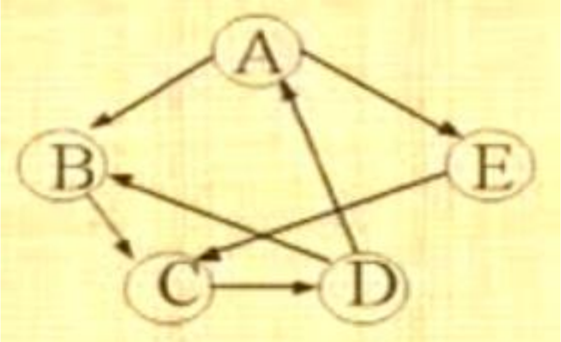
- 弧头、弧尾、弧、有向图

若<v,w>∈ {VR}，

则<v,w>表示从顶点v到顶点w的一条弧。

称顶点v为弧尾，称顶点w为弧头。 注意：起点为弧尾

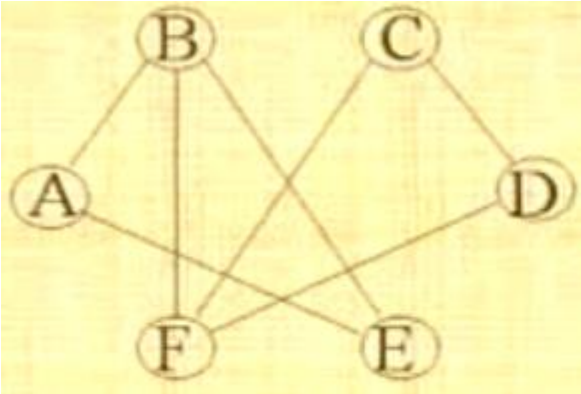
由顶点集和弧集构成的图称作有向图。



9.1 图的定义和术语

- 边、无向图

若<v,w>∈ {VR}，必有<w,v >∈ {VR}，那么VR是对称的，则称顶点v和顶点w之间存在一条边, 用无序对(v, w)表示。由顶点集和边集构成的图称作无向图。



9.1 图的定义和术语

- 网、子图

弧或边 带权的图分别称作有向网（network）或无向网。

设图G=(V,{VR})和图G’=(V’,{VR’}),

且V’$\subseteq$V, VR’$\subseteq$VR,

则称G’为G的子图，若V’==V，则为生成子图

9.1 图的定义和术语

- 完全图、稀疏图、稠密图

假设图中有n个顶点，e条边，则

含有$e=n(n-1)/2$条边的无向图称作完全图；

含有$e=n(n-1)$条弧的有向图称作有向完全图；

若边或弧的个数$e<n\log_2 n$，则称作稀疏图，
否则称作稠密图。

9.1 图的定义和术语

- 邻接点、度、入度、出度

假若顶点v和顶点w之间存在一条边，

则称顶点v和w互为邻接点，边(v,w)和顶点v和w相关联。

和顶点v关联的边的数目定义为边的度。

对有向图来说，

以顶点v为弧尾的弧的数目定义为顶点的出度；

以顶点v为弧头的弧的数目定义为顶点的入度。

度(TD) = 出度(OD) + 入度(ID)

9.1 图的定义和术语

- 路径、路径长度、简单路径、简单回路

设图G=(V,{VR})中的一个顶点序列

$\{u=v_{i,0},v_{i,1},\ldots,v_{i,m}=w\}$中，$(v_{i,j-1},v_{i,j}) \in VR$, $1<=j<=m$,

则称从顶点u到顶点w之间存在一条路径，

路径上边的数目称作路径长度。

若序列中的边不重复出现，则称作简单路径；若路径中的所有顶点和所有边都不重复出现，则称作初级路径。

若u=w，则称这条路径为回路

9.1 图的定义和术语

- 连通图、连通分量、强连通图、强连通分量

若图G中任意两个顶点之间都有路径相通，则称作此图为**连通图**；

若无向图为非连通图，则图中各个极大连通子图称作此图的**连通分量**。

对有向图，若任意两个顶点之间都存在一条有向路径，则称此有向图为**强连通图**。否则，其各个强连通子图称作它的**强连通分量**。

9.1 图的定义和术语

连通图（强连通图）：在无(有)向图G=< V, E >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）

子图：设有两个图G=（V， E）、G1=（V1， E1），若V1⊆ V，E1 ⊆ E，E1关联的顶点都在V1中，则称 G1是G的子图；

生成子图：如果V1==V

9.1 图的定义和术语

- 生成树、生成森林

假设一个连通图有n个顶点和e条边，其中n-1条边和n个顶点构成一个极小连通子图，称该极小连通子图为此连通图的生成树。

对非连通图，则称由各个连通分量的生成树的集合为此非连通图的生成森林。

9.4.1 生成树(spanning tree)

□**连通图的生成树**:假设一个连通图有 n 个顶点和 e 条边，其中 **n-1 条边和 n 个顶点**构成一个极小连通子图，称该极小连通子图为此连通图的生成树;

□**非连通图的生成森林**:对非连通图，则称由各个连通分量的生成树的集合为此非连通图的生成森林。

生成树另外一种定义：如果一个子图既是生成子图又是树，那么该子图为生成树

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

9.4.1 生成树(spanning tree)

- **生成树**：连通图G的一个子图如果是一棵**包含G的所有顶点的树**，则该子图称为G的生成树。
- **生成树是连通图的极小连通子图**。包含图G 的**所有顶点**，但只有**n-1 条边**，所谓**极小**是指：若在树中任意增加一条边，则将出现一个**回路**；若去掉一条边，将会使之变成非连通图。
- 利用深（广）度优先搜索可以实现求深（广）度优先生成树

9.1 图的定义和术语

- **ADT Graph的基本操作**

CreatGraph(&G) //建立图

DestroyGraph(&G) //销毁图

InsertVEx(&G, v) //插入顶点

DeleteVEx(&G, v) //删除顶点

InsertArc(&G, v, w) //插入弧

DeleteArc(&G, v, w) //删除弧

DFSTraverse(G, v, Visit()) //深度优先搜索

BFSTraverse(G, v, Visit()) //广度优先搜索

第九章 图

9.1 图的定义和术语（集合与图论）

9.2 图的存储结构

9.3 图的遍历

9.4 有向无环图的应用

9.5 最短路径

9.2 图的存储结构

9.2.1 邻接矩阵---顺序存储

9.2.2 邻接表------链式存储

9.2.3 有向图的十字链表存储表示

9.2.4 无向图的邻接多重表存储表示

9.2 图的存储结构

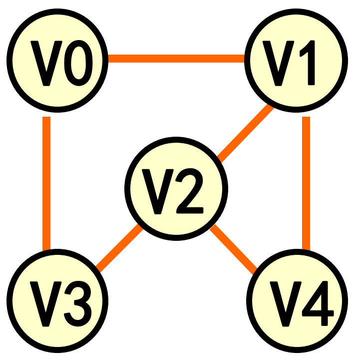
9.2.1 邻接矩阵---顺序存储

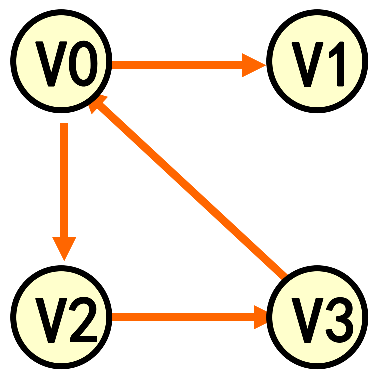
9.2.2 邻接表------链式存储

9.2.3 有向图的十字链表存储表示

9.2.4 无向图的邻接多重表存储表示

9.2.1 邻接矩阵（Adjacency Matrix）





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？

约定：G=<V, E>是图, V={v_0, v_1, v_2, … v_{n-1} }，设顶点的角标为它的编号

9.2.1 邻接矩阵（Adjacency Matrix）

用邻接矩阵通过顺序存储表示数据元素之间的关系（边或者弧）

```
typedef struct ArcCell { // 弧的定义
    VRType  adj;    // VRType是顶点关系类型。
            // 对无权图，用1或0表示相邻否；
            // 对带权图，则为权值类型。
    InfoType  *info;  // 该弧相关信息的指针
} ArcCell,
AdjMatrix[MAX_VERTEX_NUM][MAX_VERTEX_NUM];
//邻接矩阵表示弧
```

9.2.1 邻接矩阵（Adjacency Matrix）

```
typedef struct { // 图的定义
    VertexType vexs[MAX_VERTEX_NUM]; // 顶点信息
    AdjMatrix   arcs;      // 弧的信息
    int    vexnum, arcnum;   // 顶点数，弧数
    GraphKind  kind;     // 图的种类标志
} MGraph;
```

9.2.1 邻接矩阵（Adjacency Matrix）

一种简单的实现方式

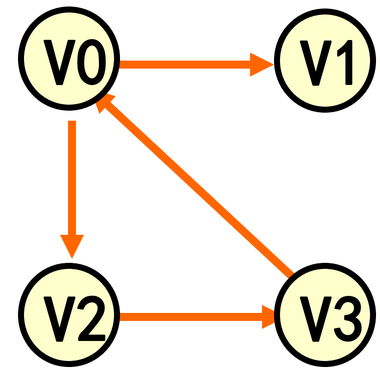
邻接矩阵表示法中图的描述

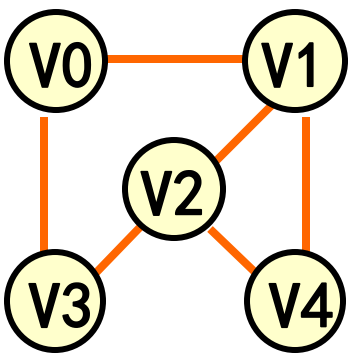
```
#define  n  6       /*图的顶点数*/
#define  e  8       /*图的边数*/
typedef char vextype;  /*顶点的数据类型*/
typedef float adjtype; /*权值类型*/
typedef struct
{ vextype vexs[n];
  adjtype arcs[n][n];
} graph;
```

9.2.1 邻接矩阵（Adjacency Matrix）

邻接矩阵： G的邻接矩阵是满足如下条件的n阶矩阵：

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





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

无向图的邻接矩阵是对称的，可以采用压缩存储（只存上三角或者下三角元素）

有向图的邻接矩阵可能是不对称的。

9.2.1 邻接矩阵（Adjacency Matrix）

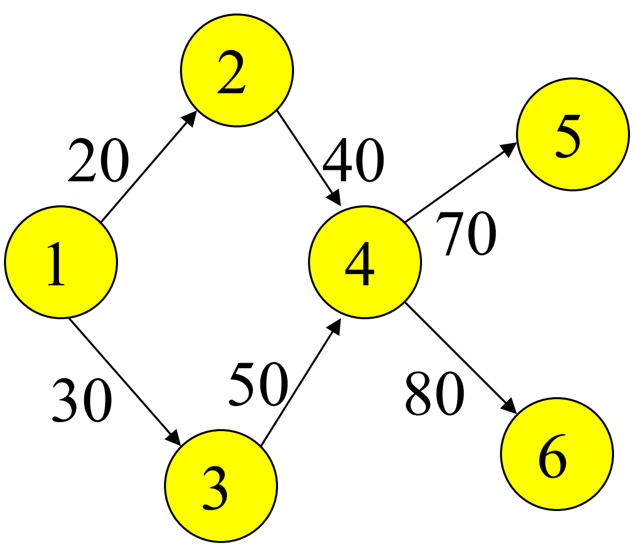
邻接矩阵表示法特点：

- 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;
- 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;
- 在有向图中, 统计第 i 行 1 的个数可得顶点 i 的出度，统计第 j 列 1 的个数可得顶点 j 的入度。
- 在无向图中, 统计第 i 行 (列) 1 的个数可得顶点i 的度

9.2.1 邻接矩阵（Adjacency Matrix）

- 网络（带权图）的邻接矩阵

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



$$arcs = \left\{ \begin{matrix} \infty & 20 & 30 & \infty & \infty & \infty \\ \infty & \infty & \infty & 40 & \infty & \infty \\ \infty & \infty & \infty & 50 & \infty & \infty \\ \infty & \infty & \infty & \infty & 70 & 80 \\ \infty & \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty \end{matrix} \right\}$$

9.2.1 邻接矩阵（Adjacency Matrix）

- 树的存储：
  - 双亲表示法；
  - 孩子链表表示法；
  - 双亲孩子表示法；
  - 孩子兄弟表示法—树的二叉链表

树的存储给图的存储的启示？
用链式存储结构表示图？

9.2 图的存储结构

9.2.1 邻接矩阵---顺序存储

9.2.2 邻接表------链式存储

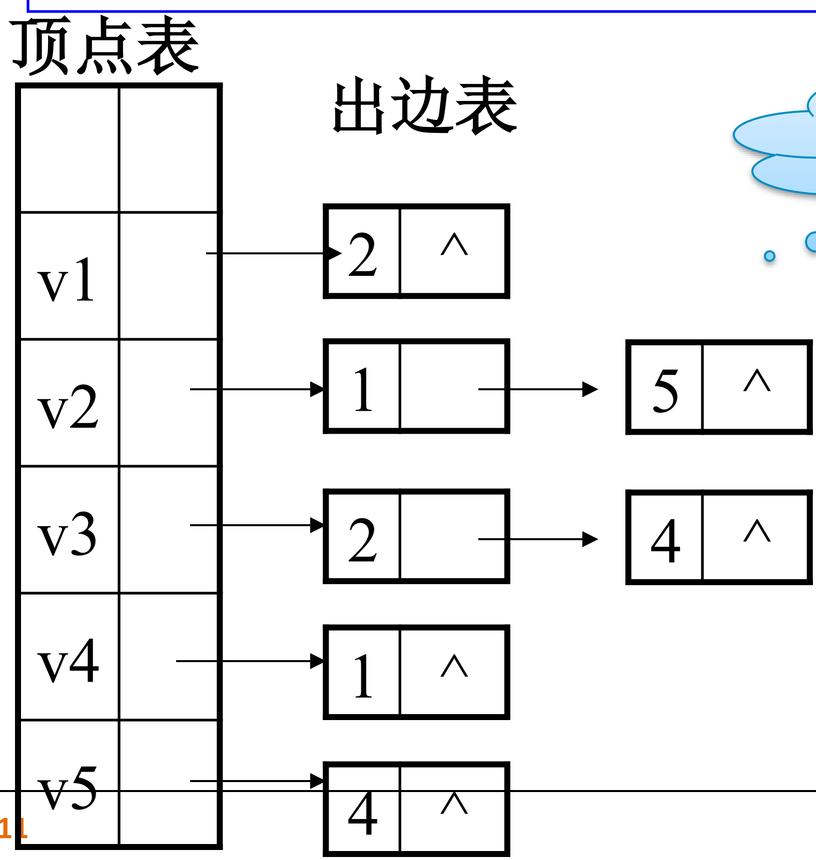
9.2.3 有向图的十字链表存储表示 （了解）

9.2.4 无向图的邻接多重表存储表示（了解）

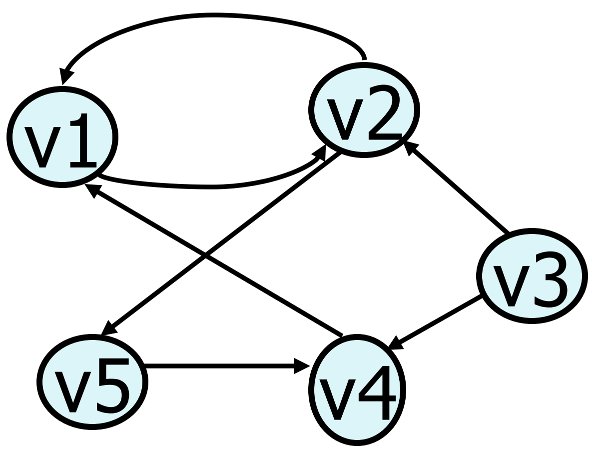
9.2.2 邻接表（Adjacency List）

- 有向图的邻接表

> ➢ 有向图的邻接表（出度）
> - 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

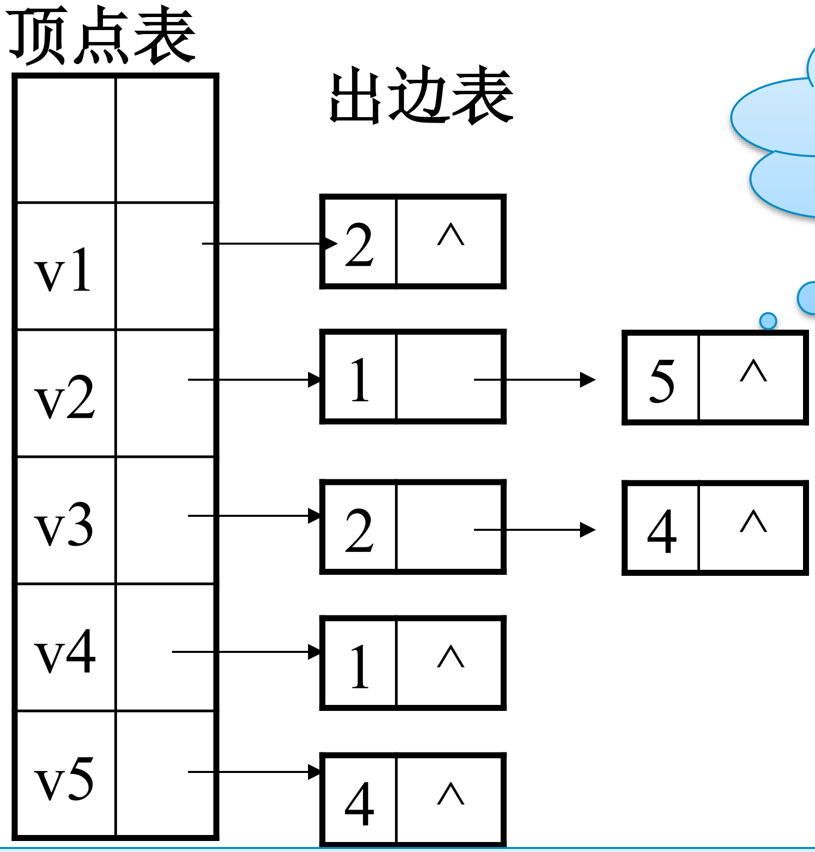


邻接表的存储类似于树的孩子链表

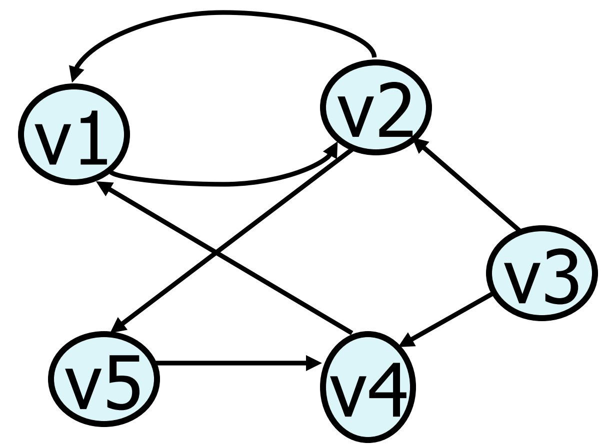


9.2.2 邻接表（Adjacency List）

讨论：求有向图邻接表中第i个顶点的出度？图的边数?



邻接表的存储类似于树的孩子链表

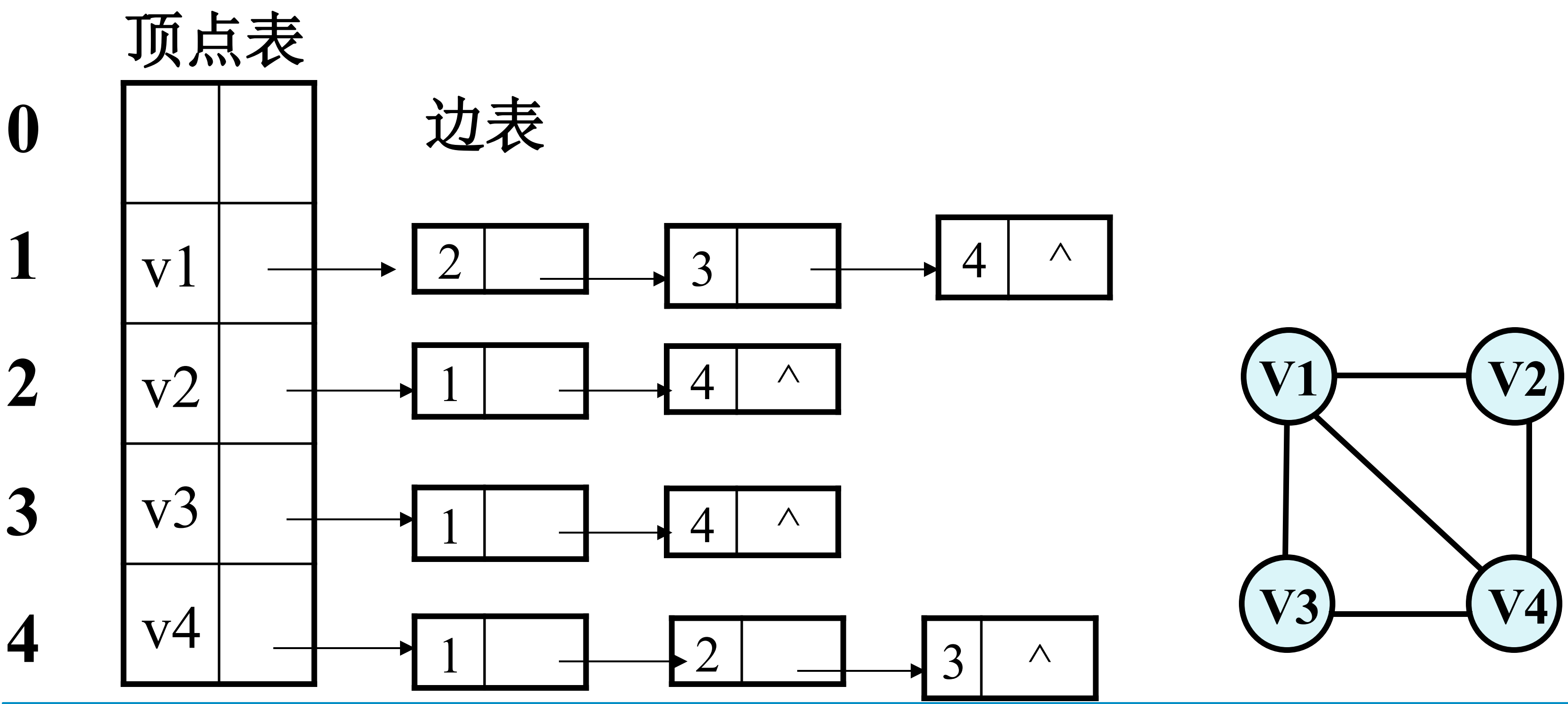


出度：第i个顶点的边链表结点个数之和。

图的边数：所有边链表结点个数之和。

9.2.2 邻接表（Adjacency List）

无向图的邻接表



顶点的度：第i个顶点的边链表结点个数之和；

图的总度数：所有边链表结点个数之和；

图的边数：所有边链表结点个数之和的一半。

9.2.2 邻接表（Adjacency List）

- 图的邻接表

邻接表的定义分两部分：
- 顶点表
- 边链表

顶点表

data	firstarc

一个顶点定义一个顶点表，然后可以用顺序存储的形式存储所有顶点，以便随机访问任一顶点的链表。

顶点表的存储类型：

```
typedef struct Vnode {
    vertextype  data;                    // 顶点信息
    Arcnode  *firstarc;     // 指向第一条依附该顶点的弧
 } VNode, AdjList[MAX_VERTEX_NUM];
```

9.2.2 邻接表（Adjacency List）

- 图的邻接表

边链表

adjvex	nextarc	info

```
#define MAX_VERTEX_NUM 20
边表的存储类型：
typedef struct ArcNode {
  int   adjvex;                    // 该弧所指向的顶点的位置
  struct ArcNode  *nextarc;  // 指向下一条弧的指针
  infoType   *info;              // 可以存储图中边的权值
} ArcNode;
```

9.2.2 邻接表（Adjacency List）

- 图的邻接表

图的存储：

```
typedef struct{
   AdjList   vertices;
   int vexnum, arcnum;  // 图的当前顶点数和弧数
   int kind;                    // 图的种类
} ALGraph;
```

对于无向图，如果有n个顶点，e条边，则邻接表需要n个顶点表（头结点）和2e个表节点。显然在边稀疏的情况下，用邻接表表示图比用邻接矩阵更节省存储空间。

9.2.2 邻接表（Adjacency List）

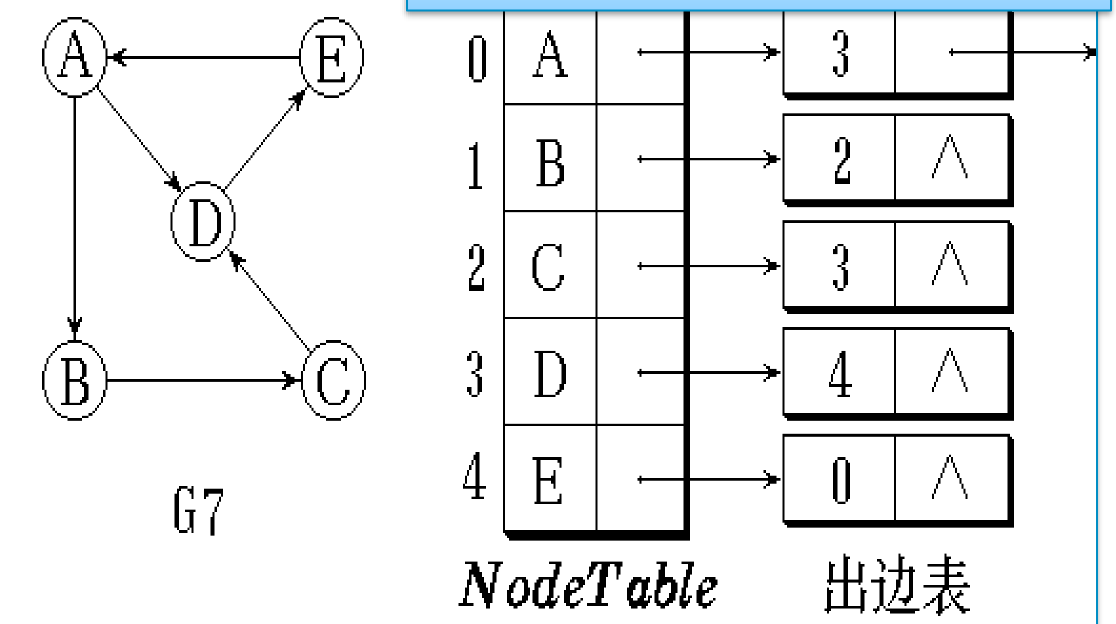
- 有向图的邻接表

一种简单的实现

邻接表的形式说明和建立算法

```
typedef struct        /*顶点表结点定义*/
{ vextype vertex;
  edgenode *link;
} vexnode;
vexnode ga[n];
```

vertex link adjvex next



（a） 邻接表

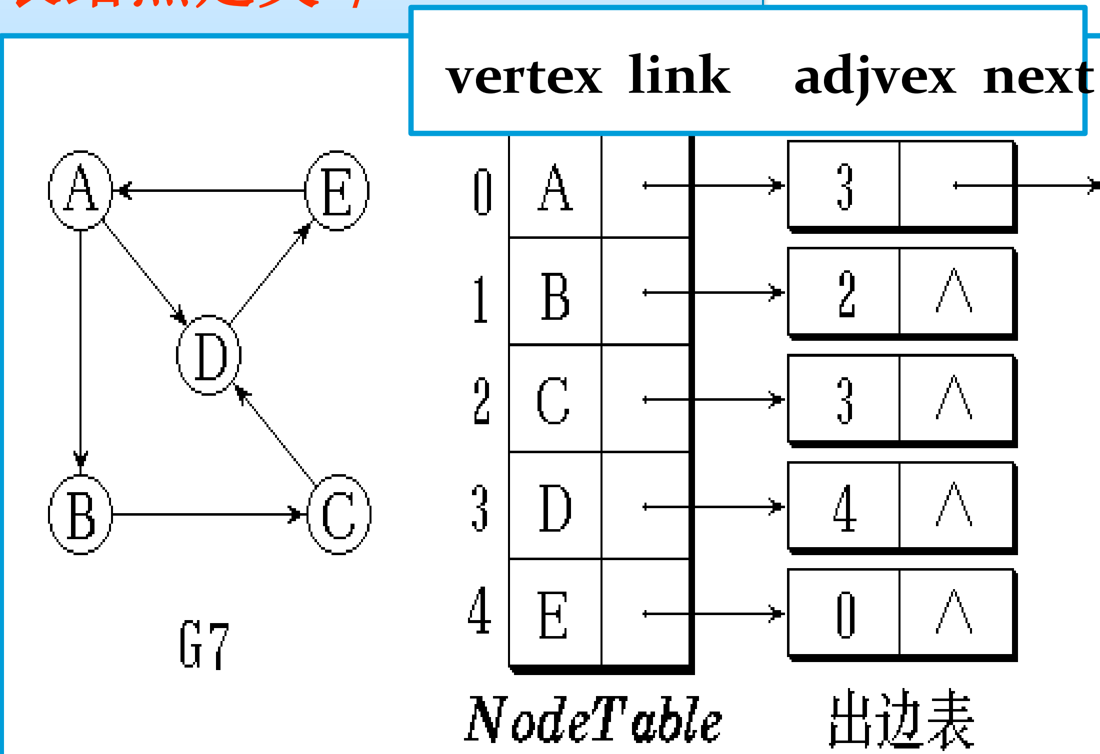
9.2.2 邻接表（Adjacency List）

- 有向图的邻接表

一种简单的实现

邻接表的形式说明和建立算法

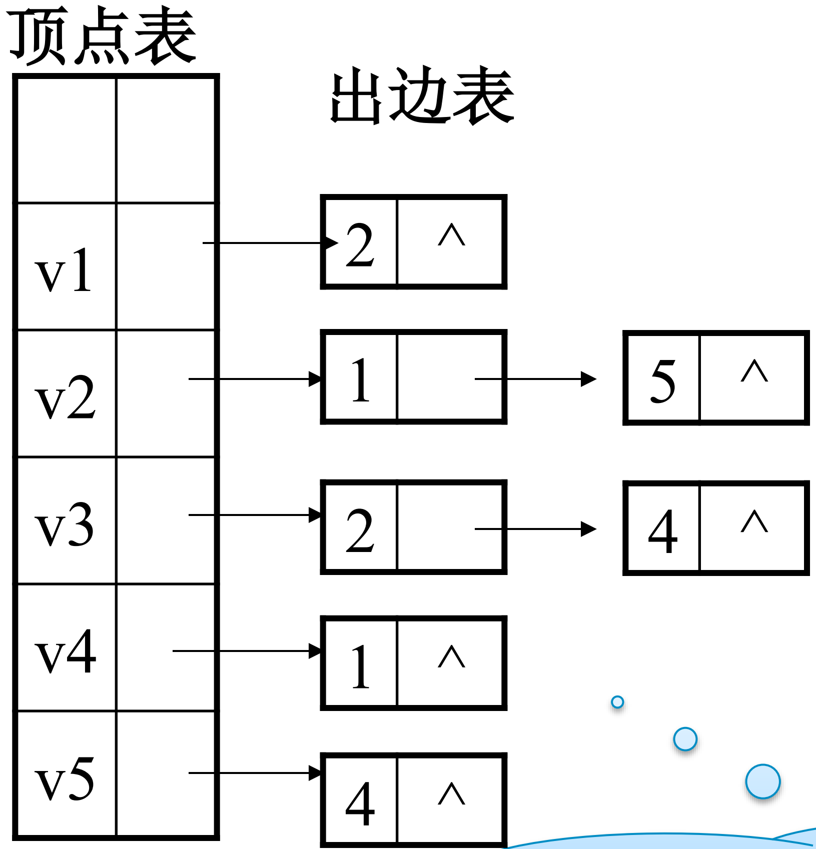
```
typedef struct node   /*边表结点定义*/
{ int adjvex;
  struct node *next;
} edgenode;
```



（a） 邻接表

9.2.2 邻接表（Adjacency List）

- 邻接表的建立



建立邻接表的主要操作是在链表中插入一个结点

9.2.2 邻接表（Adjacency List）

- 邻接表的建立---无向图

```
CREATADJLIST(vexnode ga[]){
    int i,j,k;
    edgenode *s;
    for (i=0;i<n;i++){   /*读入顶点信息并初始化*/
        ga[i].vertex＝getchar();
        ga[i].link＝NULL;
    }
```

9.2.2 邻接表（Adjacency List）

- 邻接表的建立---无向图

```
for (k=0;k<e;k++){   /*建立边表*/
    scanf("%d%d",&i,&j);
    s＝malloc(sizeof(edgenode));
    s->adjvex＝j;
    s->next＝ga[i].link
    ga[i].link＝s;
    s＝malloc(sizeof(edgenode));
    s->adjvex＝i;
    s->next＝ga[j].link;
    ga[j].link＝s;
  }/*对于无向图要插入两次，i后和j后*/
}
```

对于无向图，插入一条边(i, j)，需要插入两次：分别对顶点i和顶点j进行边的插入。

将S插入顶点的边表头部

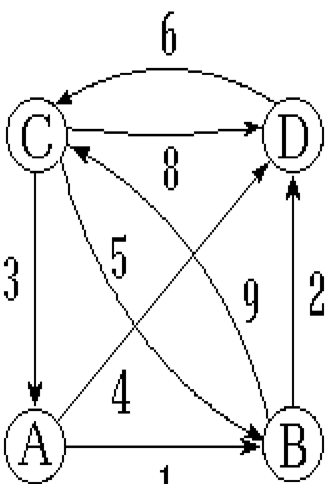
9.2.2 邻接表（Adjacency List）

- 邻接表的特点

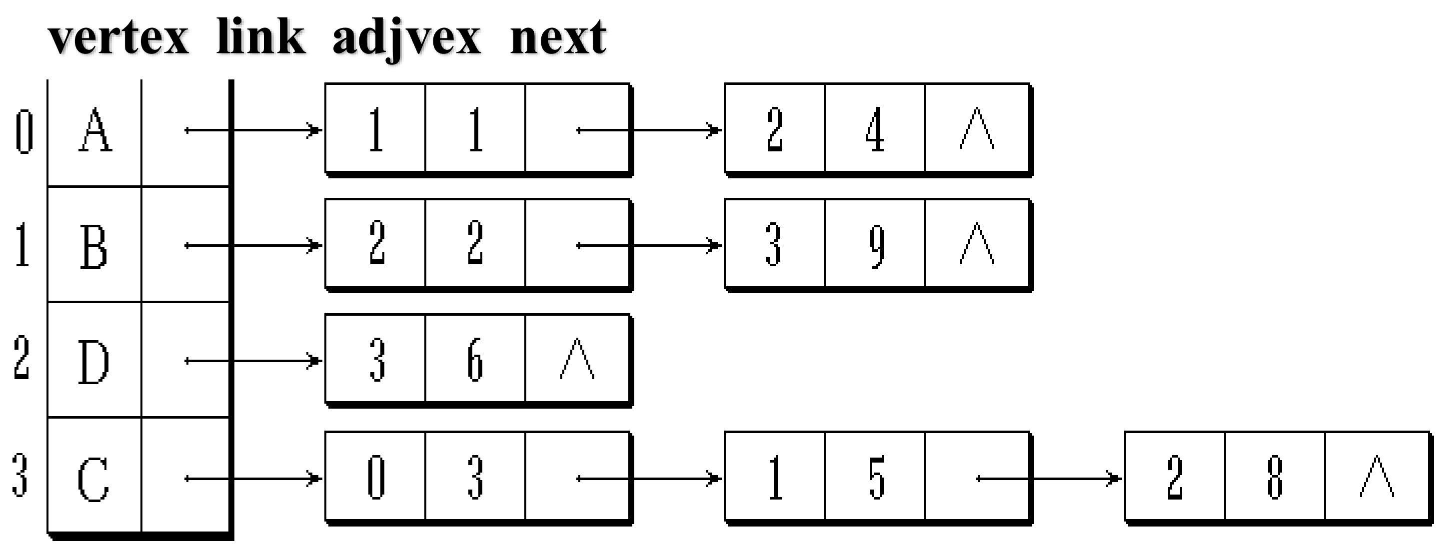
(1)对于顶点多边少的图采用邻接表存储节省空间；空间复杂度O(n+e)。

(2)容易找到任一顶点的第一个邻接点。

- 网络 (带权图) 的邻接表



G9

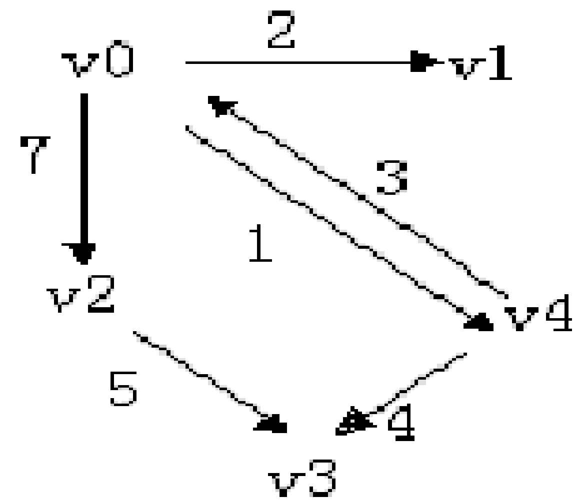


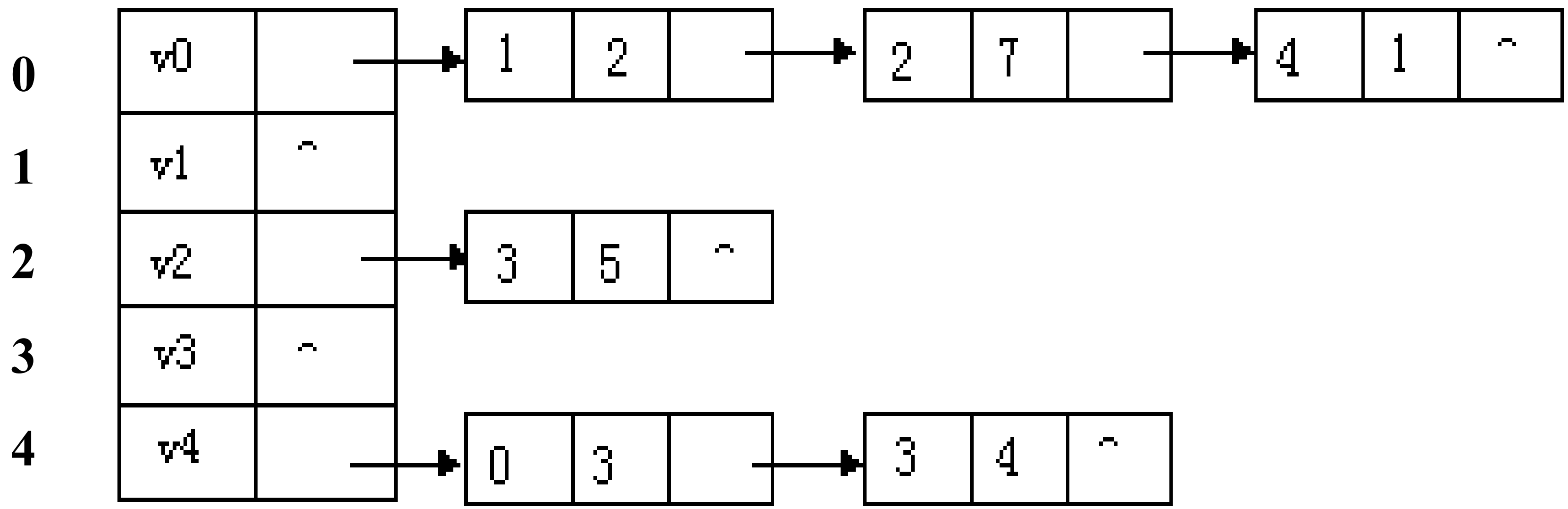
NodeTable Edge Linked List （出边表）

9.2.2 邻接表（Adjacency List）

- 网络 (带权图) 的邻接表

例 给出网络的邻接表

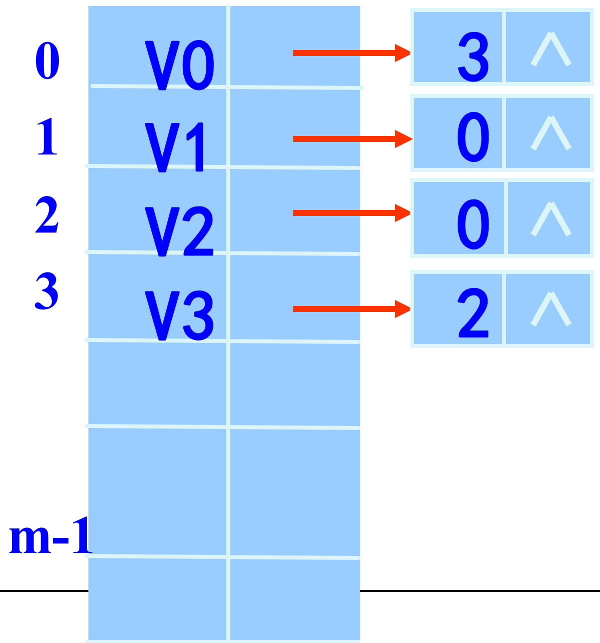


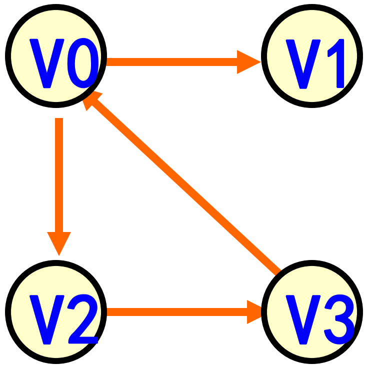


9.2.2 邻接表（Adjacency List）

- 逆邻接表

> ➢ 有向图的逆邻接表（入度）
> - 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

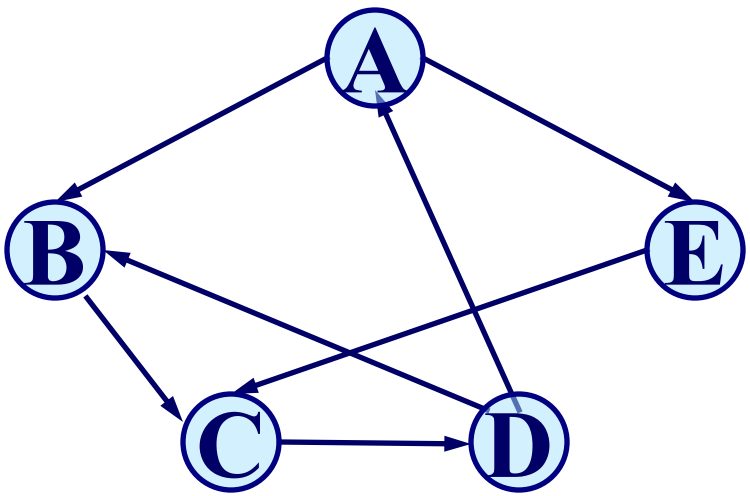


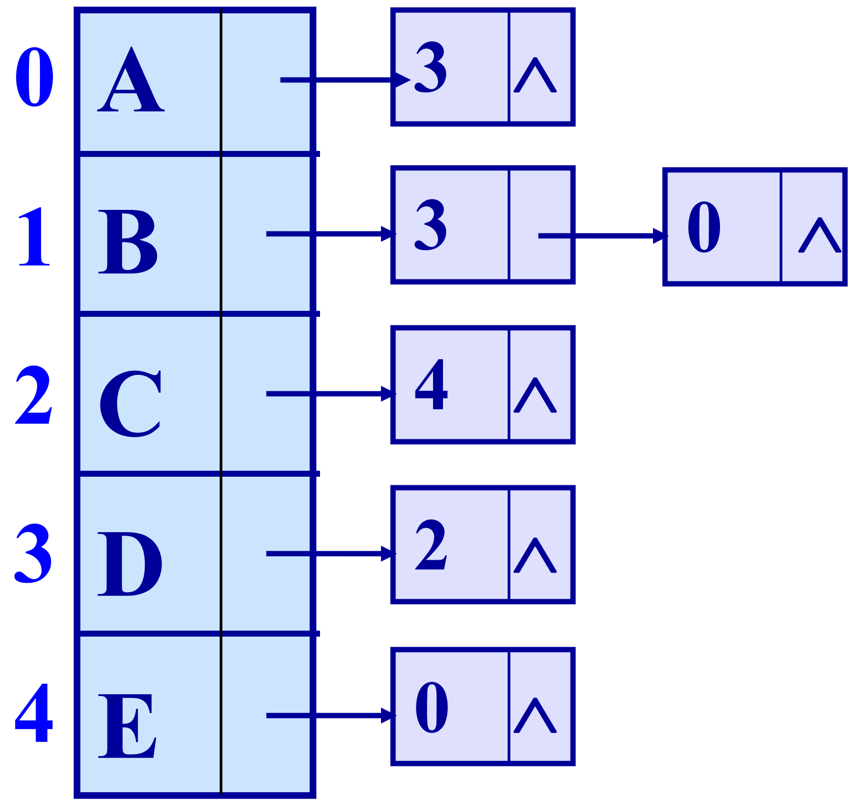


9.2.2 邻接表（Adjacency List）

- 逆邻接表

例 给出网络的逆邻接表





9.2.2 邻接表（Adjacency List）

- 邻接表的性质
  - ✓ 图的邻接表表示不唯一的，它与边结点的次序有关；
  - ✓ 无向图的邻接表中第i个顶点的度为第i个链表中结点的个数；
  - ✓ 有向图的邻接表中第i个链表的结点的个数是第i个顶点的出度；而第i个顶点的入度需遍历整个链表，采用逆邻接表,建立一个以v_i顶点为头的弧的表。
  - ✓ 无向图的边数等于邻接表中边结点数的一半，有向图的弧数等于邻接表中边结点数。

9.2 图的存储结构

9.2.1 邻接矩阵---顺序存储（集合与图论）

9.2.2 邻接表------链式存储

9.2.3 有向图的十字链表存储表示 （了解）

9.2.4 无向图的邻接多重表存储表示（了解）

9.2.3 十字链表 (Orthogonal List)

- 十字链表----有向图的链式存储

特点：将有向图的邻接表和逆邻接表结合得到的一种链表

弧的结点结构—链表

十字链表既容易找到以vi为尾的弧，也容易找到以vi为头的弧，因此容易求得顶点的出度和入度。

弧尾顶点位置	弧头顶点位置			弧的相关信息

指向下一个有相同弧头的弧结点

指向下一个有相同弧尾的弧结点

9.2.3 十字链表 (Orthogonal List)

- 十字链表----有向图的链式存储

特点：将有向图的邻接表和逆邻接表结合得到的一种链表

tailvex	headvex	hlink	tlink	info

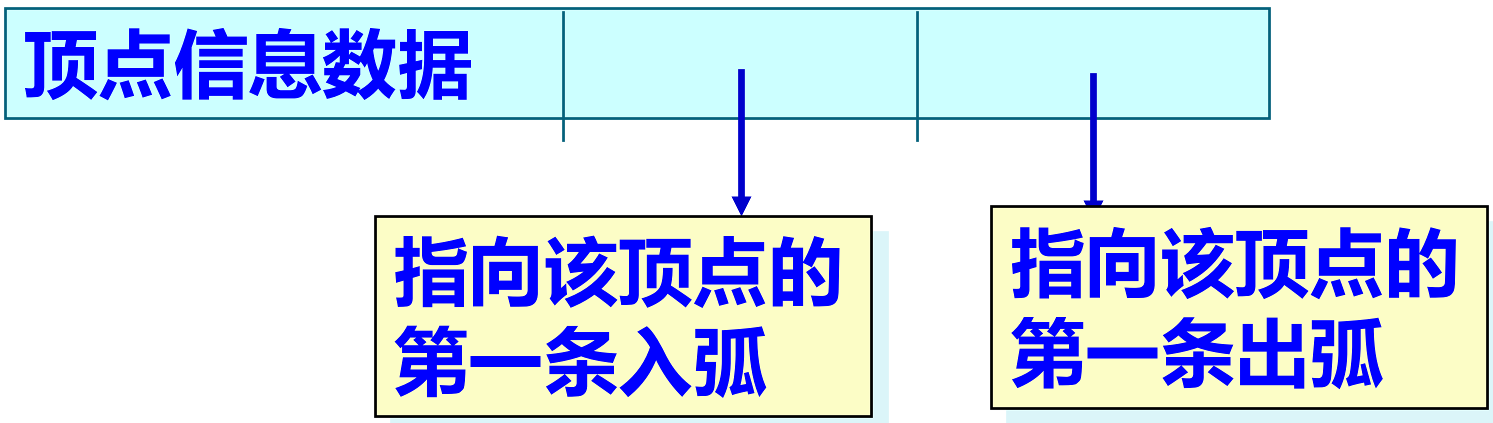
```
typedef struct ArcBox { // 弧的结构表示
    int tailvex, headvex;
    struct ArcBox  *hlink, *tlink;
    InfoType  *info; //该弧相关信息
} ArcBox;
```

9.2.3 十字链表 (Orthogonal List)

- 十字链表----有向图的链式存储

顶点的结点结构

data	firstin	firstout

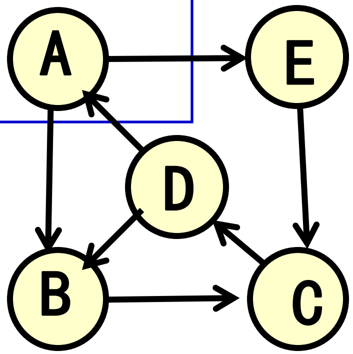



```
typedef struct VexNode { // 顶点的结构表示
    VertexType  data;
    ArcBox  *firstin, *firstout;
} VexNode;
```

9.2.3 十字链表 (Orthogonal List)

- 十字链表----有向图的链式存储

```
typedef struct {
   VexNode  xlist[MAX_VERTEX_NUM];
          // 顶点结点(表头向量)
   int   vexnum, arcnum;
         //有向图的当前顶点数和弧数
} OLGraph;
```



9.2 图的存储结构

9.2.1 邻接矩阵---顺序存储（集合与图论）

9.2.2 邻接表------链式存储

9.2.3 有向图的十字链表存储表示 （了解）

9.2.4 无向图的邻接多重表存储表示（了解）

9.2.4 邻接多重表 (Adjacency MultiList)

- 邻接多重表----无向图的链式存储
  - **边的结构表示**

mark	ivex	ilink	jvex	jlink	info

```
typedef struct EBox {
    VisitIf   mark;      // 访问标记
    int  ivex, jvex;     //该边依附的两个顶点的位置
    struct EBox  *ilink, *jlink;
                   //分别指向依附这两个顶点的下一条边
    infoType    *info;       // 该边信息指针
} EBox;
```

用普通邻接表表示无向图，一条边需要表示为两个结点，分别在顶点i和顶点j中的边链表中构建一次结点。
而邻接多重表通过ilink和jlink可以分别参与顶点i和顶点j的边链表的构建，从而实现一条边只需要用一个结点表示。

9.2.4 邻接多重表 (Adjacency MultiList)

- 邻接多重表----无向图的链式存储
  - ➤**顶点的结构表示**

data	firstedge

```
typedef struct VexBox {
   VertexType  data;
   EBox  *firstedge;
          // 指向第一条依附该顶点的边
} VexBox;
```

9.2.4 邻接多重表 (Adjacency MultiList)

- 邻接多重表----无向图的链式存储

  ➢**无向图的结构表示**

```
typedef struct {  // 邻接多重表
    VexBox  adjmulist[MAX_VERTEX_NUM];
    int   vexnum, edgenum;
} AMLGraph;
```

特点： 邻接多重表的边结点表示同时包含了两个顶点信息，同时每个边结点同时链接在两个链表中。对于无向图而言，邻接多重表和邻接表的区别：同一条边在邻接表中用两个结点表示（对应于两个顶点的边链表），而在邻接多重表中只用一个结点表示。

第九章 图

9.1 图的定义和术语（集合与图论）

9.2 图的存储结构

9.3 图的遍历

9.4 有向无环图的应用

9.5 最短路径

9.3 图的遍历

9.3.1 深度优先搜索

9.3.2 广度优先搜索

从图中某个顶点出发，访问图中每个顶点，且每个顶点仅被访问一次。
图的遍历是求解的图的连通性、拓扑排序和求关键路径等算法的基础。

9.3 图的遍历

9.3.1　深度优先搜索

DFS (Depth First Search)

9.3.2　广度优先搜索

BFS (Breadth First Search)

9.3.1 深度优先搜索DFS (Depth First Search)

DFS类似于树的先根遍历

- 算法的基本思想

(1)首先访问图中某一个顶点V_i，以该顶点为出发点；

(2)任选一个与顶点V_i邻接的未被访问的顶点V_j；访问V_j；

(3)以V_j为新的出发点继续进行深度优先搜索，直至图中所有和V_i有路径的顶点均被访问到。

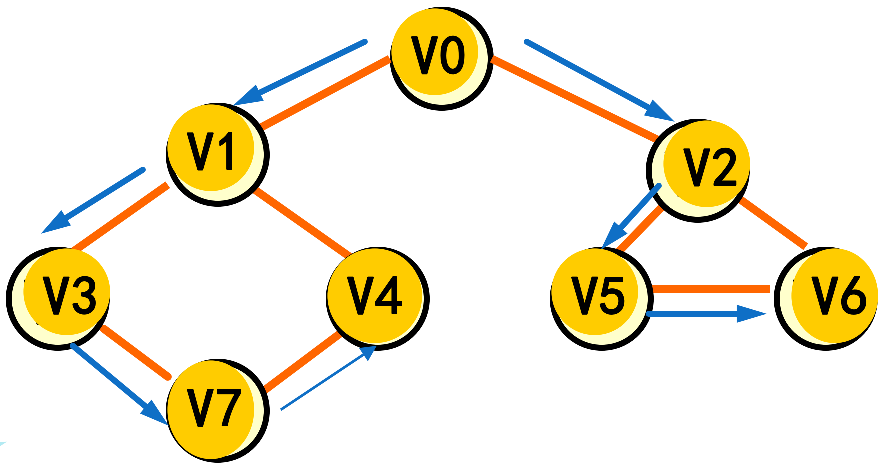
9.3.1 深度优先搜索DFS (Depth First Search)

例 求图G以V0起点的的深度优先序列：

V0, V1, V3, V7, V4, V2, V5, V6,

V0, V1, V4, V7, V3, V2, V5, V6

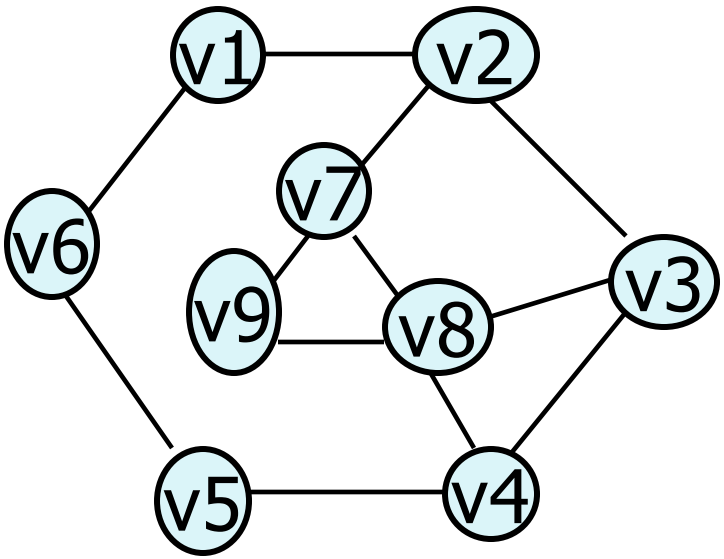
V4访问之后，由于V4的邻接点已经全部访问，搜索回溯到V7，同理，搜索继续回到V3，V1，V0，由于此时V0的另一个邻接点V2未被访问，则搜索从V2继续。

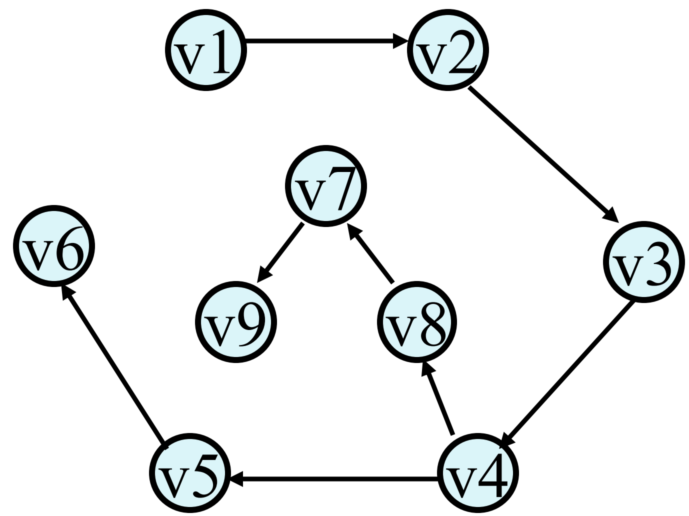


由于没有规定
访问邻接点的顺序，
深度优先序列不是唯一的

9.3.1 深度优先搜索DFS (Depth First Search)

- 深度优先搜索的示例演示





注意搜索次序的回溯

Visited[9]

1	2	3	4	5	6	7	8	9
1	1	1	1	1	1	1	1	1

9.3.1 深度优先搜索DFS (Depth First Search)

- 深度优先搜索算法概要

从某点v(设v为某顶点编号)出发。

步骤:

1. 访问顶点i;

2. 改变访问标志;

3. 任选一个与i相邻又没被访问的顶点j,从j开始继续进行深度优先搜索。

9.3.1 深度优先搜索DFS (Depth First Search)

深度优先搜索算法——图用邻接矩阵存储

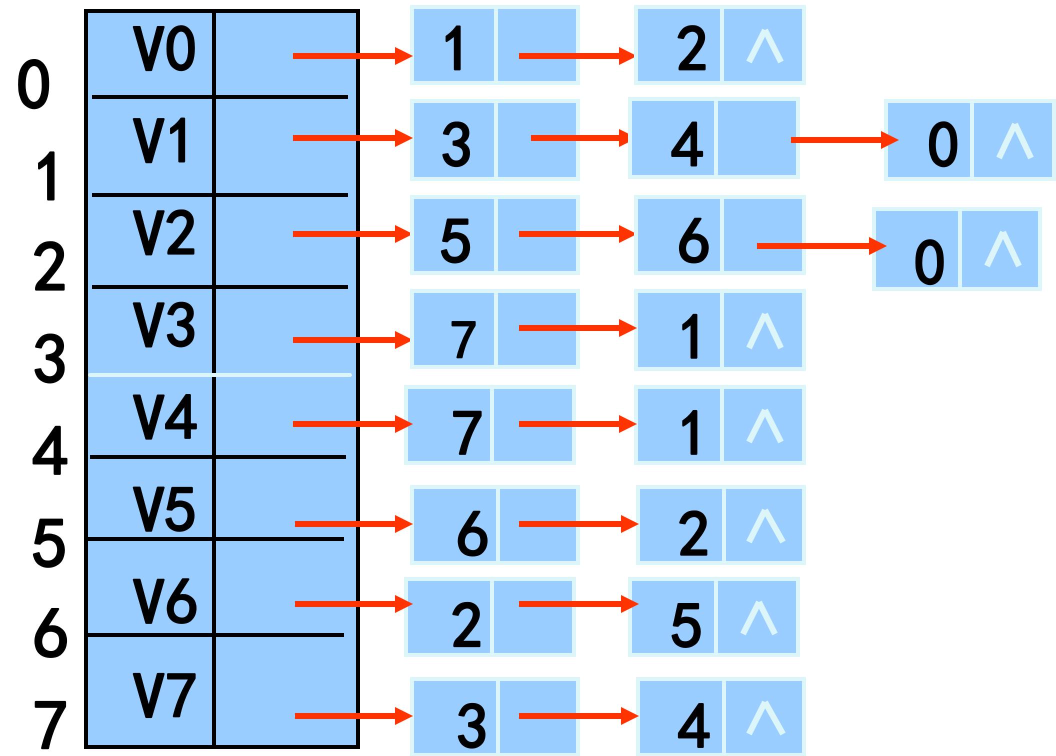
```
int visited[n]; graph g;
DFS(int i) {                              //图用邻接矩阵存储
    int j;
    printf("node:%c\n", g.vexs[i]);
    visited[i]=TRUE;
    for (j=0;j<n;j++)
        if ((g.arcs[i][j]==1)&&(!visited[j]))
            DFS(j);
}
```

采用递归算法实现搜索的回溯，并设置访问标志数组visited[]。

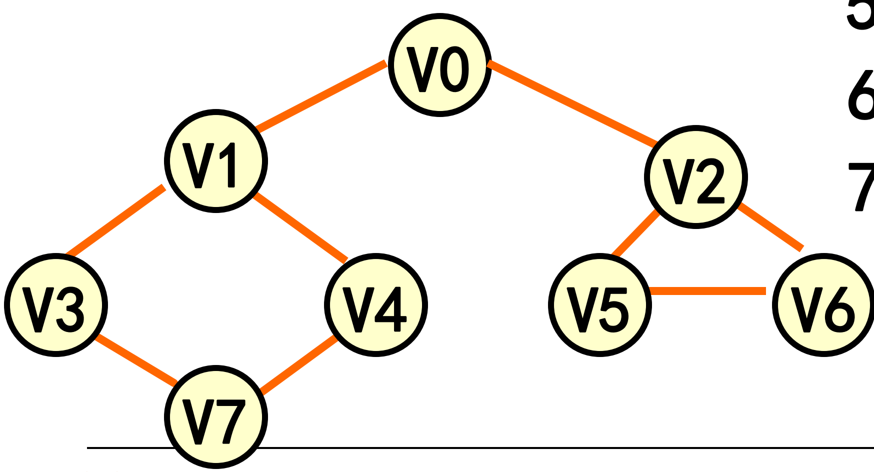
图的邻接矩阵表示是唯一的，由深度优先算法得到的DFS 序列是唯一的

9.3.1 深度优先搜索DFS (Depth First Search)

深度优先搜索算法——图用邻接表存储



请给出图的邻接表



以邻接表为存储结构，则查找邻接点的操作实际是顺序查找链表

9.3.1 深度优先搜索DFS (Depth First Search)

深度优先搜索算法——图用邻接表存储

```
vexnode g[n]; //图用邻接表存储
DFSL(int i) {
   int j;  edgenode *p;
   printf("node:%c\n",g[i].vertex);
   visited[i]=TRUE;                          //标识当前结点为访问过
   p=g[i].link;                           //得到当前结点的一条边
   while (p!=NULL){
      if (!visited[p->adjvex])    //如果存在边并未被访问过
      DFSL(p->adjvex);
      p=p->next;
   }
}
```

图的邻接表表示不是唯一的，由深度优先算法得到的DFS 序列不是唯一的

9.3.1 深度优先搜索DFS (Depth First Search)

非连通图的深度优先搜索遍历

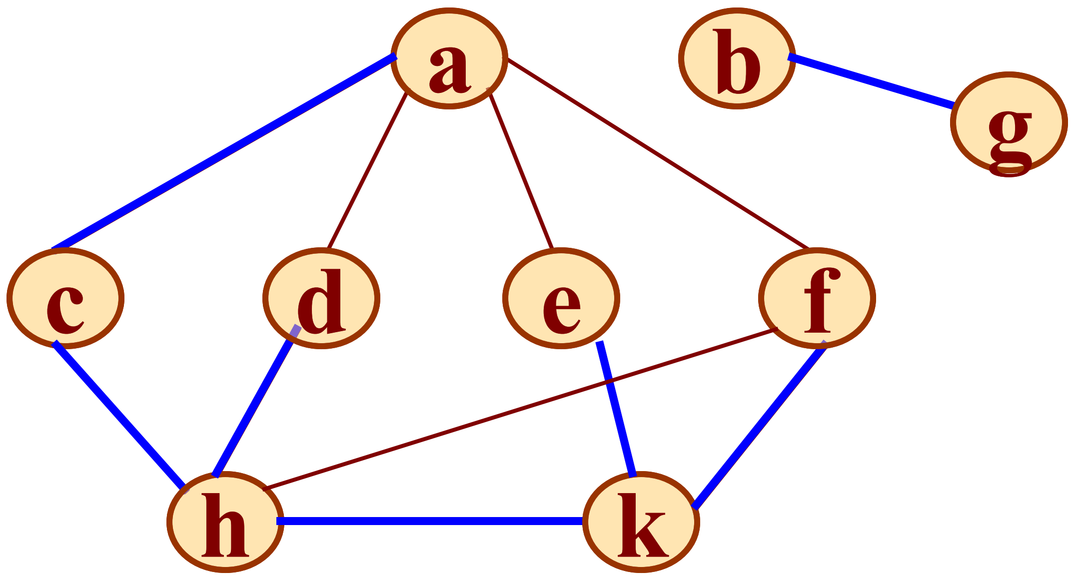
- **首先将图中每个顶点的访问标志设为 FALSE，之后搜索图中每个顶点**
- **若已被访问过，则该顶点一定是落在图中已求得的连通分量上；**
- **若还未被访问，则从该顶点出发遍历图，可求得图的另一个连通分量。**

如果一个无向图是非连通图，如何遍历？

9.3.1 深度优先搜索DFS (Depth First Search)

- 非连通图的深度优先搜索遍历

例如:



a	b	c	d	e	f	g	h	k
0	1	2	3	4	5	6	7	8

访问标志:

T	T	T	T	T	T	T	T	T

访问次序:

a	c	h	d	k	f	e	b	g

9.3.1 深度优先搜索DFS (Depth First Search)

- 非连通图的遍历

非连通图的遍历必须多次调用深度优先搜索或广度优先搜索算法，遍历算法如下：

```
TRAVER(){   // 遍历用邻接矩阵表示的非连通图
   int  i;
   for ( i = 0; i < n; i++)
      visited[i] = FALSE;  // 标志数组初始化
   for ( i = 0; i < n; i++)
      if  ( !visited[i])
         DFS(i);  // 从每一个顶点出发遍历一个连通分量
}
```

总结

利用深度优先搜索可以实现以下目标：

(1) 判断图是否连通

(2) 求图的连通分量

遍历图的本质是对每个顶点查找其邻接点的过程。当用邻接矩阵存储图时，查到邻接点的时间复杂性为O（n^2），当用邻接表存储图时，找邻接点时间为O（e），e为边（弧）数，因此，当用邻接表作为存储结构时，深度优先搜索遍历图的时间复杂度为：O（n+e）。

9.3 图的遍历

9.3.1 深度优先搜索

DFS (Depth First Search)

9.3.2 广度优先搜索

BFS (Breadth First Search)

9.3.2 广度优先搜索BFS (Breadth First Search)

- 广度优先搜索的思想

广度优先搜索BFS遍历类似于树的按层次遍历。

(1)首先访问图中某一个指定的出发点v_i;

(2)然后依次访问v_i的所有邻接点$v_{i1}, v_{i2} \ldots v_{it}$;

(3)再依次以$v_{i1}, v_{i2} \ldots v_{it}$为顶点，访问各顶点未被访问的邻接点，依此类推，直到图中所有顶点均被访问为止。

(4)若此时图中尚有顶点未被访问，则另选图中一个未曾被访问的顶点作起始点，重复上述过程，直至图中所有顶点都被访问到为止。

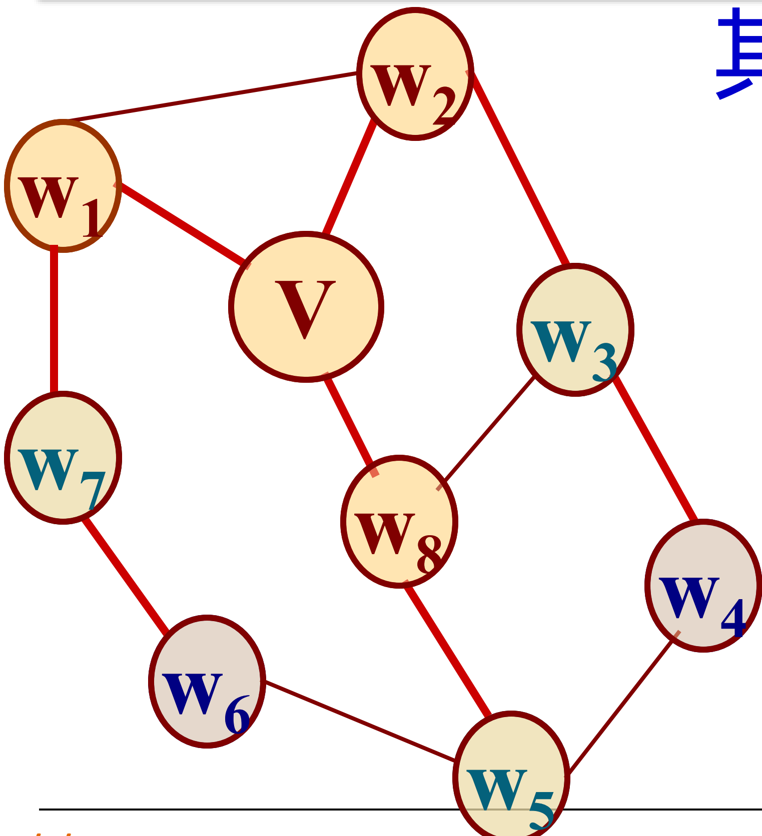
9.3.2 广度优先搜索BFS (Breadth First Search)

广度优先搜索遍历的过程好比“一石激起千重浪”。访问顶点v就好比将一块大石头扔进池塘的中央，必然激起浪花，这个浪花从中央向外四周扩散开来，渐渐波及池塘中从近到远的其它"石块"。

改变布局重新画图，将顶点v放在上方中央，则图的广度优先搜索遍历的过程类似于树的按层次遍历的过程。

9.3.2 广度优先搜索BFS (Breadth First Search)

对连通图，从起始点V到其余各顶点必定存在路径。由近及远，依次访问和V有路径相通且路径长度为1,2，..., 的顶点。



其中， $V\text{->}w_1$, $V\text{->}w_2$, $V\text{->}w_8$ 的路径长度为1；

$V\text{->}w_7$, $V\text{->}w_3$, $V\text{->}w_5$ 的路径长度为2；

$V\text{->}w_6$, $V\text{->}w_4$ 的路径长度为3。

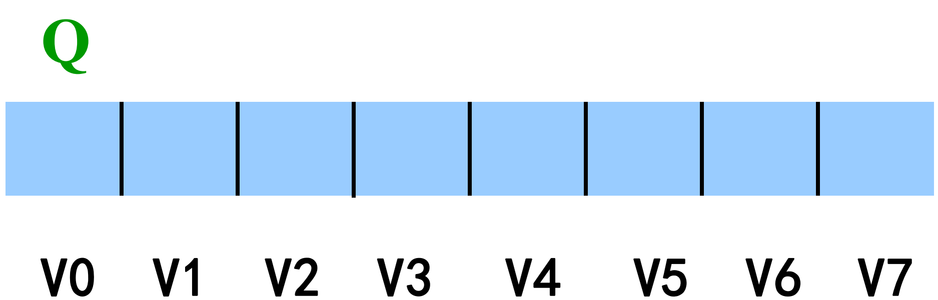
9.3.2 广度优先搜索BFS (Breadth First Search)

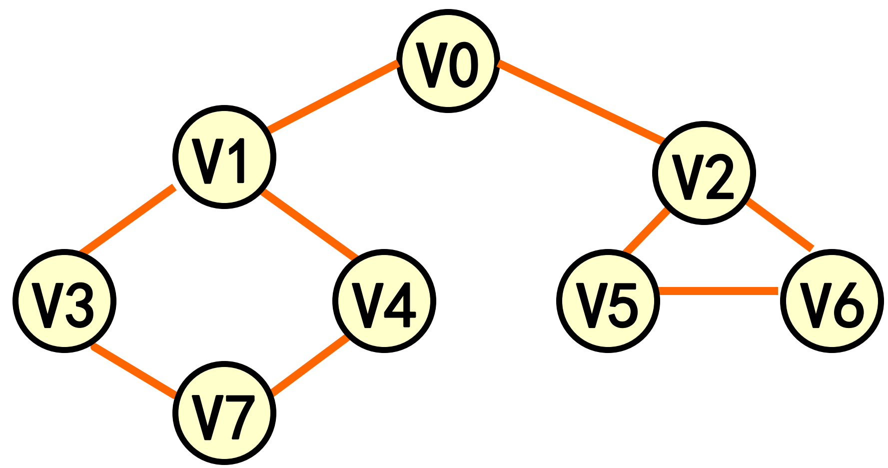
- 广度优先搜索的思想

> ➢为了实现逐层访问，算法中使用一个队列，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。
>
> ➢与深度优先搜索过程一样，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。

9.3.2 广度优先搜索BFS (Breadth First Search)

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。

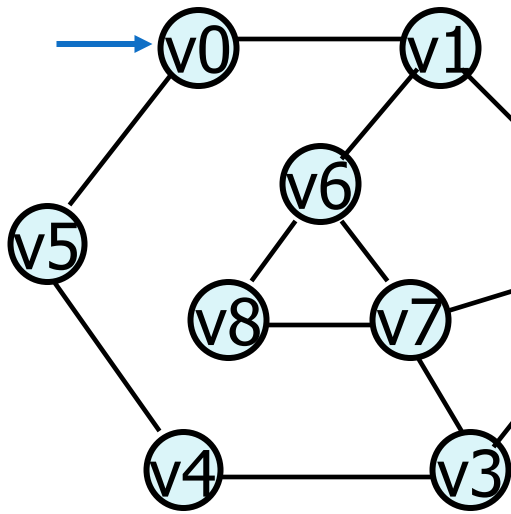


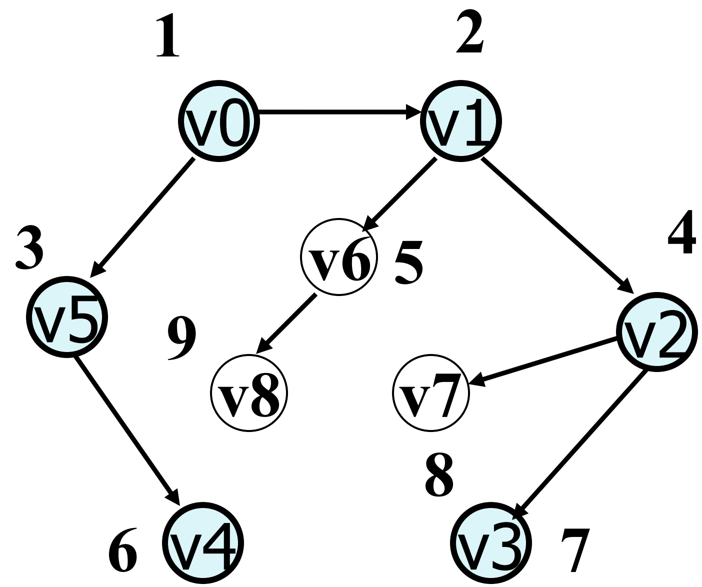


9.3.2 广度优先搜索BFS (Breadth First Search)

- 广度优先搜索示例演示

出发点





	0	1	2	3	4	5	6	7	8
Visit[v]	T	T	T	T	T	T	T	T	T
Q									
访问次序	V0	V1	V5	V2	V6	V4	V3	V7	V8

9.3.2 广度优先搜索BFS (Breadth First Search)

广度优先搜索算法——/*图用邻接矩阵表示*/

```
BSF(int k) {
  int i,j;
  SETNULL(Q);
  ENQUEUE(Q, k);//当前访问结点入队
  visited[k]=TRUE;
  while (!EMPTY(Q)) {
    i=DEQUEUE(Q);   //让当前结点出队
    for (j=0;j<n;j++)   //访问这一行所有结点
      if ((g.arcs[i][j]==1)&&(!visited[j])){ //如果当前结点有边且下一结点未被访问
        visited[j]=TRUE;
        ENQUEUE(Q, j);
      }
  }//while
}//BSF
```

如果使用邻接矩阵，则对于每一个被访问过的顶点，循环要检测矩阵中的 n 个元素，总的时间代价为$O(n^2)$。

9.3.2 广度优先搜索BFS (Breadth First Search)

广度优先搜索算法——/*图用邻接表表示*/

```
BFSL(int k) {
   int i; edgenode *p;
   SETNULL(Q);
   ENQUEUE(Q, k);
   visited[k]=TRUE;
   while (!EMPTY(Q)){
      i=DEQUEUE(Q);
      p=g1[i].link;
      while (p!=NULL) {  //访问p的整个链
            if (!visited[p->adjvex])
            {visited[p->adjvex]=TRUE;
             ENQUEUE(Q,p->adjvex);}
            p=p->next;
      }
   }
}
```

时间复杂度为O(n+*e*)

9.3.2 广度优先搜索BFS (Breadth First Search)

```
void BFSTraverse(Graph G, Status (*Visit)(int v)){
   for (v=0; v<G.vexnum; ++v)
      visited[v] = FALSE;         //初始化访问标志
   InitQueue(Q);                  // 置空的辅助队列Q
   for ( v=0;  v<G.vexnum;  ++v )
      if ( !visited[v]) {                 // v 尚未访问
      visited[u] = TRUE;  Visit(u);    // 访问u
      EnQueue(Q, v);                    // v入队列
      while (!QueueEmpty(Q))  {
         DeQueue(Q, u);  // 队头元素出队并置为u
         for(w=FirstAdjVex(G, u); w!=0; w=NextAdjVex(G,u,w))
            if ( ! visited[w])  {
               visited[w]=TRUE;  Visit(w);
               EnQueue(Q, w);     // 访问的顶点w入队列
            } // if
        } // while
   } //  for
}  // BFSTraverse
```

数据结构书中P170
算法7.6代码

从每个顶点出发，可以求得
非连通图的所有连通分量

结论

- 当无向图为非连通图时，从图中某一顶点出发，利用深度优先搜索算法或广度优先搜索算法不可能遍历到图中的所有顶点，只能访问到该顶点所在的最大连通子图(连通分量)的所有顶点。
- 若从无向图的每一个连通分量中的一个顶点出发进行遍历，可求得无向图的所有连通分量。
- 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。

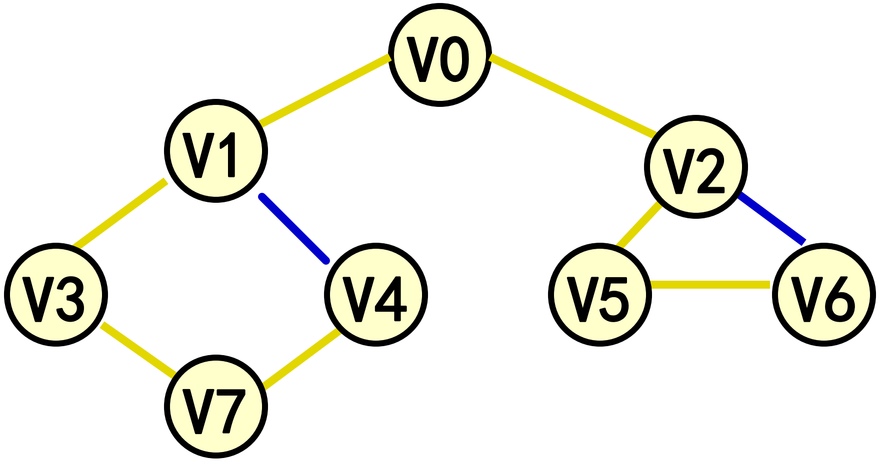
结论

- 深度优先或广度优先可用于求得无向图的生成树

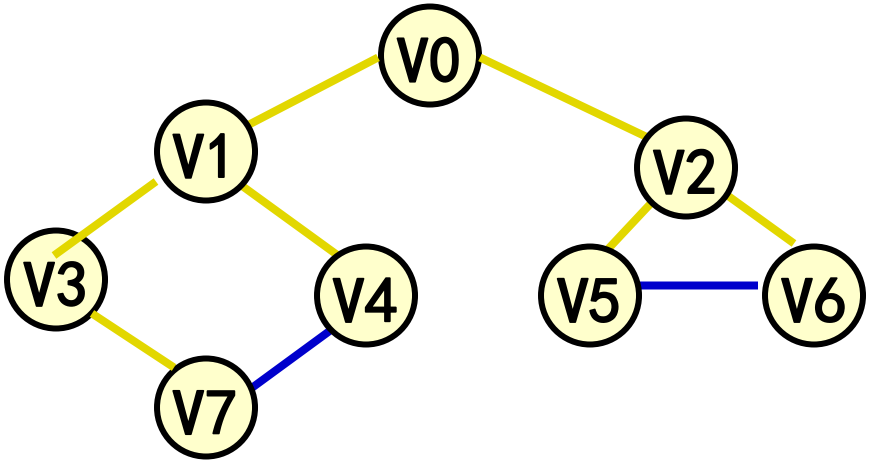
生成树可由遍历过程中所经过的边组成

➢深度优先生成树

➢广度优先生成树



深度优先生成树



广度优先生成树

第九章 图

9.1 图的定义和术语（集合与图论）

9.2 图的存储结构

9.3 图的遍历

9.4 有向无环图的应用

9.5 最短路径

9.4 有向无环图的应用

9.4.1 拓扑排序 (Topological Sort)

9.4.2 关键路径 (Critical Path)

9.4 有向无环图的应用

◘有向无环图

问题提出:一个无环的有向图称作有向无环图（directed acycline graph ），简称DAG图。

DAG图在工程计划和管理方面应用广泛。几乎所有的工程(project)都可分为若干个称作“活动”的子工程， 并且这些子工程之间通常受着一定条件的约束。如某些子工程开始必须在另一些子工程完成之后。

9.4 有向无环图的应用

□ 问题提出

某些子工程必须在另外的一些子工程完成之后才能开始。对整个工程和系统，人们主要关心的是两方面的问题：

（1）工程能否顺利进行； 拓扑排序

（2） 完成整个工程所必须的最短时间。

关键路径

9.4.1 拓扑排序

何谓“拓扑排序”？

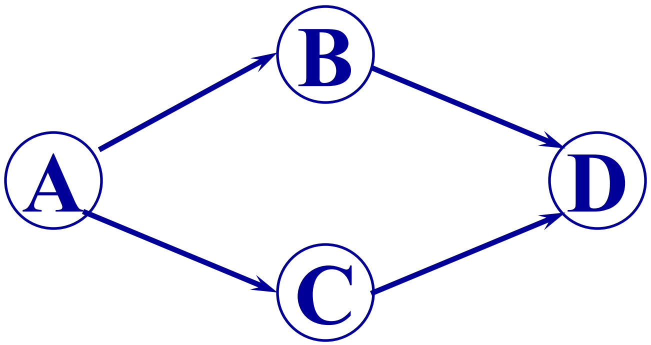
对有向图进行如下操作：

按照有向图给出的次序关系，将图中顶点排成一个线性序列，对于有向图中没有限定次序关系的顶点，则可以人为加上任意的次序关系。

由此所得顶点的线性序列称之为拓扑有序序列

9.4.1 拓扑排序

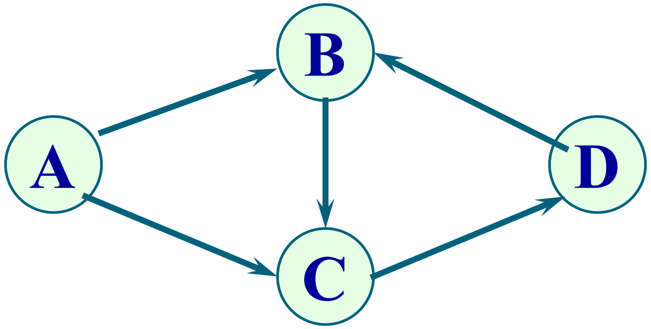
例1：求有向图的拓扑序列



可求得拓扑有序序列：

A B C D　或　A C B D

例2：求有向图的拓扑序列



不能求得它的拓扑有序序列。

因为图中存在一个回路 {B, C, D}

不是DAG

9.4.1 拓扑排序

□ 拓扑排序事例分析

计算机专业课程次序的安排就是一个简单的工程，每一门课程的学习都是整个工程的活动。

如何安排课程学习的先后次序是一个典型的拓扑排序问题。

9.4.1 拓扑排序

拓扑排序事例分析

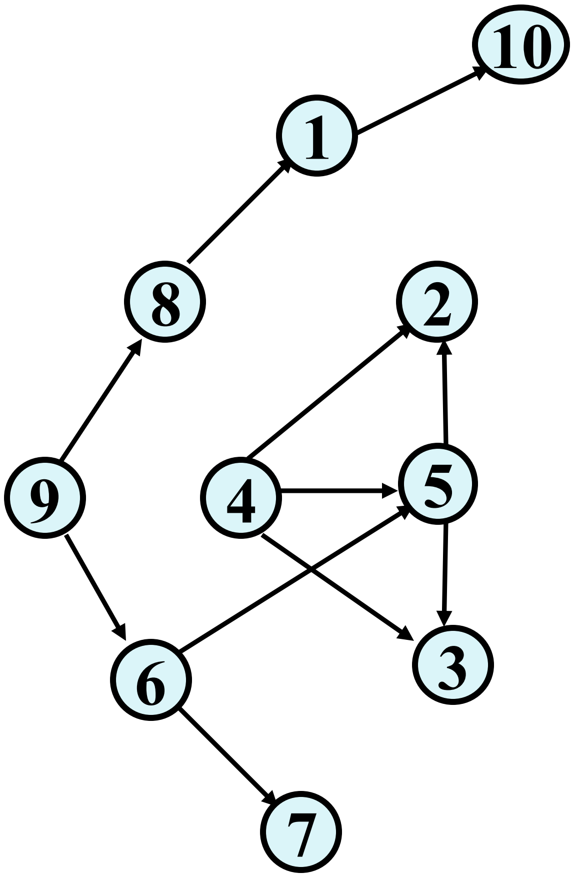
编号	课程名称	先修课程
1	计算机原理	8
2	编译原理	4,5
3	操作系统	4,5
4	程序设计	无
5	数据结构	4,6
6	离散数学	9
7	形式语言	6
8	电路基础	9
9	高等数学	无
10	计算机网络	1

9.4.1 拓扑排序

拓扑排序事例分析

编号	先修课程编号
1	8
2	4,5
3	4,5
4	无
5	4,6
6	9
7	6
8	9
9	无
10	1

生成DAG图



9.4.1 拓扑排序

□ 拓扑排序有关概念

AOV网中不能有有向环，否则存在死循环

顶点活动网(Activity On Vertex Network,简称AOV网)：将顶点表示活动，边表示活动之间的关系的网称为顶点活动网。

拓扑序列：把AOV网中的所有顶点排成一个<u>线性序列，</u>该序列满足如下条件：若AOV网中存在从v_i到v_j的路径，则在该序列中，v_i必位于v_j之前。

拓扑排序：构造AOV网的拓扑序列的操作被称为拓扑排序。

9.4.1 拓扑排序

□ 拓扑排序算法的基本思想

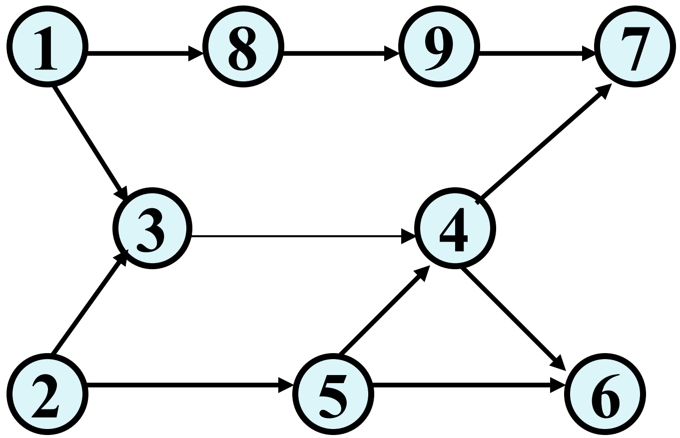
(1)在有向图中选一个入度为0的顶点输出。

(2)从图中删除该顶点及所有它的出边。

(3)重复执行a和b，直到全部顶点均已输出，或图中剩余顶点的入度均不为0(说明图中存在回路，无法继续拓扑排序)。

9.4.1 拓扑排序

◻ 拓扑排序算法的示例演示



① ② ③ ⑤ ④

⑥ ⑧ ⑨ ⑦

讨论：如果顶点还没有输出完而找不到入度为零的顶点怎么办？

答案：存在环路，无法进行拓扑排序，退出。

9.4.1 拓扑排序

拓扑排序特点

(1)一个有向图的拓扑序列不一定唯一;

(2)有向无环图一定存在拓扑序列;

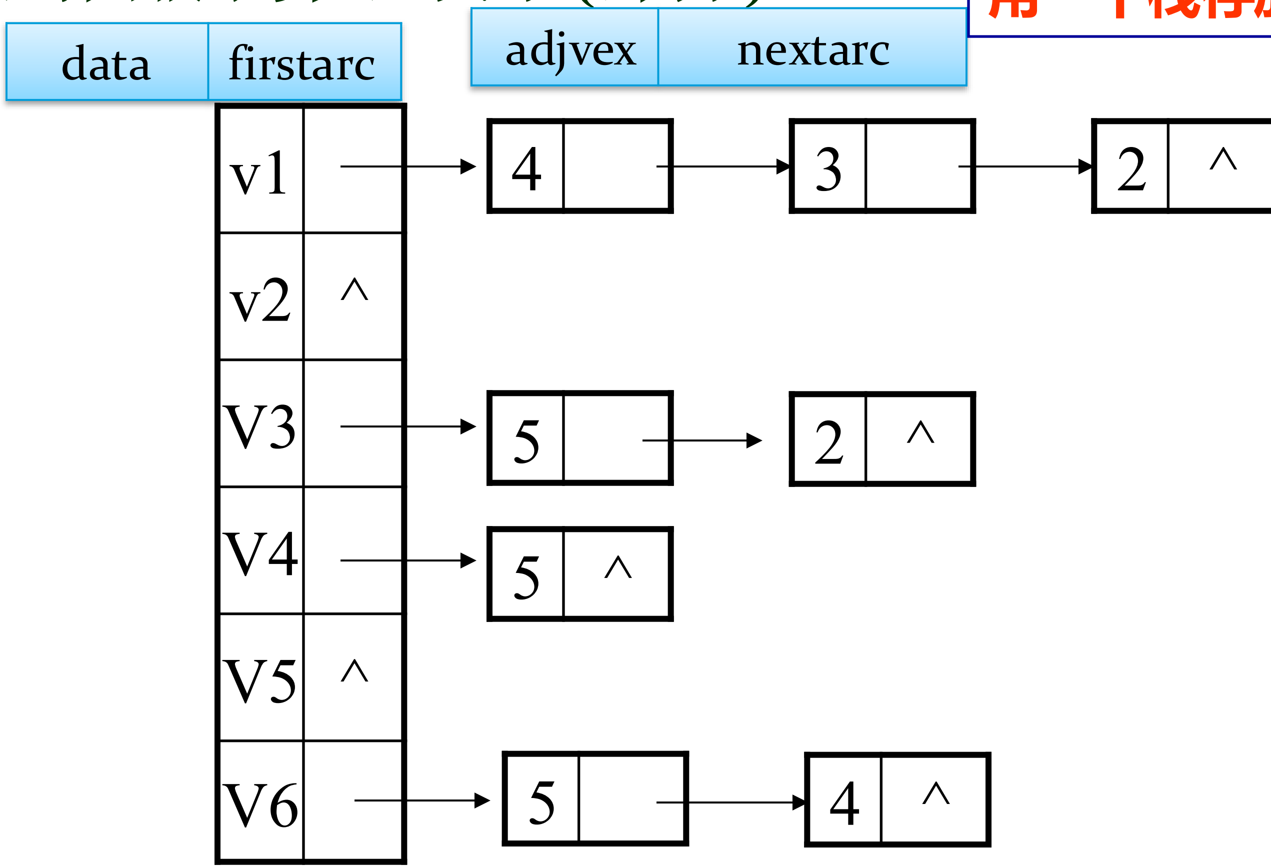
(3)有向有环图不存在拓扑序列;

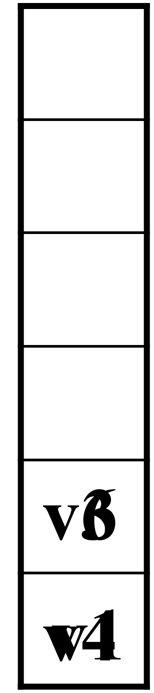
(4)通过构造拓扑序列，可判定AOV网是否存在环。

9.4.1 拓扑排序

□ 拓扑排序算法演示(部分)

用一个数组存放顶点入度
用一个栈存放入度为0的顶点





栈

入度已知

indegree[]

v1	v2	v3	v4	v5	v6
0	1	0	0	2	0

9.4.1 拓扑排序

□ 拓扑排序算法概要

增加一个存放各顶点入度的数组indegree[]

(1)扫描indegree[],将入度为零的顶点入栈;

(2)while(栈非空) {

- **弹出栈顶元素vi并输出;**
- **检查vi的出边表,将每条出边<vi,vj>的终点vj的入度减1,**
- **若vj的入度减至0,则vj入栈;**

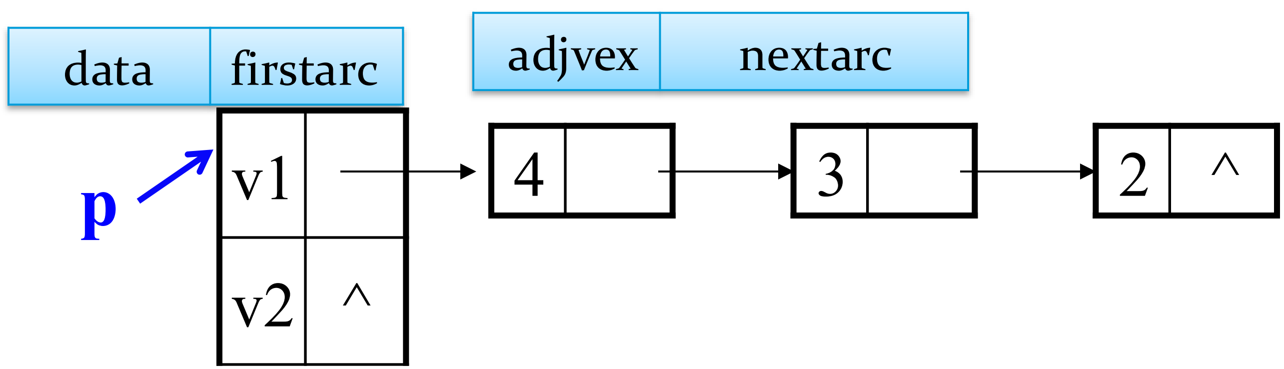
}

(3)若输出的顶点数小于n,则“有回路”；否则正常结束。

9.4.1 拓扑排序

□ 求各顶点入度的算法详解

```
void FindInDegree(ALGraph G, int indegree[])
    //对各顶点求入度indegree[0..vernum-1]
{int i; ArcNode *p;
 for (i=0;i<G.VExnum;i++)
    {p=G.Vertices[i].firstarc;
      while (p)
       {indegree[p->adjvex]++;
        p=p->next;
       }
    }
}
```



9.4.1 拓扑排序

◻ 拓扑排序算法详解

```
Status TopologicalSort(ALGraph G) {
   SqStack S;  int count,k,i;  ArcNode *p;
   int indegree[MAX_VERTEX_NUM];
   FindInDegree(G, indegree);        // 对各顶点求入度
   InitStack(S);
   for (i=0; i<G.Vexnum; ++i)        // 建零入度顶点栈S
      if (!indegree[i]) Push(S, i);        // 入度为0者进栈
   count = 0;                        // 对输出顶点计数
   while (!StackEmpty(S)){
      Pop(S, i);
      printf(i, G.Vertices[i].data);  ++count; //输入i号顶点并计数
      for (p=G.vertices[i].firstarc;  p;  p=p->nextarc){//对i号顶点的每个邻接点的入度减1
         k = p->adjvex;
         if (!(--indegree[k])) Push(S, k); //入度为0的入栈
      }
   }
   if (count<G.Vexnum) return ERROR;   //有回路
   else return OK;
} // TopologicalSort
```

□ 存储结构

采用邻接表作为AOV网的存储结构，在表头增设一个入度域

data	id	firstarc

adjvex	nextarc

v1	0	→
v2	2	→
v3	1	→
v4	3	→
v5	2	→

9.4.1 拓扑排序

算法分析

设AOV网有n个顶点，e条边。

（1）对e条弧求各顶点的入度的时间复杂度是O(e)

（2）初始建立入度为0 的顶点栈，要检查所有顶点一次，执行时间为O(n)；

（3）排序中，若AOV网无回路，则每个顶点入、出栈各一次，每个边表结点被检查一次，执行时间为O(n+e)；

拓扑排序算法的时间复杂度为O(n+e)。

9.4 有向无环图的应用

□ 问题提出

拓扑排序-AOV网表示各工序的先后关系

假设以有向网表示一个施工流图，弧上的权值表示完成该项子工程所需时间。

问：哪些子工程项是"关键工程"？

即：哪些子工程项将影响整个工程的完成期限的。

9.4.2 关键路径

◻AOE(Activity On Edge)网：

带权的有向图无环图,顶点表示事件,边表示活动,权表示活动持续的时间。

◻AOE网的特点

(1)表示实际工程计划的AOE网应该是无回路的;
(2)只有一个入度为零的顶点(称作源点),表示整个活动开始;
(3)只有一个出度为零的顶点(称作汇点)表示整个活动结束。

9.5.2 关键路径

□ 关键路径

在AOE 网中，路径长度最长的路径称为关键路径。

□ 关键活动

关键路径上的活动都是关键活动（关键工程）。

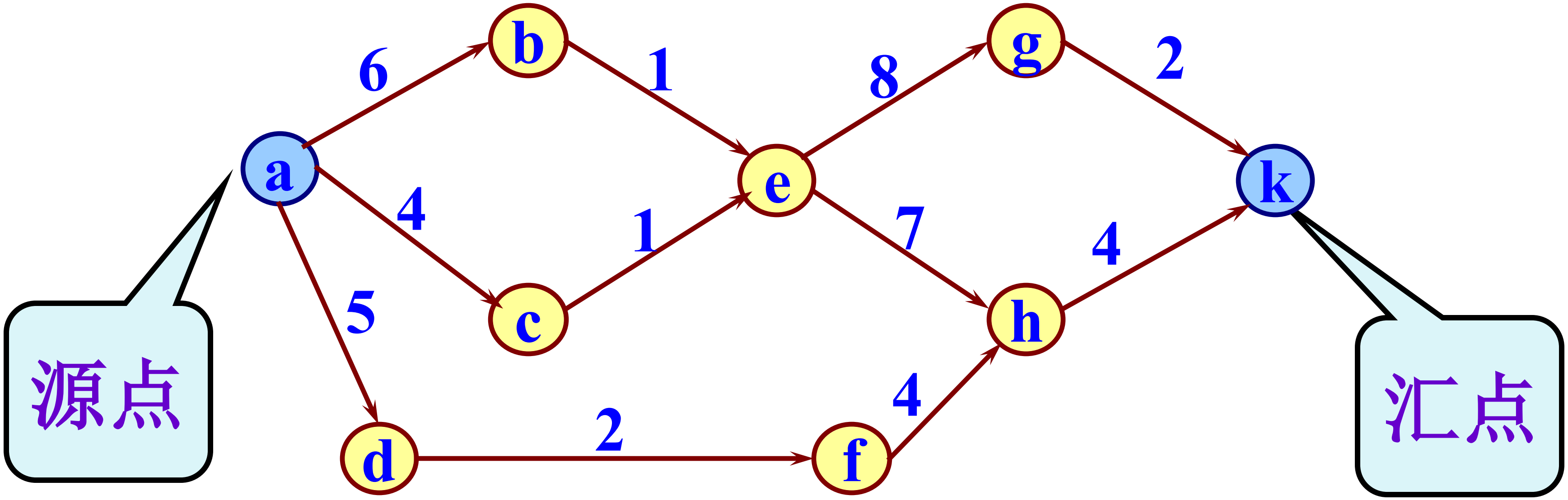
"关键活动"指的是：

该弧上（工序）的权值（加工时间）增加

将使有向图上的最长路径（工程）的长度增加。

9.5.2 关键路径

◻ AOE网图示



图中的弧表示子工程

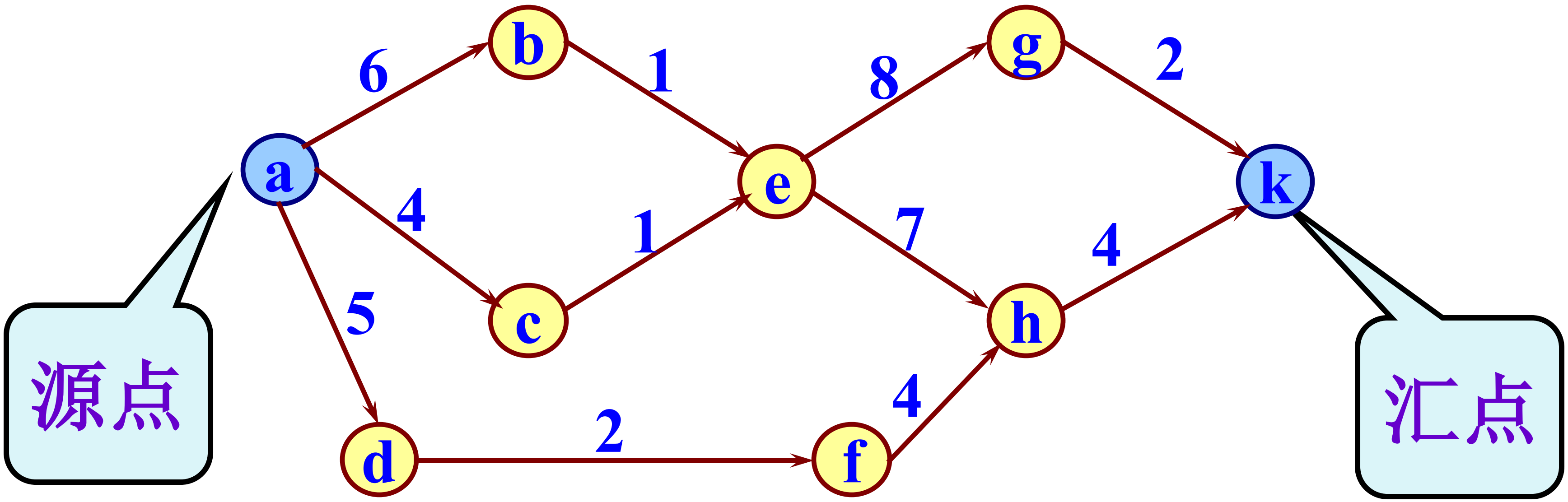
讨论：

(1)整个工程需要多少时间？

(2)哪些活动是影响工程进度的关键？

9.5.2 关键路径

◻ AOE网图示



答案:

最短时间是从源点到汇点的最长路径长度。

最长路径上的活动是影响工程进度的关键。

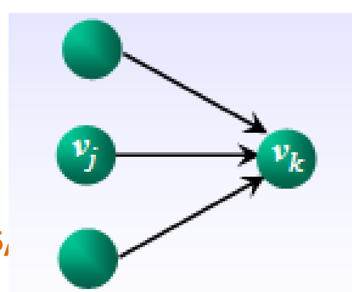
9.5.2 关键路径

□ 事件的最早发生时间

是从源点v1到vk的最长路径长度，记作ve(k)。这个长度决定了所有从顶点vk发出的活动能够开工的最早时间。

“事件(顶点)” 的 最早发生时间 *ve*(j)

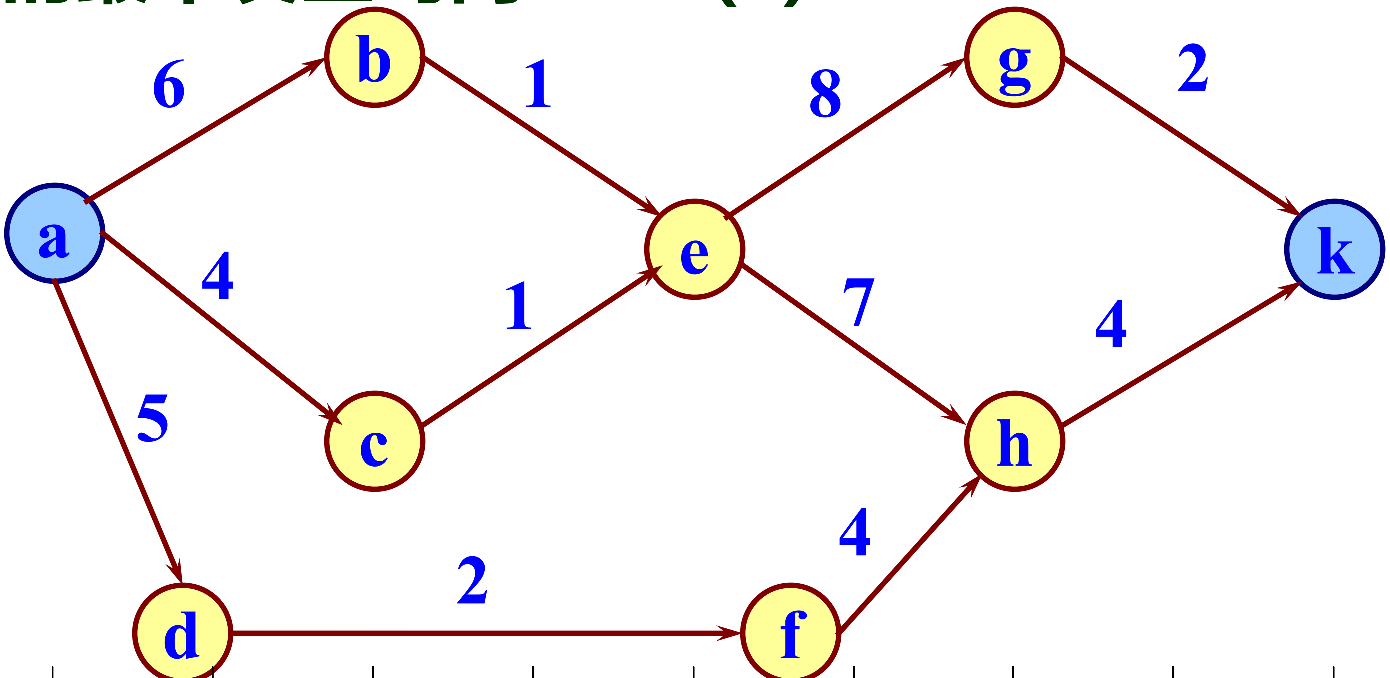
ve(j) = 从源点到顶点j的最长路径长度;



$$\begin{cases} ve[1]=0 \\ ve[k]=\max\{ve[j]+len\langle v_j, v_k\rangle\} \ (\langle v_j, v_k\rangle \in p[k]) \end{cases}$$

p[k]表示所有到达vk的有向边的集合

□ 事件的最早发生时间---ve(k)



	a	b	c	d	e	f	g	h	k
ve	0	6	4	5	7	7	15	14	18
vl									

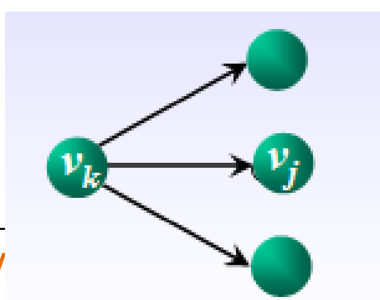
9.5.2 关键路径

- **事件的最迟发生时间—vl(k)**

是指在不推迟整个工期的前提下,事件vk允许的最晚发生时间。

从后往回计算(从汇点开始计算)

vl(k) = vn的最早发生时间ve(n)-vk到vn的最长路径长度

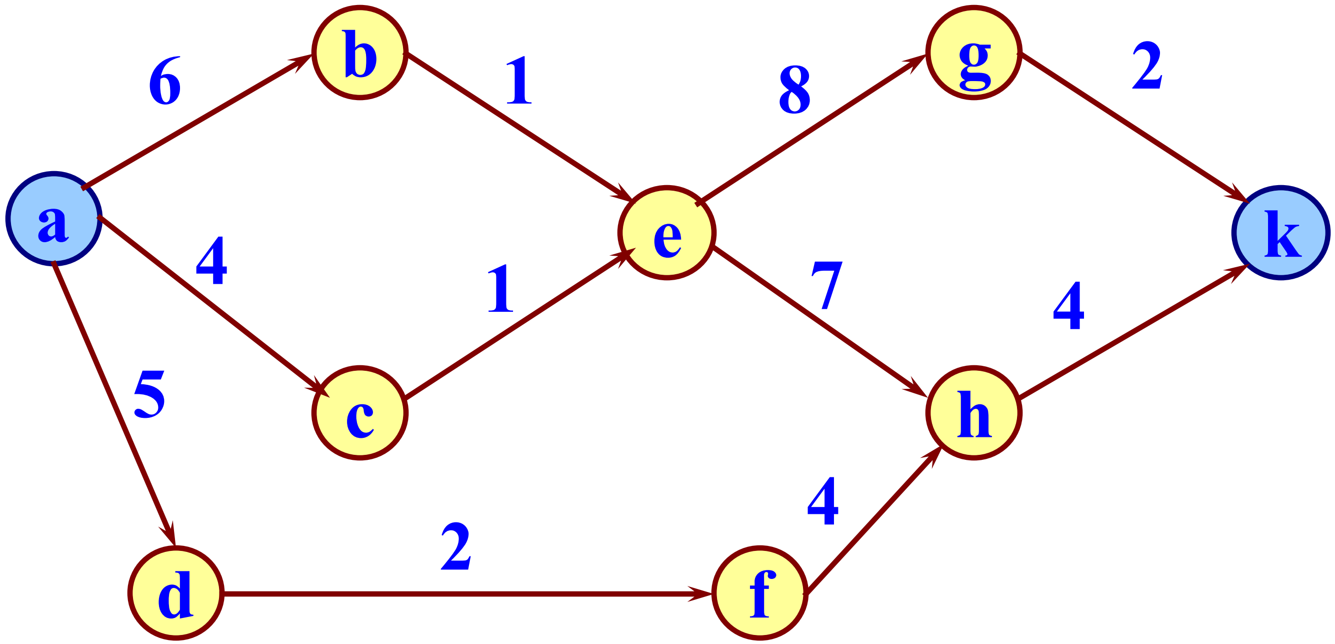


$$\begin{cases} vl[n]=ve[n] \\ vl[k]=\min\{vl[j]-len\langle v_k, v_j\rangle\} \quad (\langle v_k, v_j\rangle \in s[k]) \end{cases}$$

s[k]为所有从v_k发出的有向边的集合

9.5.2 关键路径

□ 事件的最迟发生时间—vl(k)



	a	b	c	d	e	f	g	h	k
ve	0	6	4	5	7	7	15	14	18
vl	0	6	6	8	7	10	16	14	18

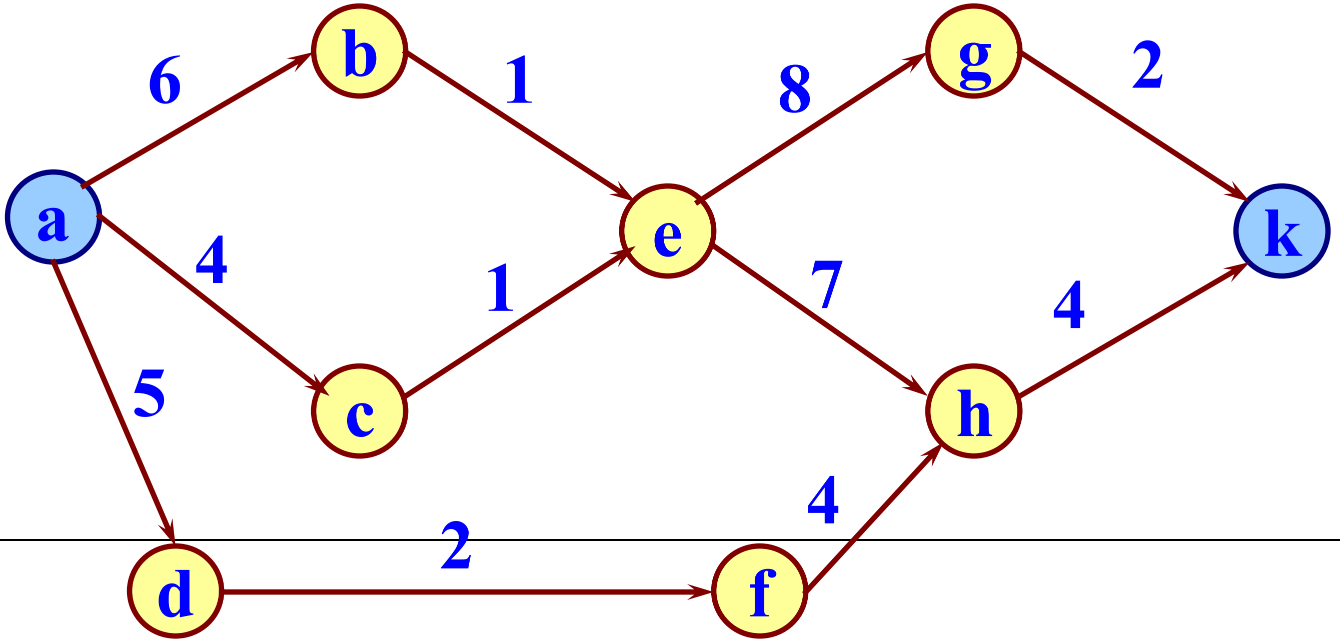
拓扑有序序列：a - d - f - c - b - e - h - g - k

9.5.2 关键路径

- **活动的最早开始时间—e[i]**

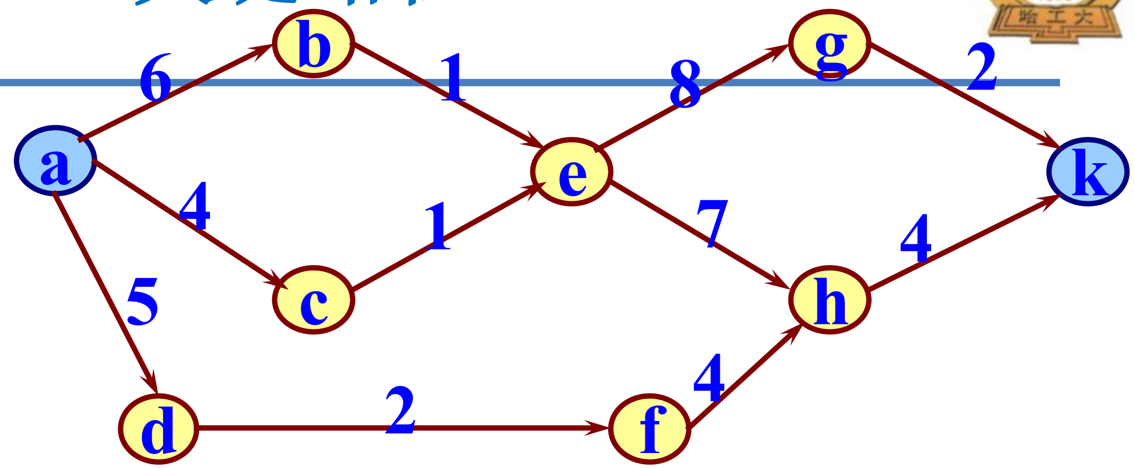
若活动ai是由弧<vk,vj>表示，则活动ai的最早开始时间应等于事件vk的最早发生时间。因此，有：

e[i] = ve[k]。



9.5.2 关键路径

□ 活动的最早开始时间—e[i]



事件	a	b	c	d	e	f	g	h	k
ve	0	6	4	5	7	7	15	14	18

	ab	ac	ad	be	ce	df	eg	eh	fh	gk	hk
权	6	4	5	1	1	2	8	7	4	2	4
e	0	0	0	6	4	5	7	7	7	15	14
l											

e[i] = ve[k]

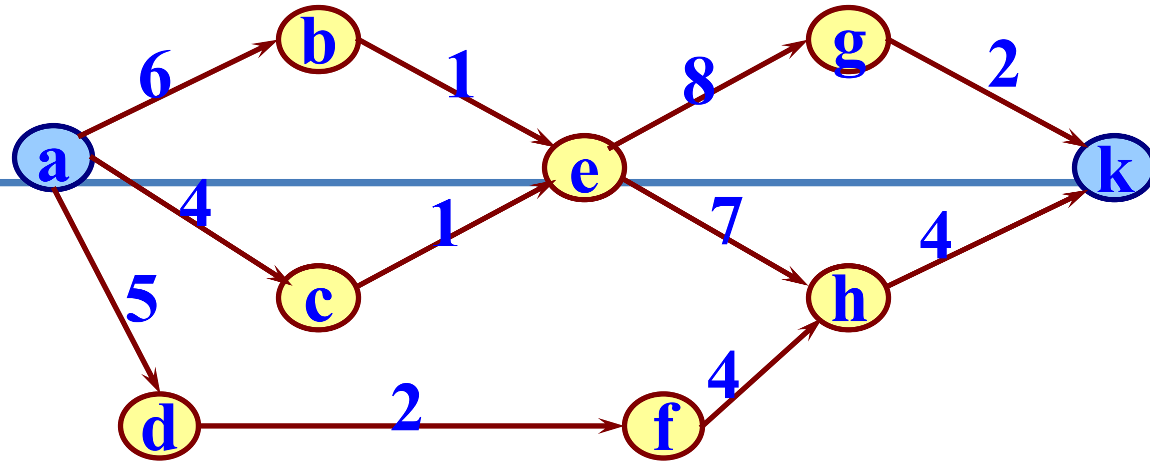
9.5.2 关键路径

- **活动的最晚开始时间--l[i]**

  在不推迟整个工期的前提下，ai必须开始的最晚时间。若ai由弧<vk，vj>表示，则ai的最晚开始时间要保证事件vj的最迟发生时间不拖后。

  因此，等于vj的最迟发生时间减去<vk,vj>的持续时间即

  $l[i]=vl[j]-dut<vk,vj>$



	a	b	c	d	e	f	g	h	k
vl	0	6	6	8	7	10	16	14	18

弧	ab	ac	ad	be	ce	df	eg	eh	fh	gk	hk
权	6	4	5	1	1	2	8	7	4	2	4

l[i]=vl[j]-dut<vk,vj>

l	0	2	3	6	6	8	8	7	10	16	14
	6-6	6-4	8-5	7-1	7-1	10-2	16-8	14-7	14-4	18-2	18-4
	✓			✓				✓			✓

9.5.2 关键路径

最早开始时间=最晚开始时间的活动为关键活动。

如何求关键活动？

9.5.2 关键路径

□ 如何求关键活动?

“事件(顶点)” 的 最早发生时间 *ve*(j)

ve(j) = **从源点到顶点j的最长路径长度;**

“事件(顶点)” 的最迟发生时间 *vl*(k)

vl(k) = **汇点最早发生时间-vk到汇点的最长路径长度**

9.5.2 关键路径

□ 如何求关键活动?

假设第 i 条弧为 <j, k>（弧尾是j ）

则 对第 i 项活动而言

“活动(弧)”的 最早开始时间 *e*(i)

e(i) = ve(j)；

//弧尾j与顶点j的最早开始时间相同

“活动(弧)”的最迟开始时间 *l*(i)

l(i) = vl(k) – dut(<j,k>)；

9.5.2 关键路径

□ 如何求关键活动?

事件发生时间的计算公式:

$$ve(\text{源点}) = 0;$$

$$ve(k) = \text{Max}\{ve(j) + dut(<j, k>)\}$$

$$vl(\text{汇点}) = ve(\text{汇点});$$

$$vl(j) = \text{Min}\{vl(k) - dut(<j, k>)\}$$

9.5.2 关键路径

- 求关键活动算法要点

拓扑逆序序列即为拓扑有序序列的逆序列

- **求ve的顺序是按拓扑有序的次序；**
- **求vl的顺序是按拓扑逆序的次序；**

方法：

在拓扑排序的过程中，另设一个“栈”记下拓扑有序序列。

9.5.2 关键路径

- 求关键活动算法要点

(1)求出每个事件i的最早发生时间ve(i)和最晚发生vl(i)

(2)求出每个活动最早开始时间e(i)和最晚开始时间l(i):

e(i)=ve[j]

假设第 i 条弧为 <j, k>（弧尾是j ）

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

(3)比较e(i)和l(i)， e(i)和l(i)相等的活动即为关键活动。

9.5.2 关键路径

- 求事件的最早发生时间和最晚发生时间步骤

第一步为前进阶段:从ve[1]=0开始,沿着路径上每条边的方向,用如下公式求每个事件的最早发生时间:

ve[j]= $\max_{i}\{ve[i]+dut(<i,j>)\}$

第二步为回退阶段:从已求出的vl[n]=ve[n]开始,沿着路径上每条边的相反方向,用如下公式求每个事件的最迟发生时间:

vl[i]= $\min_{j}\{vl(j)-dut(<i,j>)\}$

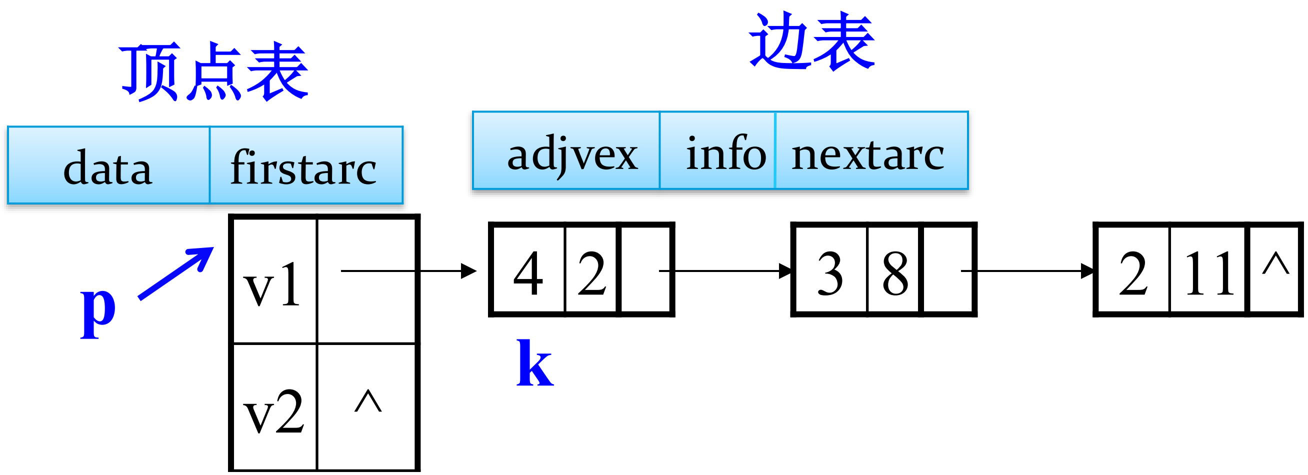
9.5.2 关键路径

- 关键路径算法所用数据结构

float ve[n],vl[n];

//记录事件的最早发生时间和最晚发生时间

邻接表的存储表示



9.5.2 关键路径

关键路径算法详解

status TopologicalOrder(ALGraph G, Stack &T)

```
//修改后的拓扑排序只是添加了求事件的最早开始时间
Stack S;int count=0,k;
 char indegree[40];  ArcNode *p;
 InitStack(S);
 FindInDegree(G, indegree);
 for (int j=0; j<G.vexnum; ++j)
    if (indegree[j]==0) Push(S, j); //建零入度顶点栈S， 入度为0者进栈
 InitStack(T);                    //建拓扑序列顶点栈T
 count = 0;
 for (int i=0; i<G.vexnum; i++) ve[i] = 0;     // 初始化
 while (!StackEmpty(S)) {
    Pop(S, j);Push(T, j);++count;
    for (p=G.vertices[j].firstarc; p; p=p->nextarc)
     { k = p->adjvex;
                   // 对j号顶点的每个邻接点的入度减1
```

□ 关键路径算法详解

接上页

```
if (--indegree[k] == 0) Push(S, k);
                                    // 若入度减为0，则入栈
   if (ve[j]+p->info > ve[k])//求事件的最早开始时间ve
         ve[k] = ve[j]+p->info;// *p->info=dut(<j,k>)
}
}
 if (count<G.vexnum) return ERROR;
 else return OK;
} // TopologicalOrder
```

修改后的拓扑排序

```
Status TopologicalOrder(ALGraph G, Stack &T)
{
    // 有向网G采用邻接表存储结构，求各顶点事件的最早发生时间ve(全局变量)。
    // T为拓扑序列定点栈，S为零入度顶点栈。
    // 若G无回路，则用栈T返回G的一个拓扑序列，且函数值为OK，否则为ERROR。
    Stack S;
    int count=0,k;
    char indegree[40];
    ArcNode *p;
    InitStack(S);
    FindInDegree(G, indegree);  // 对各顶点求入度indegree[0..vernum-1]
    for (int j=0; j<G.vexnum; ++j)     // 建零入度顶点栈S
        if (indegree[j]==0)
            Push(S, j);  // 入度为0者进栈
    InitStack(T);//建拓扑序列顶点栈T
    count = 0;
    for(int i=0; i<G.vexnum; i++)
        ve[i] = 0;  // 初始化
    while (!StackEmpty(S))
    {
        Pop(S, j);  Push(T, j);  ++count;        // j号顶点入T栈并计数
        for (p=G.vertices[j].firstarc;  p;  p=p->nextarc)
        {
            k = p->adjvex;               // 对j号顶点的每个邻接点的入度减1
            if (--indegree[k] == 0) Push(S, k);   // 若入度减为0，则入栈
            if (ve[j]+p->info > ve[k])  ve[k] = ve[j]+p->info;
        }//for  *(p->info)=dut(<j,k>)
    }
    if(count<G.vexnum)
        return ERROR;  // 该有向网有回路
    else
        return OK;
}
```

修改后的拓扑排序

9.5.2 关键路径

关键路径算法详解

```
Status CriticalPath(ALGraph G) {
            // G为有向网，输出G的各项关键活动。
 Stack T;
 int a,j,k,el,ee,dut;
 char tag;
 ArcNode *p;
 if (!TopologicalOrder(G, T))
 return ERROR;
 for(a=0; a<G.vexnum; a++) // 初始化顶点事件的最迟发生时间
    vl[a] = ve[G.vexnum-1];
 while (!StackEmpty(T)) // 按拓扑逆序求各顶点的vl值
    for (Pop(T, j), p=G.vertices[j].firstarc;  p; p=p->nextarc){
        k=p->adjvex;  dut=p->info;             //dut<j,k>
        if (vl[k]-dut < vl[j])  vl[j] = vl[k]-dut;
     }// for
```

9.5.2 关键路径

关键路径算法详解

```
  for (j=0; j<G.vexnum; ++j) // 求ee,el和关键活动
     for (p=G.vertices[j].firstarc;  p; p=p->nextarc){
        k=p->adjvex;dut=p->info;
        ee = ve[j];  el = vl[k]-dut;
        tag = (ee==el) ? '*' : ' ';
        printf(j, k, dut, ee, el, tag); // 输出关键活动
     }
  return OK;
}// CriticalPath
```

9.5.2 关键路径

```
Status CriticalPath(ALGraph G)
{
    // G为有向网，输出G的各项关键活动。
    Stack T;
    int a,j,k,el,ee,dut;
    char tag;
    ArcNode *p;
    if (!TopologicalOrder(G, T))
        return ERROR;
    for(a=0; a<G.vexnum; a++)
        vl[a] = ve[G.vexnum-1];    // 初始化顶点事件的最迟发生时间
    while (!StackEmpty(T))       // 按拓扑逆序求各顶点的vl值
        for (Pop(T, j), p=G.vertices[j].firstarc;  p;  p=p->nextarc)
        {
            k=p->adjvex;  dut=p->info;     //dut<j,k>
            if (vl[k]-dut < vl[j])
                vl[j] = vl[k]-dut;
        }
    for (j=0; j<G.vexnum; ++j)              // 求ee,el和关键活动
        for (p=G.vertices[j].firstarc;  p;  p=p->nextarc)
        {
            k=p->adjvex;dut=p->info;
            ee = ve[j];  el = vl[k]-dut;
            tag = (ee==el) ? '*' : ' ';
            printf(j, k, dut, ee, el, tag);   // 输出关键活动
        }
    return OK;
}
```

9.5.2 关键路径

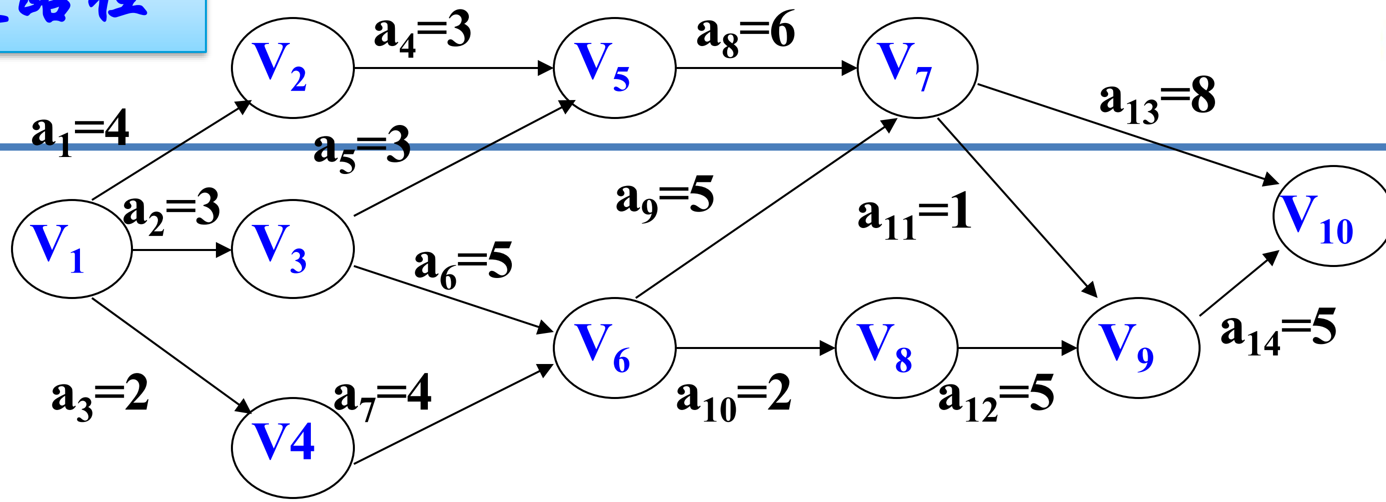
□ 算法分析

在拓扑排序求Ve[i]和逆拓扑有序求Vl[i]时，所需时间为O(n+e);

求各个活动的e[k]和l[k]时所需时间为O(e);

总共花费时间仍然是O(n+e)。

例：求关键路径



	V_1	V_2	V_3	V_4	V_5	V_6	V_7	V_8	V_9	V_{10}
	$(a_1a_2a_3)$	(a_4)	(a_5a_6)	(a_7)	(a_8)	(a_9a_{10})	$(a_{11}a_{13})$	(a_{12})	(a_{14})	
ve:	0	4	3	2	7	8	13	10	15	21
vl:	0	4	3	4	7	8	13	11	16	21
e:	(0 0 0)	(4)	(3 3)	(2)	(7)	(8 8)	(13 13)	(10)	(15)	21
l:	(0 0 2)	(4)	(4 3)	(4)	(7)	(8 9)	(15 13)	(11)	(16)	21

正向取最大值

反向取最小值

关键活动：a_1、a_2、a_4、a_6、a_8、a_9、a_{13};

关键路径：$V_1V_2V_5V_7V_{10}$ 和 $V_1V_3V_6V_7V_{10}$。

第九章 图

9.1 图的定义和术语（集合与图论）

9.2 图的存储结构

9.3 图的遍历

9.4 有向无环图的应用

9.5 最短路径

9.5 最短路径

➢**问题的提出：** 给定一个带权有向图G与源点v，求从v到G中其它顶点的最短路径。

应用：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

9.5 最短路径

9.5.1 单源最短路径(集合与图论)

9.5.2 每一对顶点之间的最短路径

9.6 最短路径

9.6.1 单源最短路径(集合与图论)

9.6.2 每一对顶点之间的最短路径

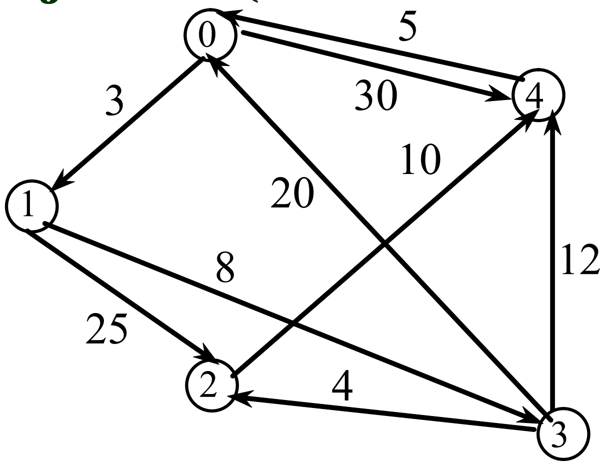
9.6.1从某源点到其余各点的最短路径

□Dijkstra 算法基本思想：

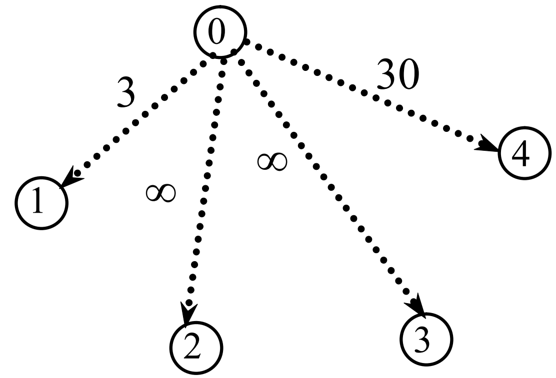
1. 集合S的初值为S={1}
2. D为各顶点当前最短路径
3. 从V-S中选择顶点w，使D[w]的值最小（选择权值最小的边加入）
4. 并将 w加入集合 S，则w的最短路径已求出。
5. 调整其它各结点的当前最短路径
6. D[k]=min{D[k], D[w]+C[w][k]}
7. 直到S中包含所有顶点

9.6.1 从某源点到其余各点的最短路径

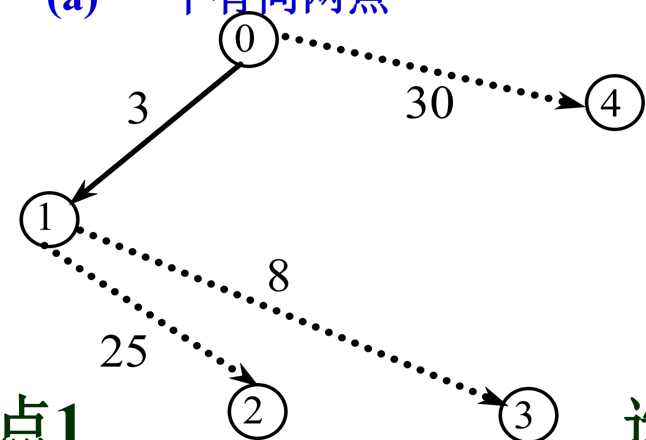
□Dijkstra(迪杰斯特拉)算法的求解过程



(a) 一个有向网点

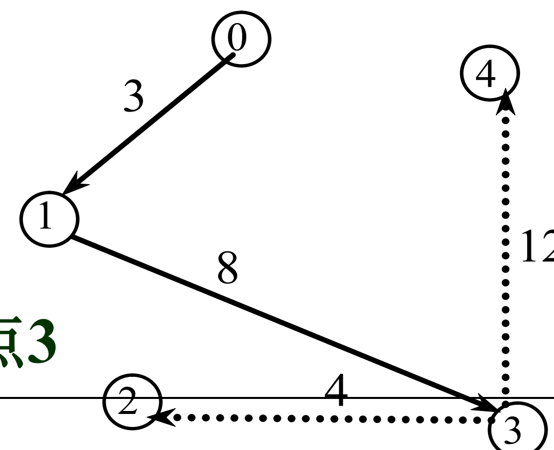


(b) 源点 0 到其它顶点的初始距离



选择顶点1

(c) 第一次求得的结果

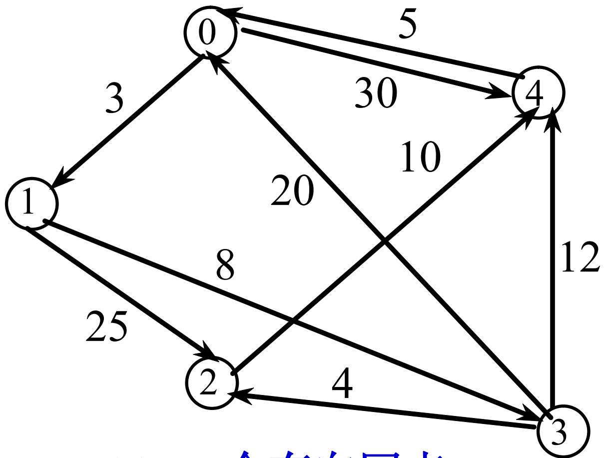


选择顶点3

(d) 第二次求得的结果

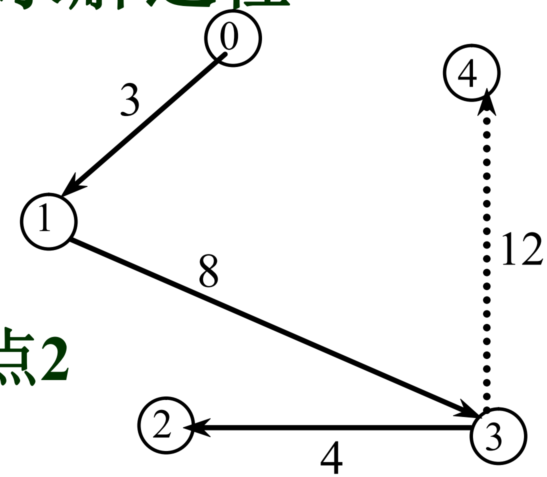
9.6.1 从某源点到其余各点的最短路径

□Dijkstra(迪杰斯特拉)算法的求解过程

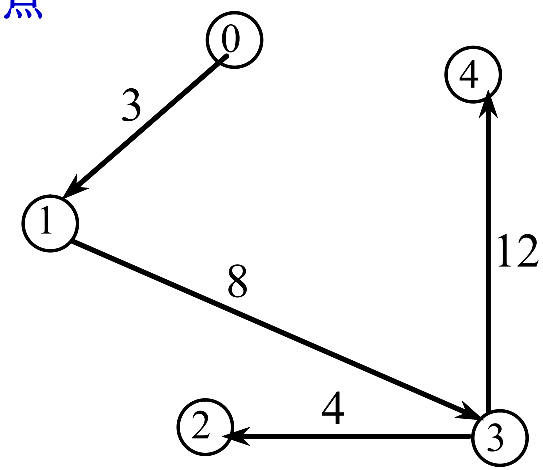


(a) 一个有向网点

选择顶点2



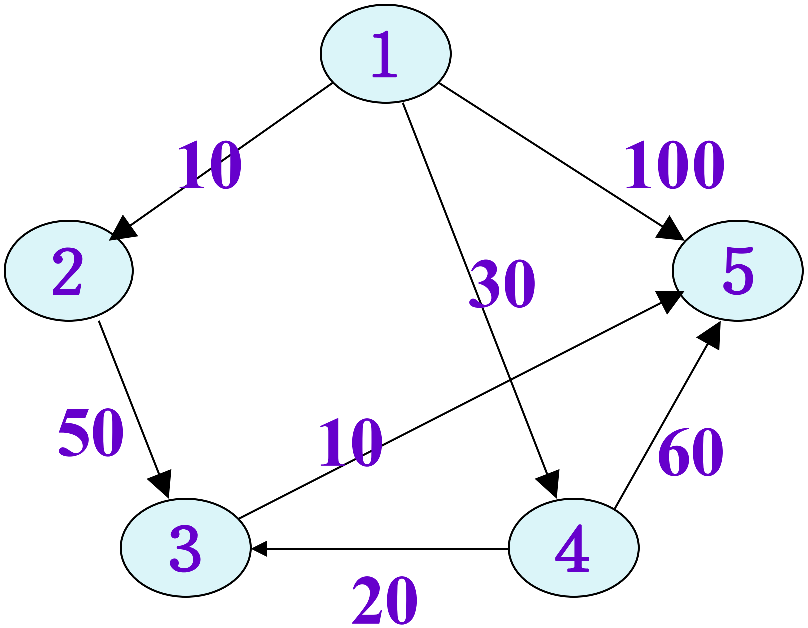
(e) 第三次求得的结果



选择顶点4

(f) 第四次求得的结果

9.6.1 从某源点到其余各点的最短路径



循环	源点集	k+1	D[0]…D[4]	P[]…P[4]
初始化	{4}	-	∞ ∞ 20 0 60	0 0 4 0 4
1	{4,3}	3	∞ ∞ 20 0 30	0 0 4 0 3
2	{4,3,5}	5	∞ ∞ 20 0 30	0 0 4 0 3

P[i-1]表示从源点到达i点的最短路径上该点的前驱顶点

9.6.1 从某源点到其余各点的最短路径

Dijkstra算法要点

1. 将V（顶点集合）分成两个集合S(开始只包含源点)和V-S。
2. 每一步从V-S中选择一结点w加入S，使S中从源点到其余结点的路长最短，此过程进行到V-S变成空集为止。

9.6.1 从某源点到其余各点的最短路径

Dijkstra算法概要

```
Void Dijkstra（C）//用邻接矩阵表示有向图G
{S={1}；
 for (i=2;i<=n; i++)
    D[i]=C[1][i];//D置初值
 for (i=1;i<=n-1; i++)
    {从V-S 中选出一个顶点w，使D[w]的值最小；
     把w加入S；
      for （V-S中的每个顶点v）
          D[v]=min（ D[v], D[w]+C[w][v] ）；
    }
}
```

9.6.1 从某源点到其余各点的最短路径

```
Void ShortestPath_DIJ(Graph G,int v0){
//求有向图G的v0顶点到其余顶点v的带权长度D[v]
//final[v]为TRUE当且仅当v∈S，即已经求得v0到v的最短路径
  for(v=0;v<G.vexnum;++v){
     final[v]=FALSE;
     D[v]=G.arcs[v0][v];
  }//初始化
  D[v0]=0； final[v0]=TRUE；  //从顶点v0出发，首先将v0加入S集
```

9.6.1 从某源点到其余各点的最短路径

```
//开始主循环，每次求得v0到某个v顶点的最短路径，并加v到S集
  for(i=1;i<G.vexnum;++i){ //其余G.vexnum-1个顶点
    min=INFINITY;                    //当前所知离v0顶点的最近距离
    for (w=0;w<G.vexnum;++w)
      if(!final[w]) //w顶点在V-S中，即顶点w还没有加入
        if（D[w]<min）{
          v=w;min=D[w]；  //w顶点离v0顶点更近
        }//选择最小的D[w]
    final[v]=TRUE;     //离v0顶点最近的v加入S集
    for (w=0;w<G.vexnum;++w)//更新当前最短距离
      if(！ final[w]&&(min+G.arcs[v][w]<D[w]))
        D[w]=min+G.arcs[v][w];
  }
}
```

时间复杂度 O（n^2）

9.6 最短路径

9.6.1 单源最短路径(集合与图论)

9.6.2 每一对顶点之间的最短路径

9.6.2 各顶点之间的最短路径

依次把有向网络的每个顶点作为源点，重复执行Dijkstra算法n次，即可求得每对顶点之间的最短路径。

是否有更简洁的方法?

Floyd算法

9.6.2 各顶点之间的最短路径

- **Floyd算法的基本想法：动态规划算法**

✓ 如果vi与vj之间有有向边，则vi与vj之间有一条路径，但不一定是最短的；也许经过某些中间点会使路径长度更短。

✓ 经过哪些中间点会使路径长度缩短呢?经过哪些中间点会使 路径长度最短呢?

- 只需尝试在原路径中间加入其它顶点作为中间顶点。

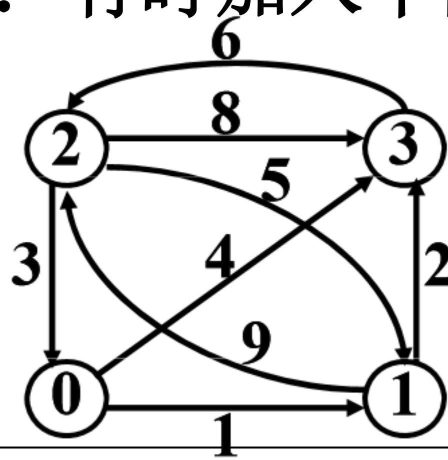
✓ 如何尝试?

- 系统地在原路径中间加入每个顶点，然后不断地调整当前 路径(和路径长度)即可。

9.6.2 各顶点之间的最短路径

▫示例:

- <2,1>5; <2,0><0,1>4; a[2][1]=a[2][0]+a[0][1] 调整
- 注意:考虑v0做中间点可能还会改变其它顶点间的距离:<2,0,3>7; <2,3>8 ; a[2][3]=a[2][0]+a[0][3]
- <2,3>：<2,0><0,3>：<2,0><0,1><1,3>=<2,0,1,3> a[2][3]=6 调整
- 注意：有时加入中间顶点后的路径比原路径长 保持



$$
\begin{array}{cc}
 & \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\
A= & \left(\begin{array}{cccc} \infty & 1 & \infty & 4 \\ \infty & \infty & 9 & 2 \\ 3 & 5 & \infty & 8 \\ \infty & \infty & 6 & \infty \end{array}\right) \begin{array}{c} 0 \\ 1 \\ 2 \\ 3 \end{array}
\end{array}
$$

9.6.2 各顶点之间的最短路径

□ Floyd算法的基本思想：

- 假设求顶点v_i到顶点v_j的最短路径。如果从 v_i 到 v_j 存在一条长度为C[*i*][*j*]的路径，该路径不一定是最短路径，尚需进行 *n* 次试探。
- 首先考虑路径 (v_i, v_0, v_j) 是否存在。如果存在，则比较 (v_i, v_j) 和 (v_i, v_0, v_j) 的路径长度取长度较短者为从 v_i 到 v_j 的中间顶点的序号不大于0的最短路径。
- 假设在路径上再增加一个顶点 v_1，也就是说，如果 $(v_i,\ldots, v_1)$和 $(v_1,\ldots, v_j)$ 分别是当前找到的中间顶点的序号不大于0的最短路径,那么 $(v_i,\ldots, v_1,\ldots, v_j)$就是有可能是从$v_i$到$v_j$的中间顶点的序号不大于1的最短路径。将它与已经得到的从v_i到v_j中间顶点序号不大于0的最短路径相比较，从中选出中间顶点的序号不大于1的最短路径，再增加一个顶点v_2, 继续进行试探。
- 一般情况下，若 $(v_i,\ldots, v_k)$ 和 $(v_k, \ldots, v_j)$ 分别是从 v_i到v_k和从v_k到v_j的中间顶点序号不大于 k-1 的最短路径，则将 $(v_i,\ldots, v_k,\ldots, v_j)$ 和已经得到的从 v_i到 v_j且中间顶点序号不大于k-1的最短路径相比较，其长度较短者便是从v_i到 v_j 的中间顶点的序号不大于 *k* 的最短路径。

9.6.2 各顶点之间的最短路径

□ Floyd算法的数据结构

- **图的存储结构：**
  - ✓ 带权的有向图采用邻接矩阵C[n][n]存储
- **数组D[n][n]：**
  - ✓ 存放在迭代过程中求得的最短路径长度。迭代公式：

$$\begin{cases} D_{-1}[i][j]=C[i][j] \\ D_k[i][j]=\min\{ D_{k-1}[i][j],\ D_{k-1}[i][k]+D_{k-1}[k][j] \}\quad 0\le k\le n-1 \end{cases}$$

- **数组P[n][n]：**
  - ✓ 存放从v_i到v_j求得的最短路径。初始时，P[i][j]=-1

9.6.2 各顶点之间的最短路径

□ Floyd算法思想

对顶点进行编号，设顶点为0,1,...,n-1，算法仍采用邻接矩阵G.arcs[n][n]表示有向网络.

弗洛伊德算法的基本操作为:

```
if (D[i][k]+D[k][j] < D[i][j])
    {
        D[i][j] = D[i][k]+D[k][j];
    }
```

其中 k 表示在路径中新增添考虑的顶点号，i 为路径的起始顶点号，j 为路径的终止顶点号。

9.6.2 各顶点之间的最短路径

□ Floyd算法思想

对顶点进行编号，设顶点为0,1,...,n-1，算法仍采用邻接矩阵G.arcs[n][n]表示有向网络.

$D^{(-1)}$[n][n]表示中间不经过任何点的最短路径；

它就是邻接距阵G.arcs[n][n] ；

$D^{(0)}$[n][n]中间只允许经过0的最短路径；

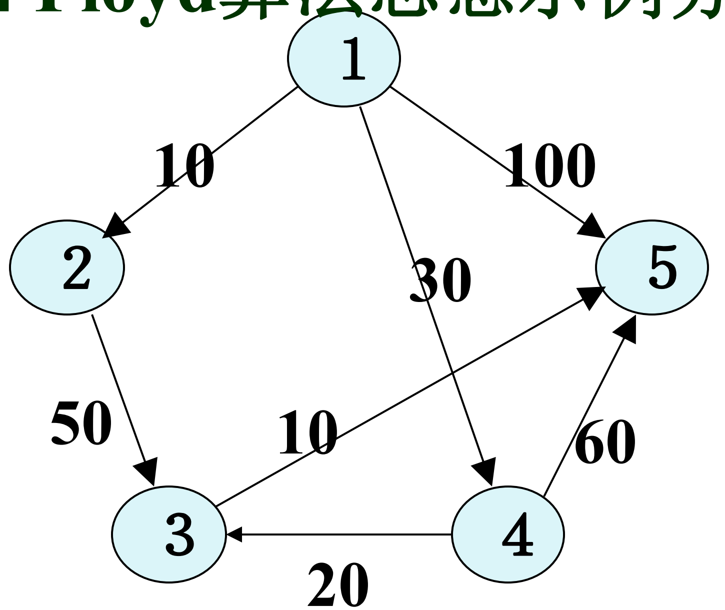
$D^{(1)}$[n][n]中间只允许经过0、1的最短路径；

....

$D^{(n-1)}$[n][n],中间可经过所有顶点的最短路径。

9.6.2 各顶点之间的最短路径

□ Floyd算法思想示例分析



$$A_0 = \begin{bmatrix} 0 & 10 & \infty & 30 & 100 \\ \infty & 0 & 50 & \infty & \infty \\ \infty & \infty & 0 & \infty & 10 \\ \infty & \infty & 20 & 0 & 60 \\ \infty & \infty & \infty & \infty & 0 \end{bmatrix}$$

$$A_4 = \begin{bmatrix} 0 & 10 & 50 & 30 & 60 \\ \infty & 0 & 50 & \infty & 60 \\ \infty & \infty & 0 & \infty & 10 \\ \infty & \infty & 20 & 0 & 30 \\ \infty & \infty & \infty & \infty & 0 \end{bmatrix}$$

$$A_2 = \begin{bmatrix} 0 & 10 & 60 & 30 & 100 \\ \infty & 0 & 50 & \infty & \infty \\ \infty & \infty & 0 & \infty & 10 \\ \infty & \infty & 20 & 0 & 60 \\ \infty & \infty & \infty & \infty & 0 \end{bmatrix}$$

$$A_3 = \begin{bmatrix} 0 & 10 & 60 & 30 & 70 \\ \infty & 0 & 50 & \infty & 60 \\ \infty & \infty & 0 & \infty & 10 \\ \infty & \infty & 20 & 0 & 30 \\ \infty & \infty & \infty & \infty & 0 \end{bmatrix}$$

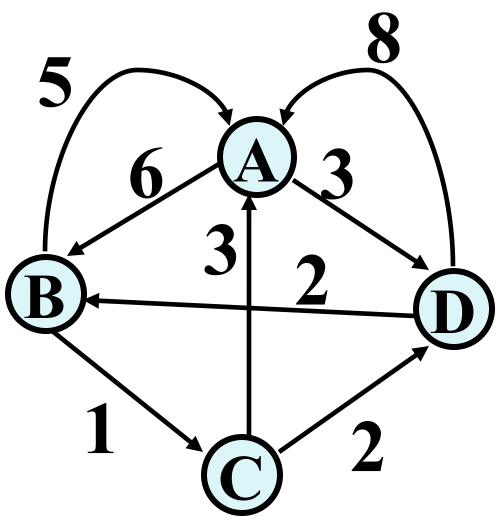
可以经过第二个点后的矩阵

9.6.2 各顶点之间的最短路径

☐ Floyd算法思想示例分析

例

求下图各顶点之间的最短路径



两点之间直达的距阵$D^{(-1)}$：

AA	AB	AC	AD
BA	BB	BC	BD
CA	CB	CC	CD
DA	DB	DC	DD

0	6	∞	3
5	0	1	∞
3	∞	0	2
8	2	∞	0

两点之间只允许经过A点的距阵$D^{(0)}$：

AA	AB	AC	AD
BA	BB	BC	BAD
CA	CAB	CC	CD
DA	DB	DC	DD

0	6	∞	3
5	0	1	8
3	9	0	2
8	2	∞	0

9.6.2 各顶点之间的最短路径

□ Floyd算法

```
Void Floyd(A,C,n)
{
   for (i=1;i<=n;i++)
      for (j=1;j<=n; j++)
            A[i][j]=C [i][j];
            //A初始化，A表示任意两点之间
            //C是有向图G的邻接矩阵
   for (k=1;k<=n;k++)
       for (i=1;i<=n;i++)
          for (j=1;j<=n; j++)
                if(A[i][k]+ A[k][j]< A[i][j]) //经过k点后路径长度是否变短
                     A[i][j]= A[i][k]+ A[k][j];
}
```

时间复杂度 O（n^3）

本章小结

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. 熟练掌握图的深度优先和广度优先搜索遍历及算法。

3. 理解图的遍历算法与树的遍历算法之间的类似和差异；

4.掌握拓扑排序、关键路径和最短路径**的求解**、应用背景和思想；

5.掌握构造最小生成树的过程和的算法。