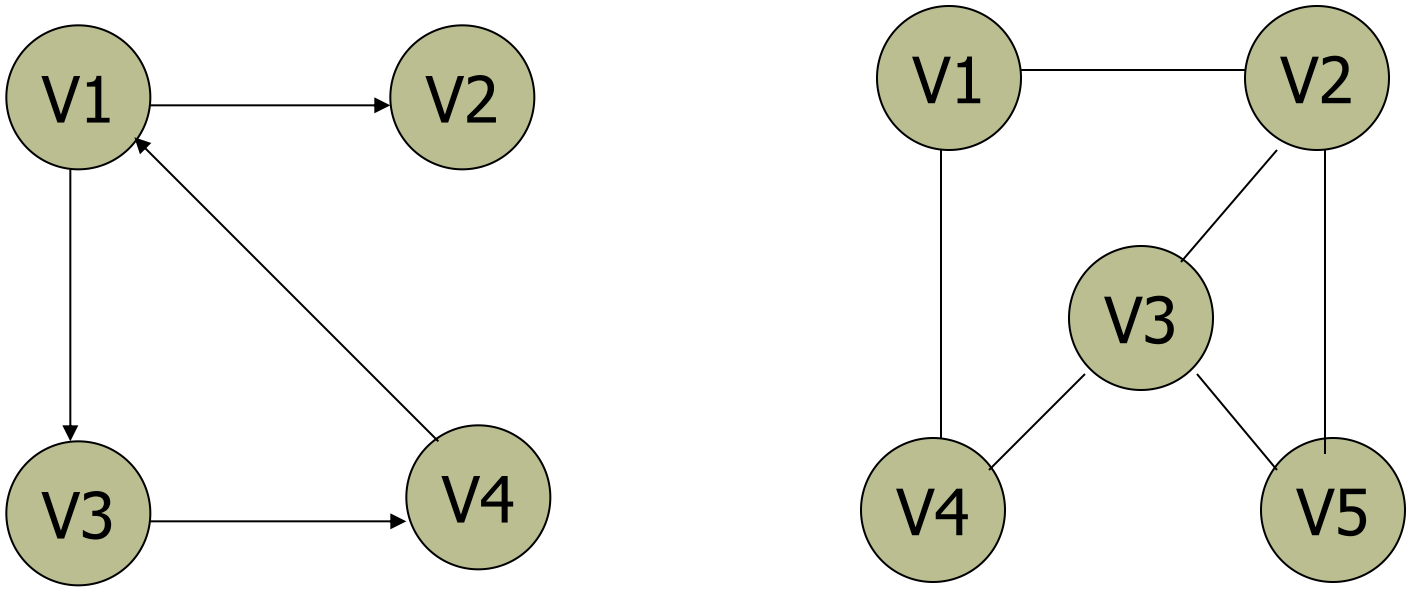
第7章 图

本章目录

主要内容

- **知识点**
  - 1．图的定义，概念、术语及基本操作。
  - 2．图的存储结构，特别是邻接矩阵和邻接表。
  - 3．图的深度优先遍历和宽度优先遍历。
  - 4．图的应用（连通分量，最小生成树，拓扑排序，关键路经，最短路经）。
- **重点难点**
  - 1．基本概念中，完全图、连通分量、生成树和邻接点是重点。
  - 2．建立图的各种存储结构的算法。
  - 3．图的遍历算法及其应用。
  - 4．通过遍历求出深（宽）度优先生成树。
  - 5．最小生成树的生成过程。
  - 6．拓扑排序的求法。关键路经的求法。
  - 7．最短路径的手工模拟。

图的示例（有向图与无向图）



结点之间的关系：多对多，任两个结点之间都可能有关系存在

7.1 图的定义和术语

- 图(Graph)——图G是由两个集合V(G)和E(G)组成的,记为G=(V,E)

  其中：V(G)是顶点的非空有限集

  E(G)是边的有限集合，边是顶点的无序对或有序对

- 有向图——有向图G是由两个集合V(G)和E(G)组成的

  其中：V(G)是顶点的非空有限集

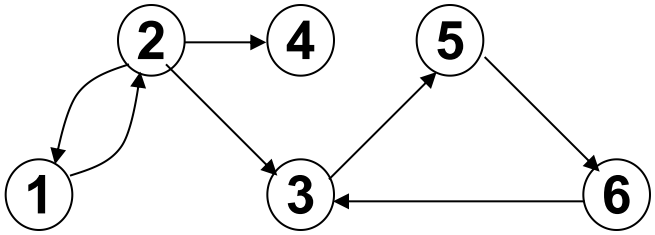
  E(G)是有向边（也称弧）的有限集合，弧是顶点的有序对，记为<v,w>，v,w是顶点，v为弧尾，w为弧头

- 无向图——无向图G是由两个集合V(G)和E(G)组成的

  其中：V(G)是顶点的非空有限集

  E(G)是边的有限集合，边是顶点的无序对，记为（v,w）或（w,v），并且（v,w)=(w,v)

例

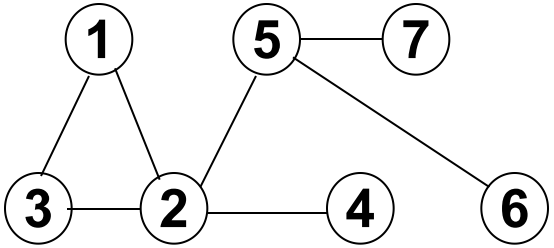


G1

图G1中：V(G1)={1,2,3,4,5,6}

E(G1)={<1,2>, <2,1>, <2,3>, <2,4>, <3,5>, <5,6>, <6,3>}

例



G2

图G2中：V(G2)={1,2,3,4,5,6,7}

E(G1)={(1,2), (1,3), (2,3), (2,4),(2,5), (5,6), (5,7)}

性质：无向图最多有n(n-1)/2 条边

有向图最多有n(n-1)条边

- 有向完全图
  - n个顶点的有向图有n(n-1)条边，则此图称为有向完全图
- 无向完全图
  - n个顶点的无向图有n(n-1)/2条边，则此图称为无向完全图
- 邻接点
  - 若(vi,vj)是一条无向边，则称vi和vj互为~
- 关联
  - vi和vj互为邻接点，称边(vi,vj)关联于顶点vi和vj

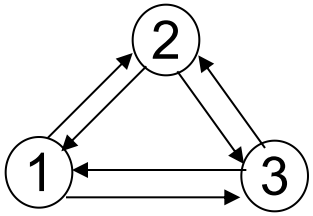
- **顶点的度**：一个顶点***v***的度是与它相关联的边的条数。记作**TD(*v*)**。
- **入度**：顶点 ***v*** 的入度是以 ***v*** 为终点的有向边的条数, 记作 **ID(*v*)**;
- **出度**：顶点 ***v*** 的出度是以 ***v*** 为始点的有向边的条数, 记作 **OD(*v*)**。
- 边数与度的关系：

$$\sum TD(v_i)=2e$$

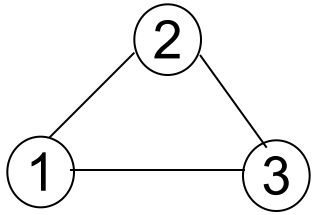
i=1, 2, …, n ， **e**为图的边数。

性质：有向图各顶点入度之和等于出度之和，等于边数

例

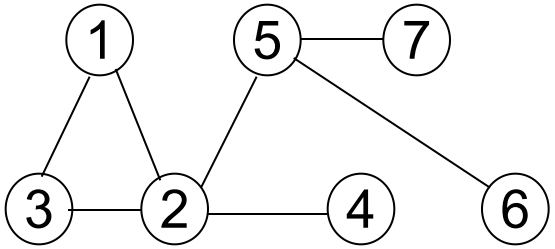


有向完全图



无向完全图

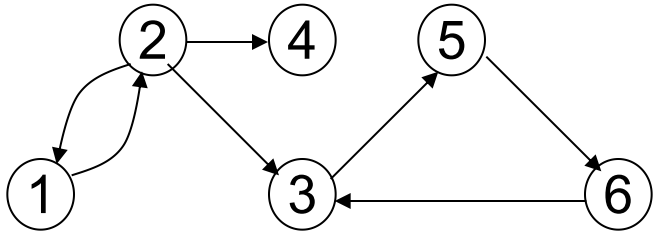
例



G2

顶点5的度：3
顶点2的度：4

例

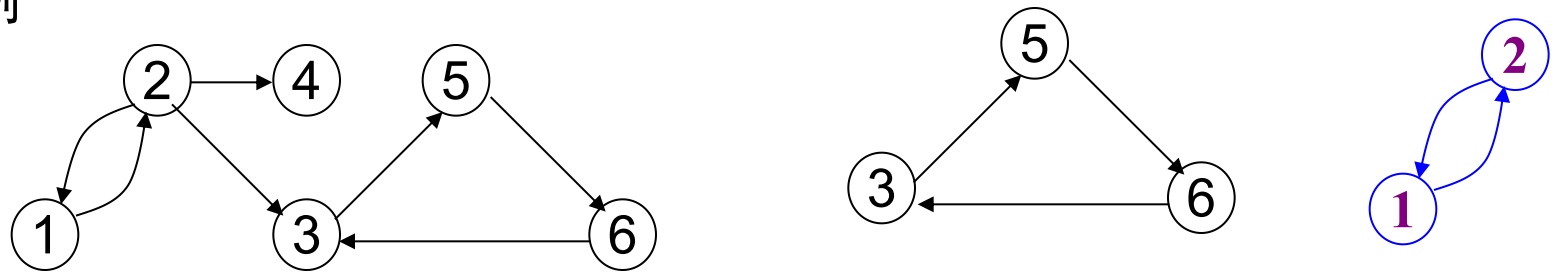


G1

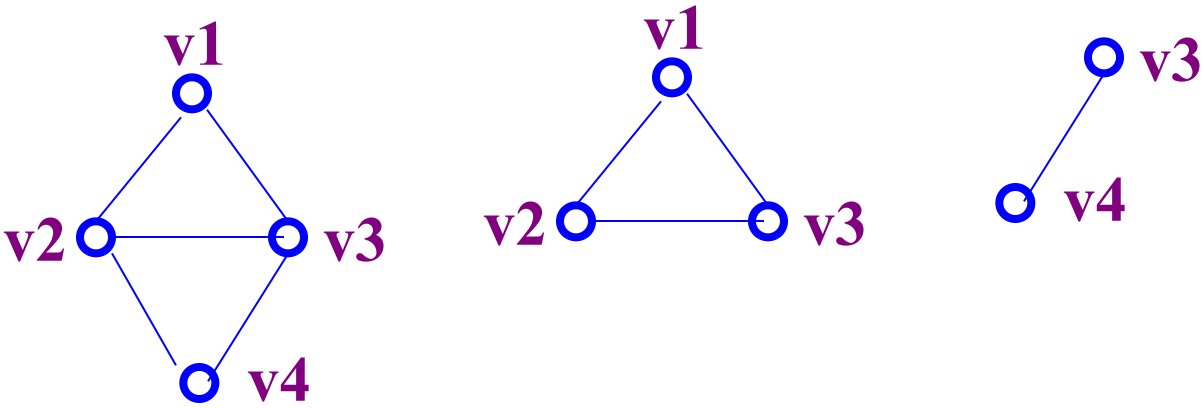
顶点2入度：1　出度：3
顶点4入度：1　出度：0

- 子图——如果图G(V,E)和图G’(V’,E’),满足：V’⊆V，E’⊆E， 则称G’为G的子图

例

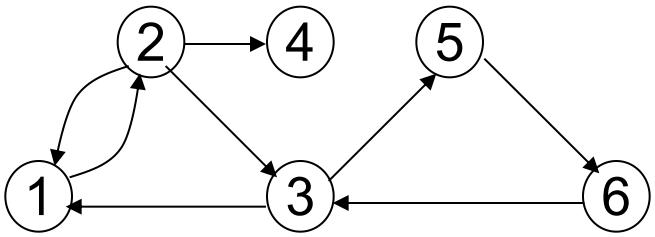


图与子图



- **路径：** 在图 G＝(V, E) 中, 若从顶点 vi 出发, 沿一些边经过一些顶点 vp1, vp2, …, vpm，到达顶点vj。则称顶点序列 (vi vp1 vp2 ... vpm vj) 为从顶点vi 到顶点 vj 的路径。它经过的边(vi, vp1)、(vp1, vp2)、...、(vpm, vj) 应是属于E的边。
- **路径长度：** 非带权图的路径长度是指此路径上边的条数。带权图的路径长度是指路径上各边的权之和。
- **简单路径 ：** 若路径上各顶点 $v_1, v_2, \ldots, v_m$ 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：** 在一个回路中，若除第一个与最后一个顶点外，其余顶点不重复出现的回路称为简单回路(简单环)。

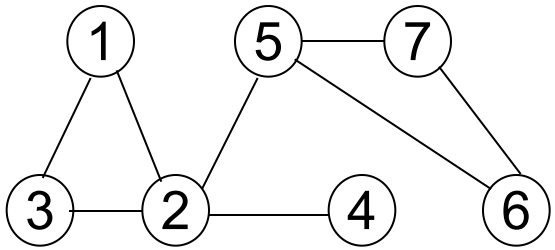
例



G1

路径：1，2，3，5，6，3
路径长度：5
简单路径：1，2，3，5
回路：1，2，3，5，6，3，1
简单回路：3，5，6，3

例



G2

路径：1，2，5，7，6，5，2，3
路径长度：7
简单路径：1，2，5，7，6
回路：1，2，5，7，6，5，2，1
简单回路：1，2，3，1

- **连通图与连通分量** ：在无向图中，若从顶点v1到顶点v2有路径，则称顶点v1与v2是连通的。如果图中**任意一对顶点都是连通的**，则称此图是连通图。
- 非连通图的极大连通子图叫做**连通分量**。
- “**极大**”的含义：指的是对子图再增加图G中的其它顶点，子图就不再连通。
- **考点：**

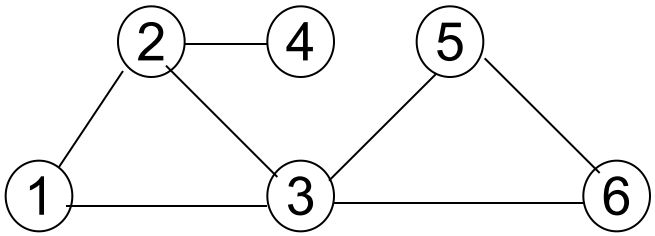
  （1）n个顶点的无向图最少需要多少条边，才能保证连通。答案：n-1条

  （2）要保证n个顶点的无向图G在任何情况下都是连通的，至少需要多少条边？

  答案:(n-1)(n-2)/2+1

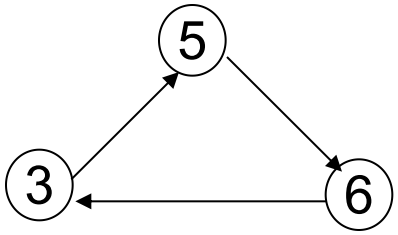
- **强连通图与强连通分量**：在有向图中，若对于每一对顶点vi和vj，都存在一条从vi到vj和从vj到vi的路径，则称此图是**强连通图**。非强连通图的极大强连通子图叫做**强连通分量**。
- **考点：**n个顶点的有向图是强连通图，至少需要多少条边？答案：n条

例



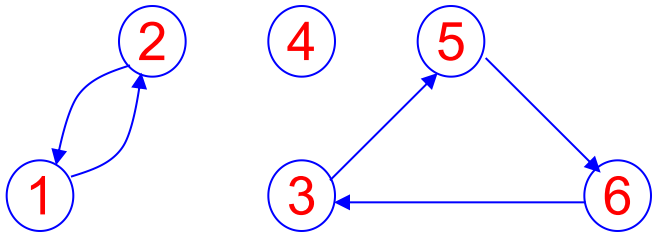
连通图

例



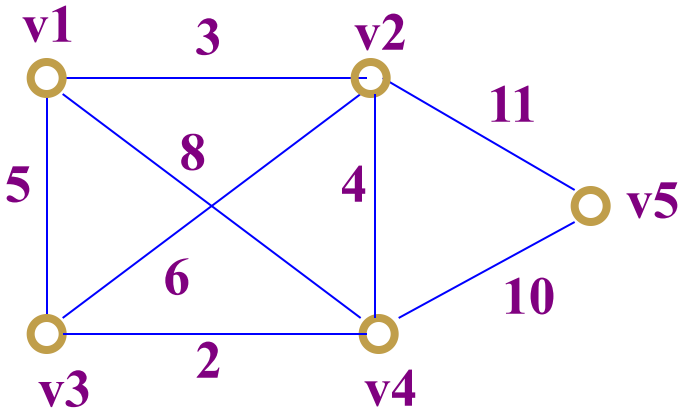
强连通图

例

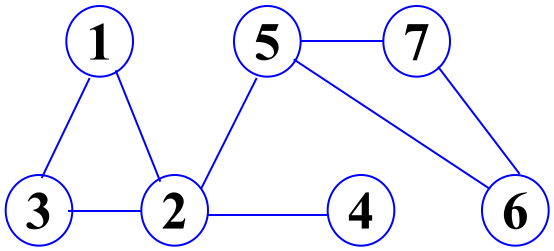


非连通图的
强连通分量

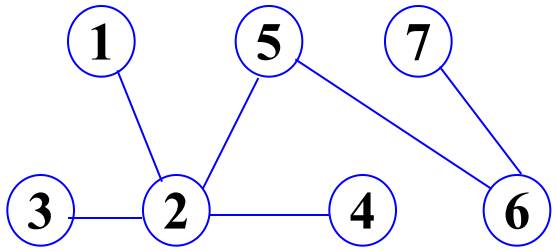
- 权——与图的边或弧相关的数叫权
- 网(络)——带权的图叫网



- 一个连通图的生成树是一个极小连通子图，它含有图中全部顶点，但只有足以构成一棵树的n-1条边。
- 一棵有n个顶点的生成树有且仅有n-1条边。

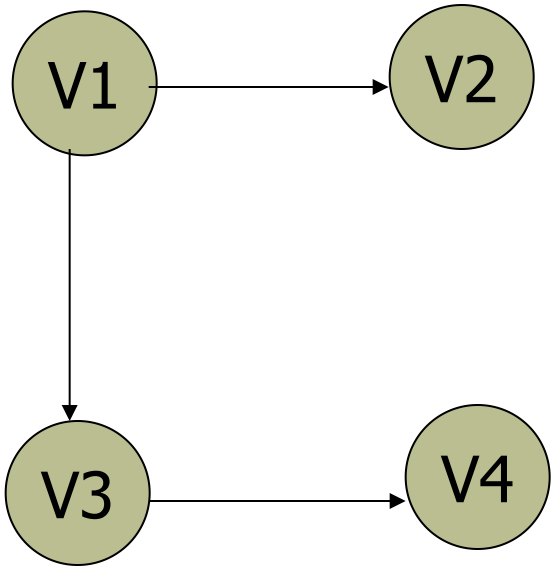


连通图



生成树（自由树）

- 无回路的图称为树或自由树或无根树
- **有向树：只有一个顶点的入度为0，其余顶点的入度为1的有向图。**



有向树是弱连通的

图的基本运算

- 1．创建一个空图
  - Graph createGraph ( )
- 2．判断图G是否空图，是则返回1，否则返回0
  - int isNullGraph (G)
- 3．把图G置为空图
  - void makeNullGraph (G)
- 4．销毁一个图G，即释放图G占用的空间
  - void destroyGraph (G)
- 5．找图中的第一个顶点，如果图是空图则返回NULL
  - Vertex firstVertex (G)

图的基本运算

- 6．找图中顶点vi的下一个顶点
  - Vertex nextVertex (G , vi)
- 7．查找图中值为value的顶点
  - Vertex searchVertex (G , value)
- 8．在图G中增加一个值为value的顶点
  - Graph addVertex (G , value)
- 9．在图G中删除一个顶点和与该顶点相关联的所有边
  - Graph deleteVertex (G , vertex)
- 10．在图G中删除一条边e（ <vi,vj>或者(vi,vj) ）
  - Graph deleteEdge (G , vi , vj)

图的基本运算

- 11．在图G中增加一条边<vi,vj>或者(vi,vj)
  - Graph addEdge (G , vi , vj)
- 12．判断图G中是否存在一条指定边<vi,vj>或者(vi,vj)
  - int findEdge (G , vi , vj)
- 13．找图G中与顶点v相邻的第一个顶点
  - Vertex firstAdjacent (G , v)
  - v与返回顶点构成的边也称为与v相关联的第一条边。
- 14．找图G中与vi相邻的，相对相邻顶点vj的，下一个相邻顶点
  - Vertex nextAdjacent (G , vi , vj)
  - vi与返回值构成的边也称为是vi与vj构成的边的下一条边

7.2 图的存储结构(1)：数组表示（邻接矩阵）

- 数据结构的存储:
  - 既要表示数据元素（值），又要表示数据元素之间的关系；
- 图的任意两个结点（顶点）之间都可能有关系，因此用二维数组来表示；
  - 二维数组中元素a_{ij}= 0表示vi和vj没有关系，即vi和vj不邻接（或没有vi邻接到vj）；a_{ij} = 1表示vi和vj有关系，即vi和vj邻接。
  - 对于网，可以用a_{ij}表示权。若vi和vj不邻接（或没有vi邻接到vj），则可以用无限大∞表示权。

图的存贮结构(1)：数组表示

```
#define MAXNODE  64          // 图中顶点的最大个数
typedef char VertexType;     // 顶点的数据类型
typedef int EdgeType;        // 边或弧的类型
typedef struct
 { VertexType  vexs[MAXNODE];   // 顶点向量
   EdgeType arcs[MAXNODE][MAXNODE];  // 邻接矩阵
   int vexnum,arcnum;      // 图的顶点数和弧数
 }MGraph;
MGraph G;
```

数据类型描述（教材）

```
#define INFINITY  INT_MAX          //最大值∞
#define MAX_VERTEX_NUM  20   //假设的最大顶点数
typedef  enum { DG, DN, AG, AN } GraphKind;
//有向/无向 图，有向/无向网
typedef  struct {    //弧（边）结点的定义
  VRType adj;//顶点间关系
  InfoType *info;  //该弧相关信息的指针
}
AdjMatrix[MAX_VERTEX_NUM ][MAX_VERTEX_NUM];
```

数据类型描述（教材）

```
typedef  struct{    //图的定义
  VertexType vexs [MAX_VERTEX_NUM ] ;  //顶点表
  AdjMatrix arcs;          //邻接矩阵
  int Vernum,  arcnum; //顶点数，弧（边）数
  GraphKind kind;    //图的种类
}MGraph;  //图邻接矩阵类型
```

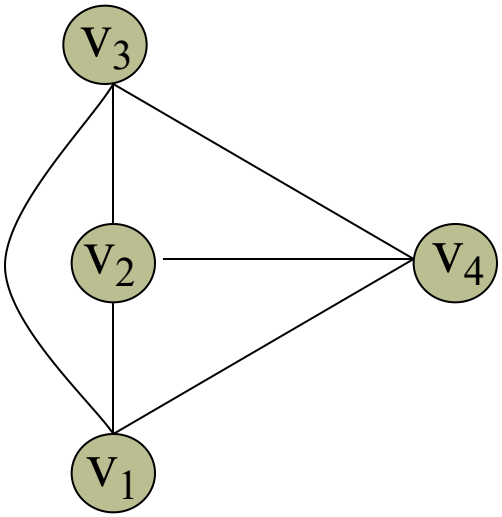
邻接矩阵的构造框架

```
int CreateGraph(MGraph &G){
  scanf(&G.kind);
  switch(G.kind){
     case DG: return CreateDG(G);
     case DN: return CreateDN(G);
     case UDG: return CreateUDG(G);
     case UDN: return CreateUDN(G);
     default: return 0;
     }}
```

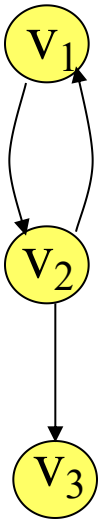
无向网的构造

```
int CreateUDN(MGraph &G){
   scanf(&G.vexnum, &G.arcnum, &IncInfo);
   for( i=0; i<G.vexnum; i++) scanf(&G.vexs[i]);
   for(i=0; i<G.vexnum; i++)
     for(j=0; j<G.vexnum; j++)  G.arcs[i][j]={INFINITY,
    NULL};
   for(k=0; k<G.arcnum; k++) {
          scanf(&v1, &v2, &w);
   i=LocateVex(G, v1);   j=LocateVex(G, v2);
   G.arcs[i][j].adj=w;
   if(IncInfo)  input(*G.arcs[i][j].info);
   G.arcs[j][i]=G.arcs[i][j];  }  }
```

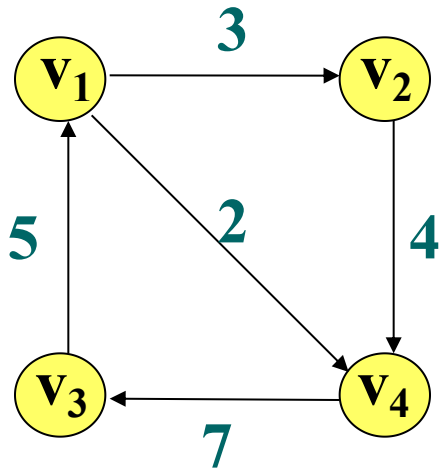
图的存储结构：邻接矩阵表示



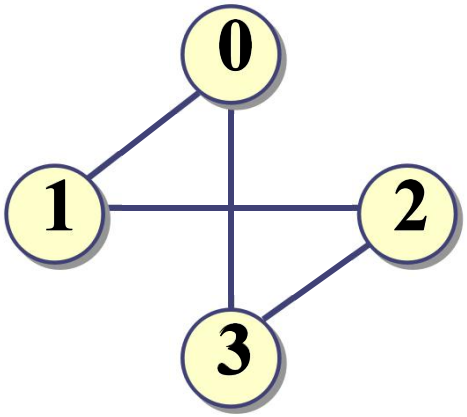
$$\begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix}$$



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



$$\begin{bmatrix} \infty & 3 & \infty & 2 \\ \infty & \infty & \infty & 4 \\ 5 & \infty & \infty & \infty \\ \infty & \infty & 7 & \infty \end{bmatrix}$$



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

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

无向图各顶点的度为：2，2，2，2（会通过邻接矩阵求）

有向图各顶点的度为：2，3，1

图的邻接矩阵表示法的特点

- 对于无向图：
  - 邻接矩阵一定是一个对称矩阵，可压缩存储；有n个顶点的无向图需存储空间为n(n+1)/2
  - 行（列）非零元素个数，表示度
- 对于有向图：
  - 矩阵不一定是一个对称矩阵，有n个顶点的有向图需存储空间为n^2
  - 行非零元素个数，表示出度
  - 列非零元素个数，表示入度
- 邻接矩阵的存储空间只和顶点个数有关，和边数无关

邻接矩阵存储特点

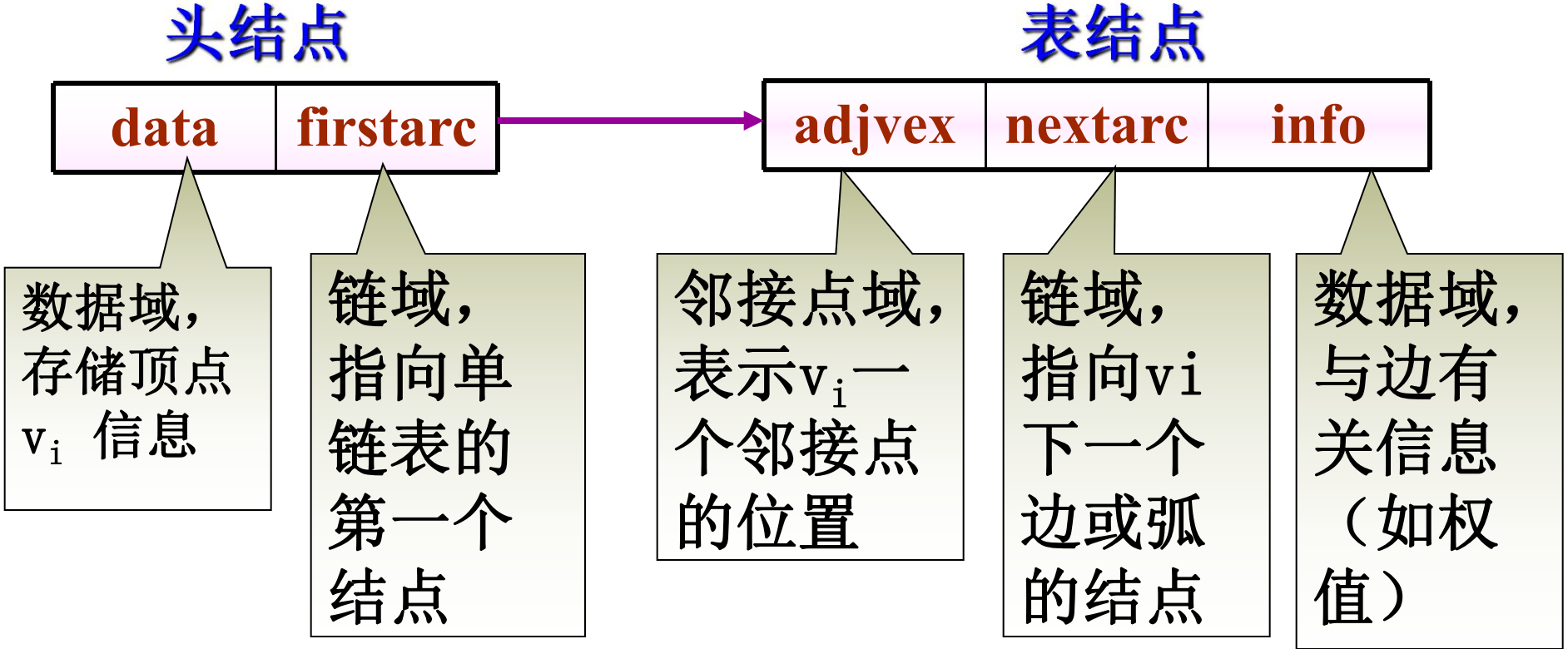
- 容易实现图的操作，如：求某顶点的度、判断顶点之间是否有边（弧）、找顶点的邻接点等等。
- n个顶点需要n*n个单元存储边(弧)；空间效率为$O(n^2)$。 对稀疏图而言尤其浪费空间。

图的存储结构(2)：邻接表

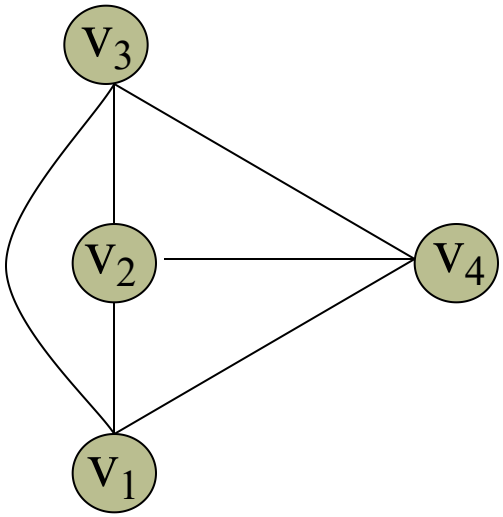
- 对图中每个顶点建立一个邻接关系的单链表，并将其表头指针用向量(一维数组)存储，该结构称为邻接表。
- 无向图的第i个链表将图中与顶点v_i相邻接的所有顶点链接起来，也就是链表中的表头结点到链表中的每个结点表示了与表头结点v_i相关的边。
- 有向图的第i个链表，链接了以顶点v_i为弧尾（射出）的所有顶点，也就是链表中的表头结点到链表中的每个结点表示了图中关联到表头结点v_i的所有弧。

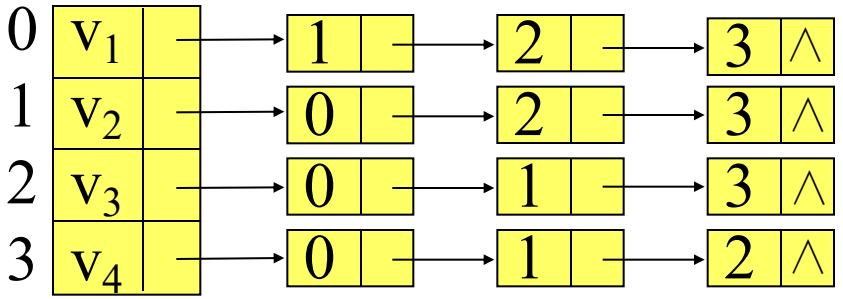
邻接表

- 图的链式存储结构
  - 对每个顶点v_i 建立一个单链表，把v_i关联的边的信息链接起来，表中每个结点设3个域
  - 每个单链表附设一个头结点（设2个域），存v_i信息，头结点另外用顺序存储结构存储



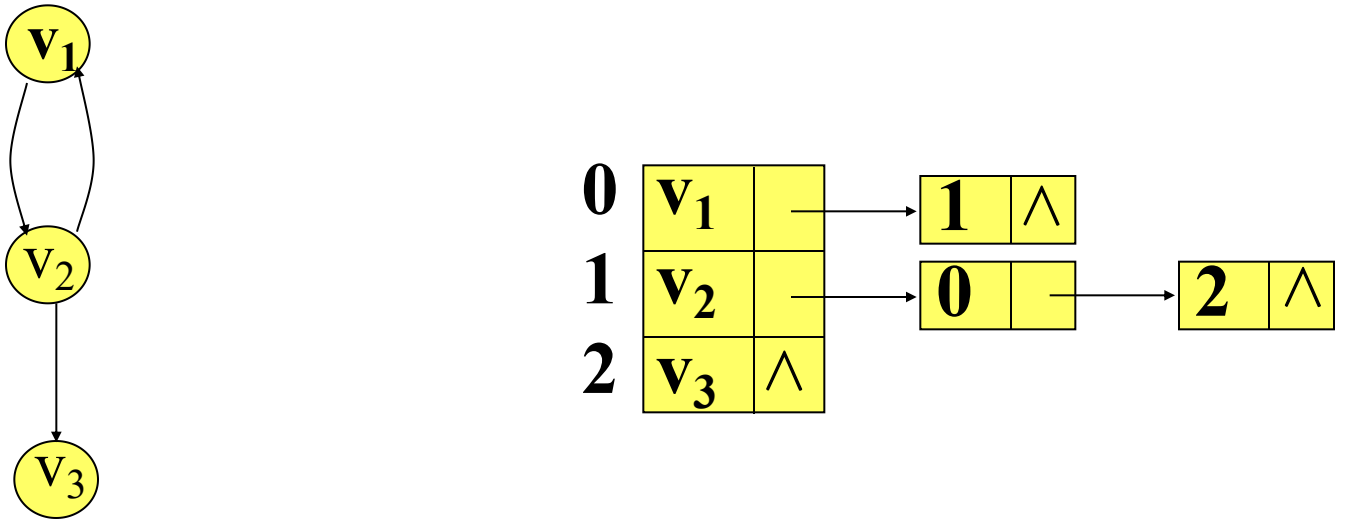
邻接表示例（无向图）





对于无向图，第i个顶点的度为第i个链表的结点数

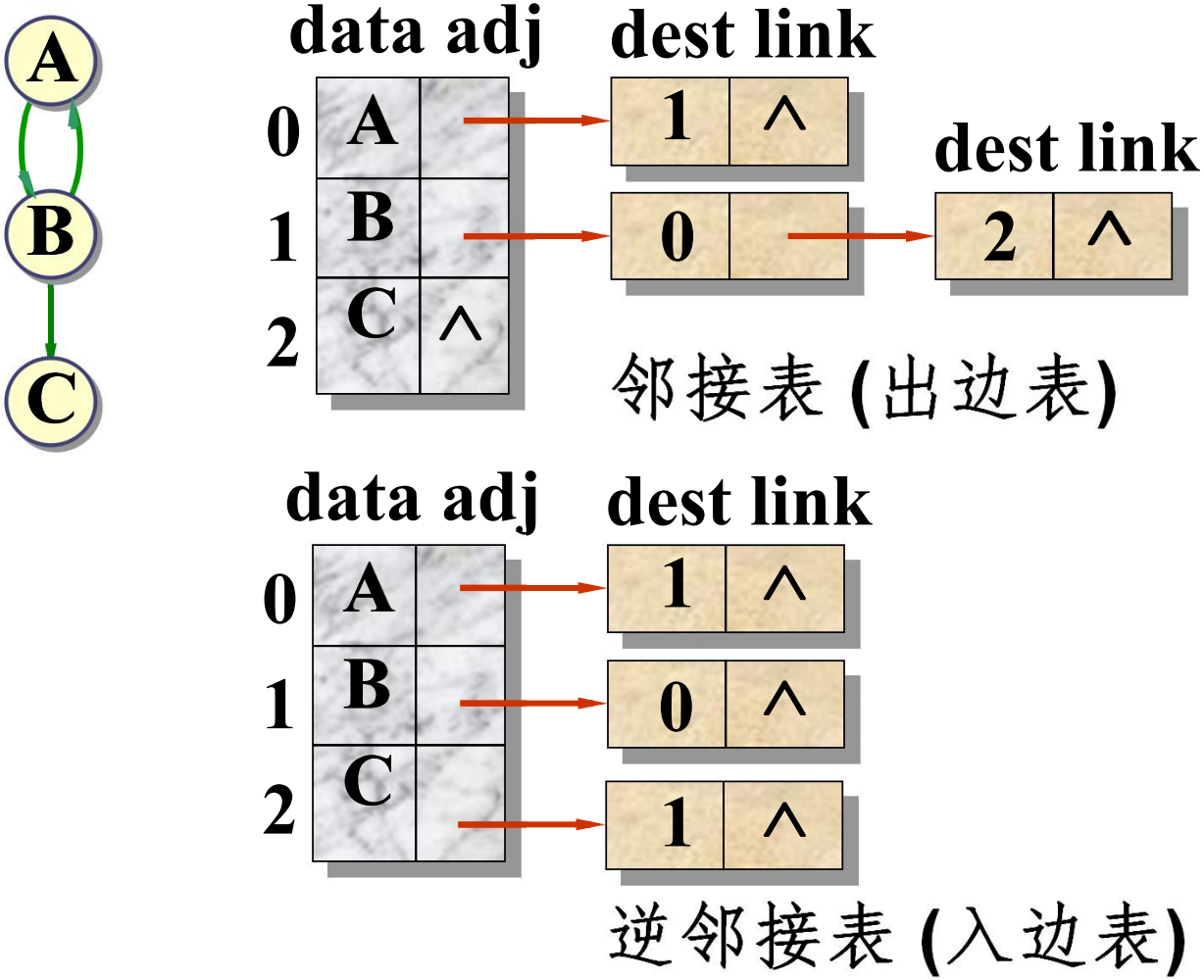
邻接表示例（有向图）



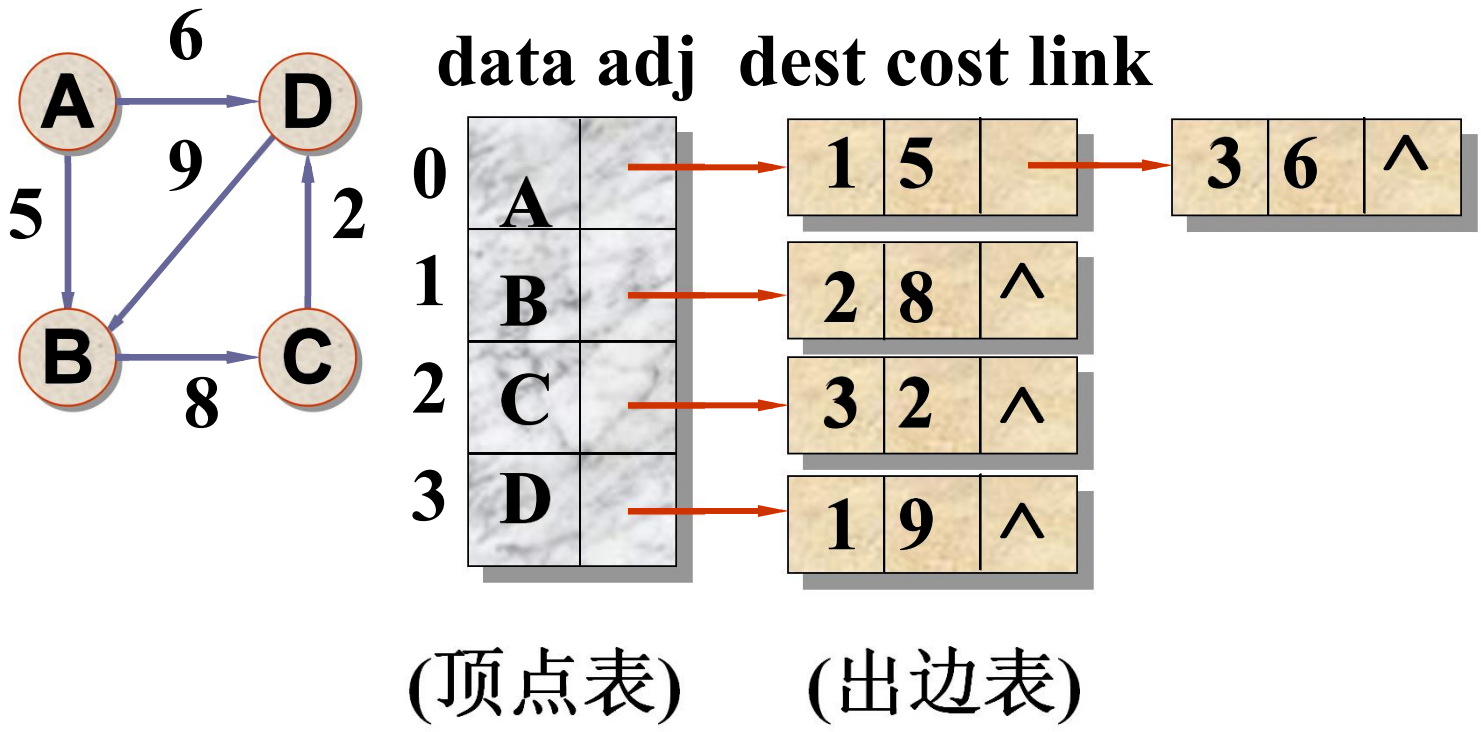
对于有向图，第i个顶点的出度为第i个链表的结点数

第i个顶点的入度为整个单链表中邻接点域值是i的结点个数

有向图的邻接表和逆邻接表



网络 (带权图) 的邻接表


6
A
D
9
5
2
B
C
8
data adj dest cost link
0
A
1
B
2
C
3
D
1 5
3 6 ^
2 8 ^
3 2 ^
1 9 ^
(顶点表)
(出边表)

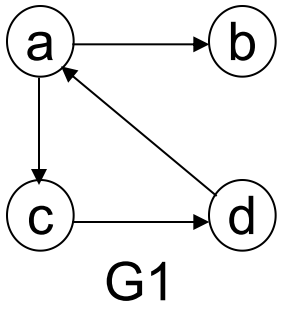
数据类型描述

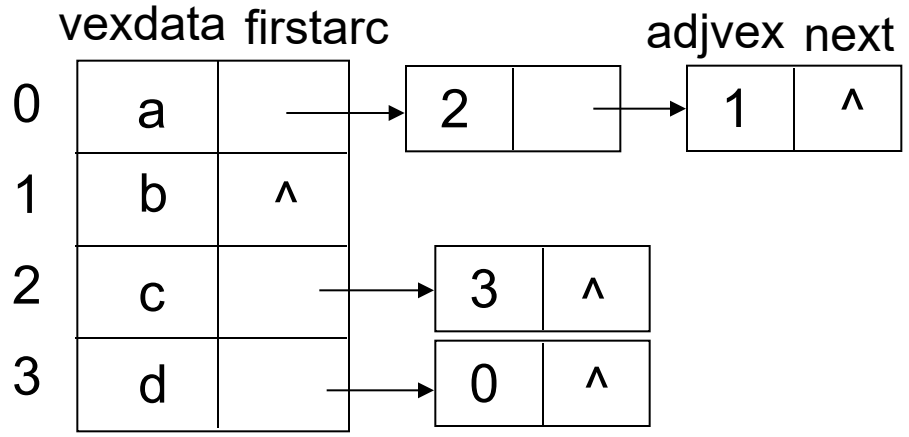
```
#define MAX_VERTEX_NUM  20

typedef struct ArcNode{ //边（弧）结点的类型定义
        int  adjvex;   //边（弧）的另一顶点的在数组中的位置
        ArcNode *nextarc;//指向下一边（弧）结点的指针
       InfoType *info;     //该弧相关信息的指针
}ArcNode;

typedef struct Vnode{//顶点结点及其数组的类型定义
VertexType data; //顶点信息
ArcNode * firstarc;        //指向关联该顶点的边（弧）链表
} Vnode, AdjList[MAX_VERTEX_NUM];
```

数据类型描述

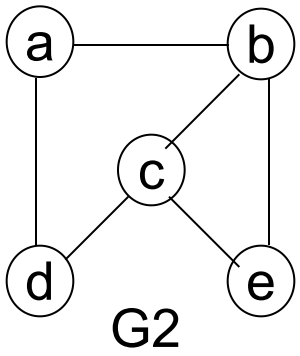
```
typedef struct {
        AdjList  vertices;
        int  vexnum, arcnum;    //图的当前顶点数和弧数
        int  kind;     //图的种类标志
} ALGraph;          //图邻接表类型
```

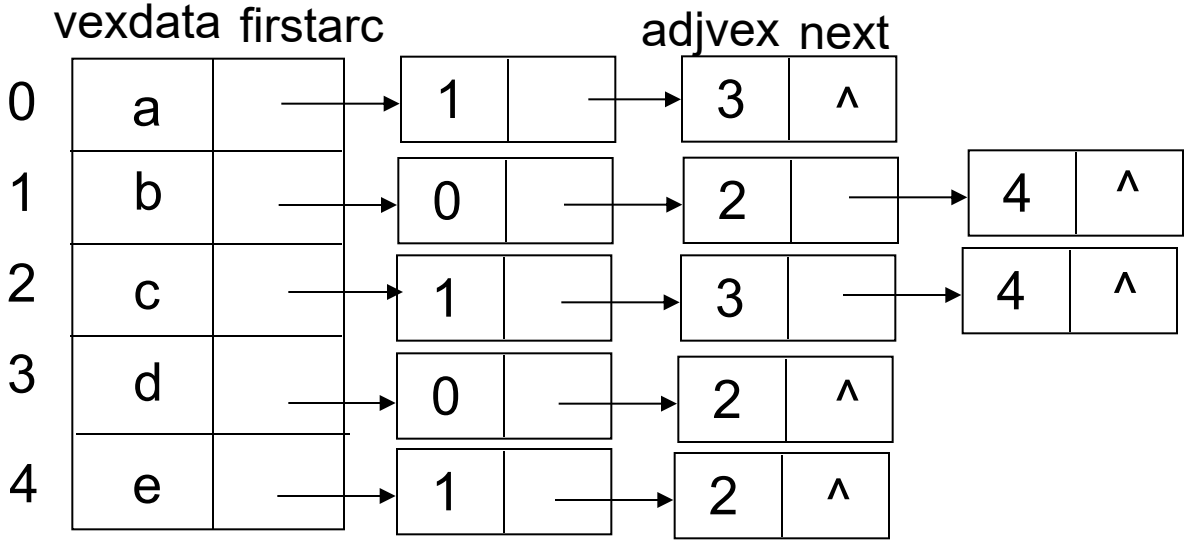
例


a
b
c
d
G1


vexdata
firstarc
adjvex
next
0
a
2
1
^
1
b
^
2
c
3
^
3
d
0
^

例


a
b
c
d
e
G2


vexdata
firstarc
adjvex
next
0
a
1
3
^
1
b
0
2
4
^
2
c
1
3
4
^
3
d
0
2
^
4
e
1
2
^

有向带权图的创建1

```
void CreatALGraph(ALGraph &G)
{
    int i,j,k,weight;
    ArcNode *s;
    printf("input vexnum and arcnum:"); //输入顶点数和边数
    scanf("%d%d",&G.vexnum, &G.arcnum);
    printf("input vertex information:");  //输入顶点信息
    for(i=0;i<G.vexnum;i++)
    {scanf("%d",&(G.vertices[i].data));
        G.vertices[i].firstarc=NULL;    }
```

有向带权图的创建2

```
    printf("input edges, input the xuhao\n");
    for(k=0;k<G.arcnum;k++)
    {   scanf("%d%d%d",&i,&j,&weight);
        s=(ArcNode*)malloc(sizeof(ArcNode));
        s->adjvex=j;
        s->weight=weight;        //对应类型定义里
  的info
        s->nextarc=G.vertices[i].firstarc;
        G.vertices[i].firstarc=s;
    }
}
```

思考题：无向带权图的创建

邻接表的输出

```
void OutputALGraph(ALGraph G)
{   int i;
   for(i=0;i<G.vexnum;i++)
   {    ArcNode * s;
        printf("%d:%d",i,G.vertices[i].data);
  //顶点信息
        s=G.vertices[i].firstarc;
        while(s)
        {  printf("\t%d,%d",s->adjvex,s-
  >weight);
             s=s->nextarc;   }
        printf("\n");   }}
```

图的顶点定位

图的顶点定位实际上是确定一个顶点在AdjList数组中的下标。

算法实现：

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

图中增加顶点：在AdjList数组的末尾增加一个数据元素。

```
int  AddVertex(ALGraph &G , VertexType vp)
   {  if  (G.vexnum>=MAX_VEX)
         {  printf("Vertex Overflow !\n") ;  return(-1) ; }
      if  (LocateVex(G , vp)!=-1)
      {  printf("Vertex has existed !\n") ; return(-1) ; }
      G.vertices[G.vexnum].data=vp ;
      G.vertices[G.vexnum].firstarc=NULL ;
      k=++G.vexnum ;
      return(k) ;
   }
```

图中增加一条弧

根据给定的弧或边所依附的顶点，修改单链表：无向图修改两个单链表；有向图修改一个单链表。

算法实现：

```
int  AddArc(ALGraph  &G , VertexType  v1, v2)
   {  int  k , j ;
      ArcNode *p ,*q ;
      k=LocateVex(G , v1)  ;
      j=LocateVex(G , v2)  ;
      if (k==-1||j==-1)
         {  printf("Arc's Vertex do not existed !\n") ;
            return(-1) ; }
```

```
p=(ArcNode *)malloc(sizeof(ArcNode)) ;
p->adjvex=k ; p->info=arc->info ;
p->nextarc=NULL ;   /*  边的起始表结点赋值   */
q=(ArcNode *)malloc(sizeof(ArcNode)) ;
q->adjvex=j ; q->info=arc->info ;
q->nextarc=NULL ;   /*  边的末尾表结点赋值   */
```

```
    if (G.kind==UG)  /*无向图情况 */
       {  q->nextarc=G.vertices[k].firstarc ;
          G.vertices[k].firstarc=q ;
          p->nextarc=G.vertices[j].firstarc ;
          G.vertices[j].firstarc=p ;
       }
    else       /*  建立有向图的邻接链表, 用头插入法  */
       {  q->nextarc=G.vertices[k].firstarc ;
          G.vertices[k].firstarc=q ;  /*  建立正邻接链表用 */
       }
    return(1);
}
```

有向图的十字链表表示法

弧结点：

```
typedef  struct  ArcBox
{   int   tailvex,  headvex;  //弧尾、弧头在表头数组中位置
    struct   ArcBox   *hlink；//指向弧头相同的下一条弧
    struct   ArcBox   *tlink;    //指向弧尾相同的下一条弧
    InfoType *info
}ArcBox;
```

顶点结点：

```
typedef  struct  VexNode
{   int   data;  //存与顶点有关信息
   ArcBox  *firstin；//指向以该顶点为弧头的第一个弧结点
   ArcBox  *firstout; //指向以该顶点为弧尾的第一个弧结点
}VexNode;
```

- 有向图的十字链表表示法

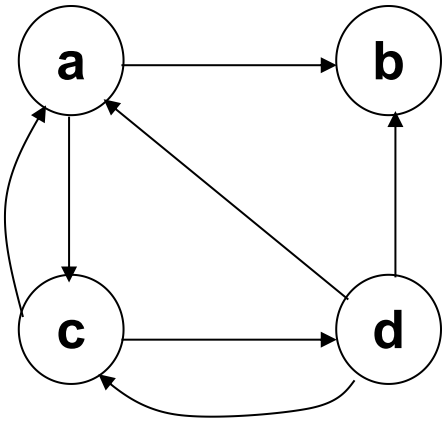
- typedef struct {

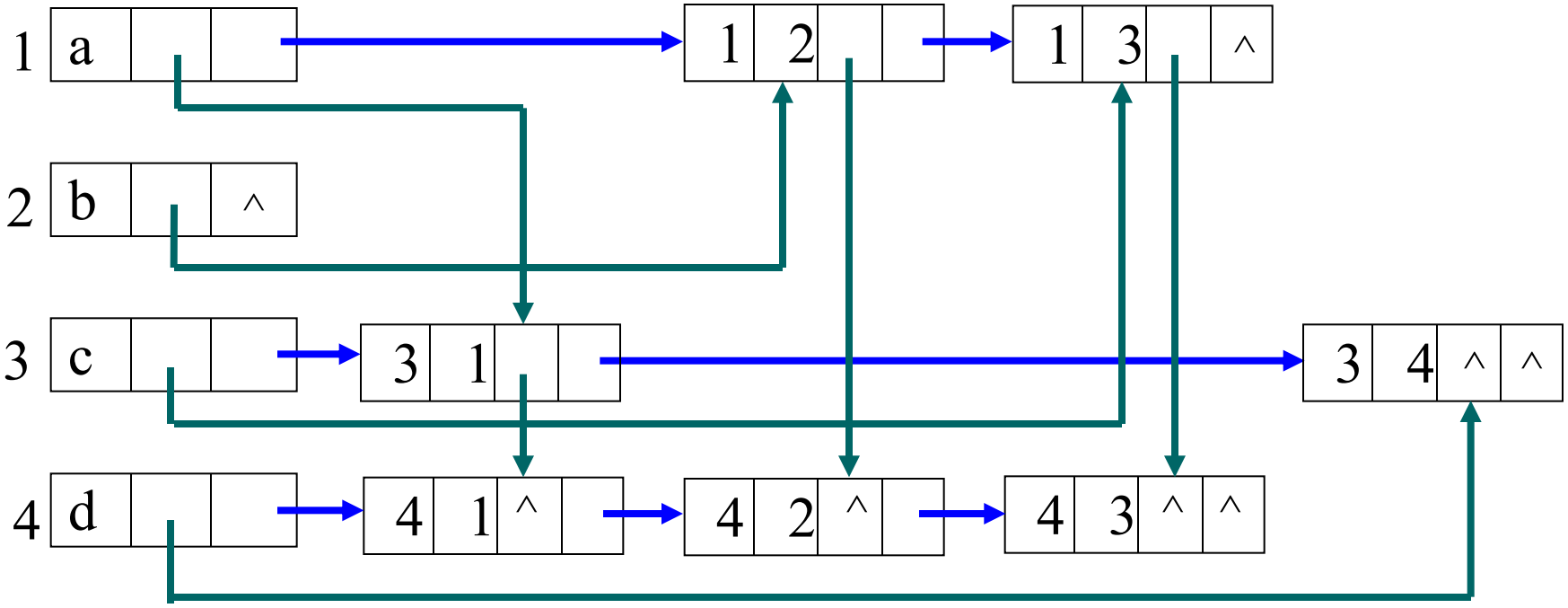
VexNode xlist[MAX_VERTEX_NUM]；//表头向量

Int vexnum, arcnum；//有向图的当前顶点和弧数

}OLGraph；

例





有向图的构造

```
void CreateGraphic(OLGraph &G)
{  int i,j,k;
   ArcBox *arcBox;
   printf_s("请输入顶点数和弧数： ");
   scanf("%d,%d",&G.vexNum, &G.arcNum);
    printf("建立顶点表\n");
   for (i = 0; i < G.vexNum; i++)
    { scanf(“%c”, G.xList[i].data);   //输入顶点值
      G.xList [i].firstin = NULL;      //初始化指针
      G.xList[i].firstOut = NULL;    //初始化指针}
```

```
printf("建立弧表\n");
for (k = 0; k < G.arcNum; k++)
  { printf("请输入（headVex-tailVex）的顶点对序号");
    scanf("%d,%d", &i, &j);
    arcBox = (ArcBox *)malloc(sizeof(ArcBox));  //对弧结点赋值
    arcBox->headVex = j;
    arcBox->tailVex = i;
    arcBox->hlink = G.xList[j].firstin;
    arcBox->tlink = G.xList[i].firstOut;
    arcBox->info = NULL;
   //完成在入弧和出弧链表表头的插入
   G.xList[j].firstin = arcBox;
   G.xList[i].firstOut = arcBox;
  }}
```

无向图的邻接多重表表示法

边结点：

```
typedef   struct  EBox
{   int  mark;   //标志域
    int  ivex, jvex;  //该边依附的两个顶点在表头数组中位置
    struct  EBox   *ilink,  *jlink;  //分别指向依附于ivex和jvex的下一条边
     InfoType * info;
}EBox;
```

mark	ivex	ilink	jvex	jlink	info

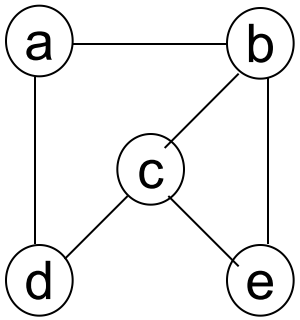
无向图的邻接多重表表示法

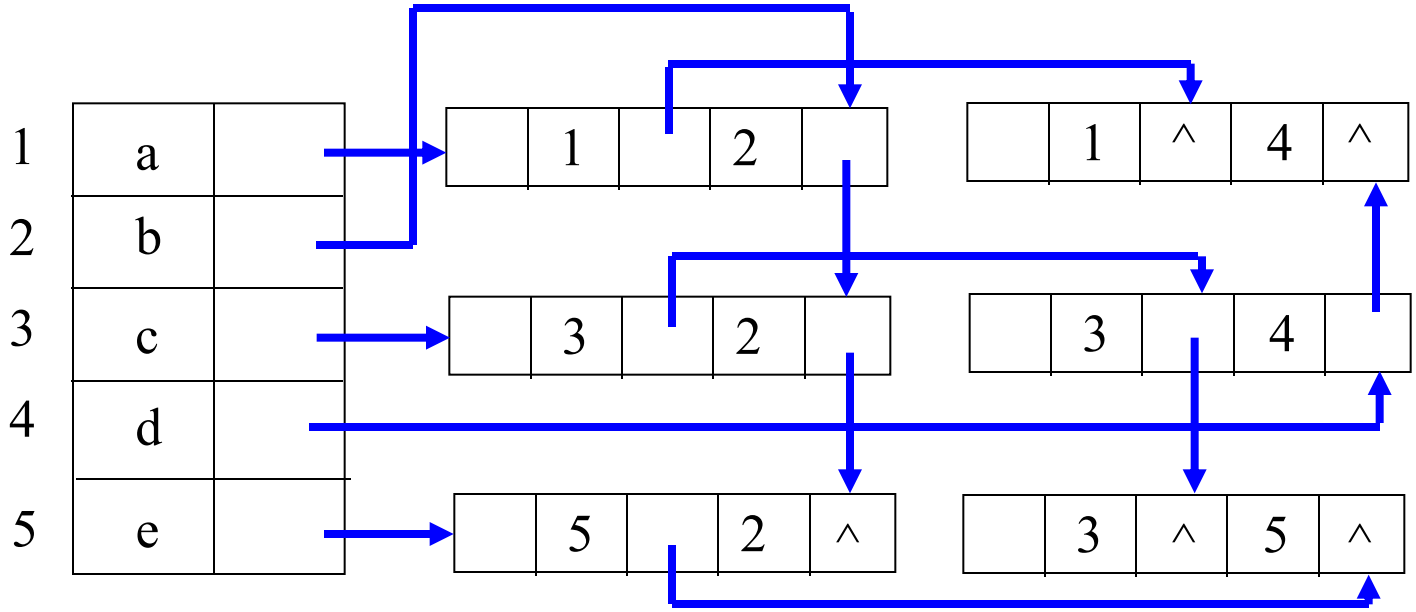
顶点结点：

```
typedef   struct  dnode
{    int   data;   //存与顶点有关的信息
     EBox *firstedge;  //指向第一条依附于该顶点的边
}VexBox;
```


```
Typedef struct{
   VexBox adjmulist[MAX_VERTEX_NUM];
    int vexnum, edgenum;
}AMLGraph;
```

例





7.3图的遍历

- 图的遍历是从图中某个顶点出发，按照某种方式系统地访问图中的所有顶点，使每个顶点仅被访问一次。
- 图的遍历通常有两种方法：深度优先搜索和广度优先搜索。它们对有向图和无向图都适用。

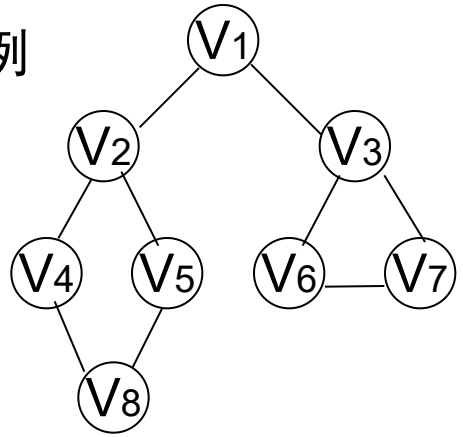
- 深度优先遍历(DFS)

  1 从任一个顶点v出发，访问该顶点；

  2 然后依次从v的未被访问的邻接点出发深度优先遍历图，直至图中所有和v有路径相通的顶点都被访问到；

  3 此时若图中尚有顶点未被访问，则另选中下一个未被访问的顶点作起始点，访问该顶点，继续过程2，直到所有顶点都被访问完。

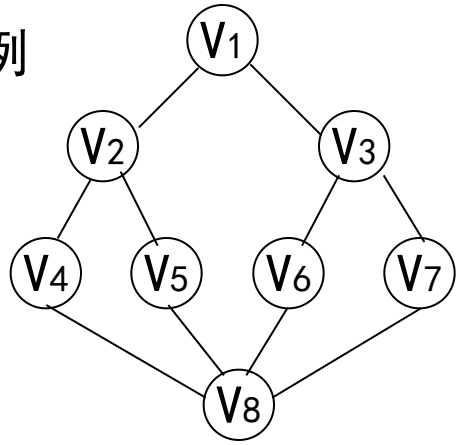
例



深度遍历：V1⇒ V2 ⇒V4 ⇒ V8 ⇒V5 ⇒V3 ⇒V6 ⇒V7

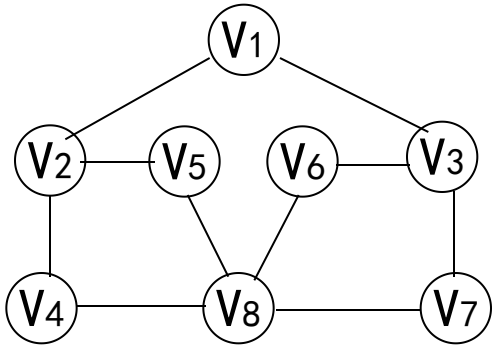
- 注意：
  - 连通图或强连通图：则从图中任意一顶点出发都可以访问图中所有顶点。
  - 由于图中每个顶点都可能与图中其它多个顶点邻接并存在回路，为了避免重复访问已访问过的顶点，在图的遍历中，通常对已访问的顶点作标记。
  - **若不确定存储结构，遍历结果不唯一。**

例



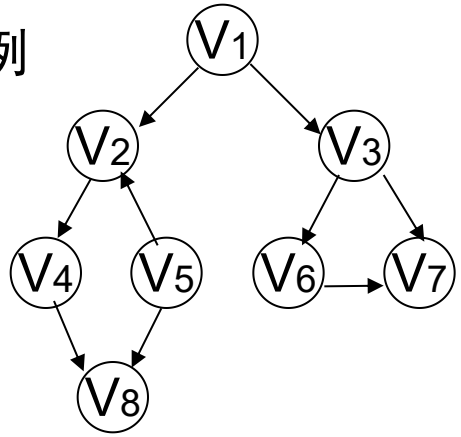
深度遍历：V1⇒ V2 ⇒V4 ⇒ V8 ⇒V5 ⇒V6 ⇒V3 ⇒V7

例



深度遍历：V1⇒ V2 ⇒V4 ⇒ V8 ⇒V5 ⇒V6 ⇒V3 ⇒V7

例



深度遍历：V1⇒ V2 ⇒V4 ⇒ V8 ⇒V3 ⇒V6 ⇒V7 ⇒V5

2 算法实现

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。

```
typedef  emnu {FALSE , TRUE} BOOLEAN ;
BOOLEAN  Visited[MAX_VEX] ;
```

```
void  DFS(ALGraph &G , int v)
   {  ArcNode  *p ;
      Visited[v]=TRUE ;
      Printf(G.vertices[v].data);      /*  置访问标志，访问顶点v  */
      p=G.vertices[v].firstarc;   /*  链表的第一个结点  */
      while (p!=NULL)
         {  if  (!Visited[p->adjvex]) DFS(G, p->adjvex) ;
                /*  从v的未访问过的邻接顶点出发深度优先搜索   */
            p=p->nextarc ;
         }
   }
```

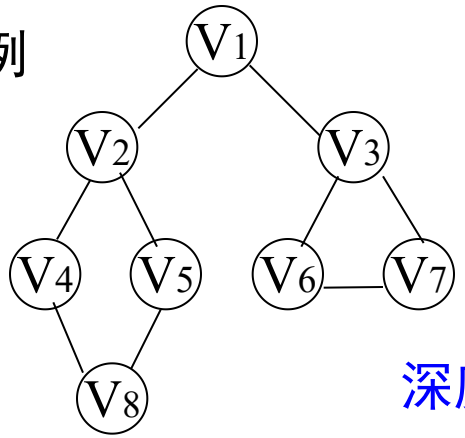
```
void DFS_traverse_Grapg(ALGraph &G)
  {  int v ;
     for (v=0 ; v<G.vexnum ; v++)
        Visited[v]=FALSE ;    /*  访问标志初始化  */
     for (v=0 ; v<G->vexnum ; v++)
        if (!Visited[v])   DFS(G , v);
  }
```

3 算法分析

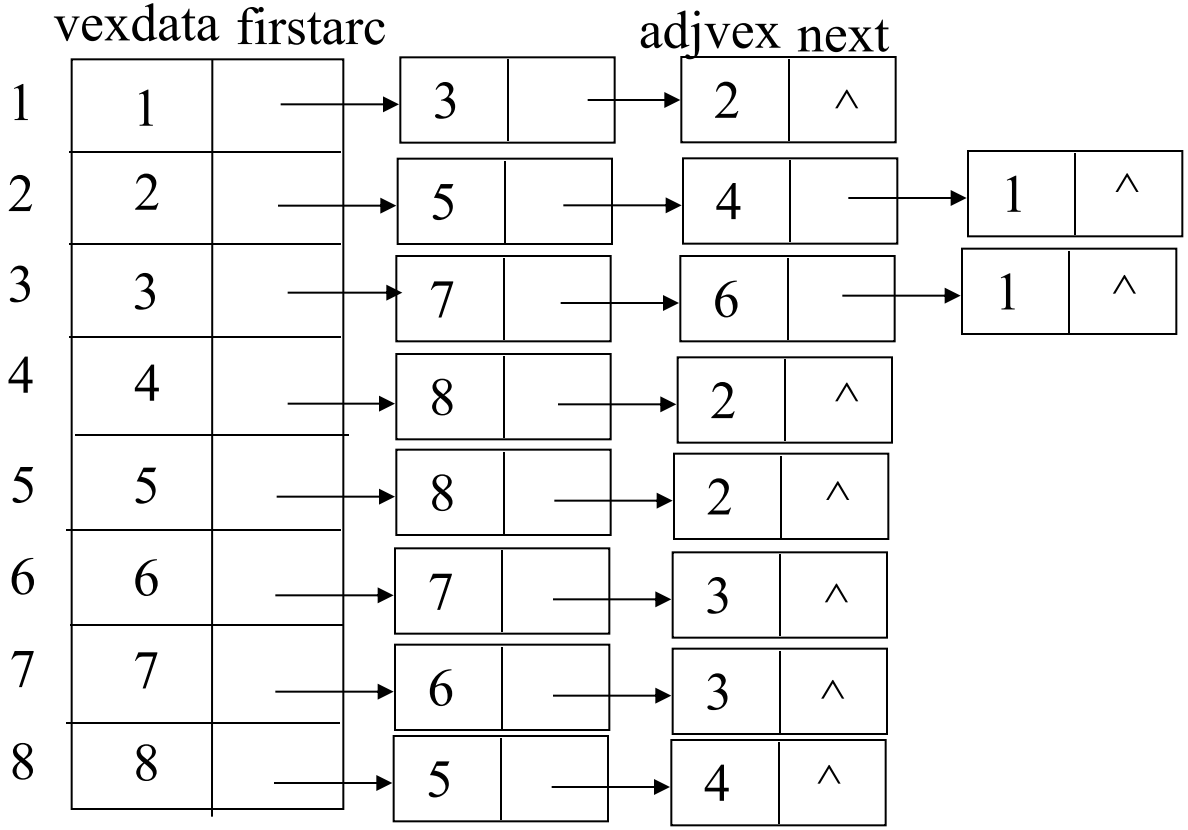
邻接表存储结构：遍历时，对图的每个顶点至多调用一次DFS函数。其实质就是对每个顶点查找邻接顶点的过程，取决于存储结构。当图有e条边，其时间复杂度为O(e)，总时间复杂度为O(n+e) 。

邻接矩阵：时间复杂度为O(n*n)

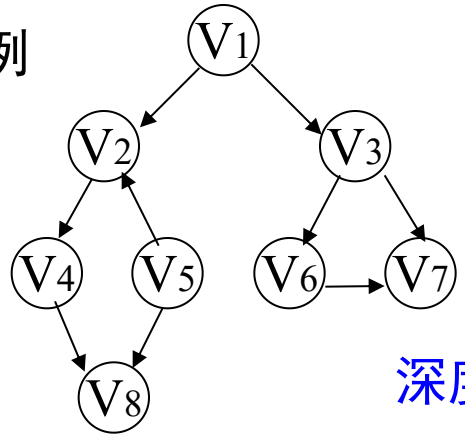
例



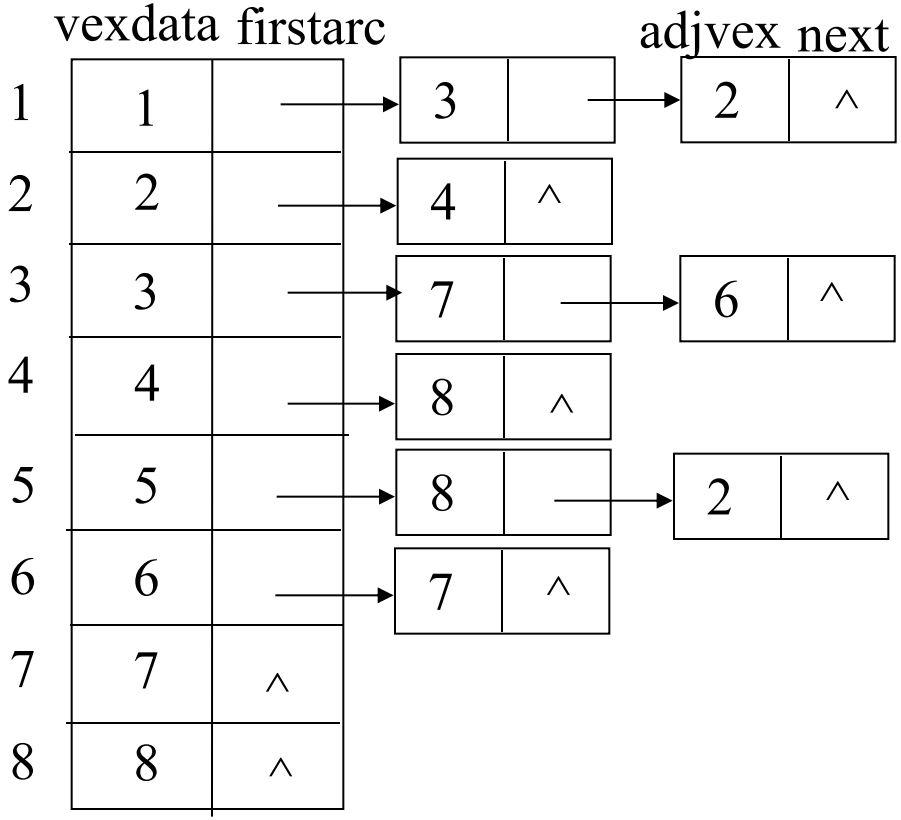
深度遍历：V1⇒V3 ⇒V7 ⇒V6 ⇒V2 ⇒V5 ⇒V8 ⇒V4



例



深度遍历：V1⇒V3 ⇒V7 ⇒V6 ⇒V2 ⇒V4 ⇒V8 ⇒V5



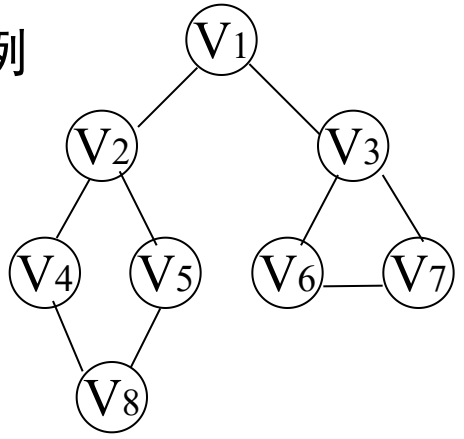
- 广度优先遍历(BFS)

  1 从图中某顶点v出发，在访问v之后，

  2 依次访问v的各个未曾被访问过的邻接点，然后分别从这些邻接点出发依次访问它们的未被访问的邻接点，并使“先被访问的顶点的邻接点”先于“后被访问的顶点的邻接点”被访问，直至图中所有已经被访问的顶点的邻接点都被访问到；

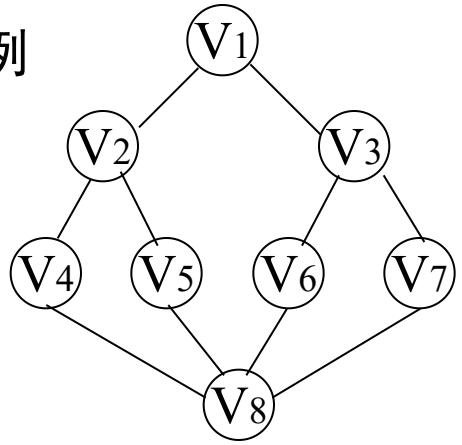
  3 若图中尚有顶点未曾被访问，则另选图中一个未曾被访问的顶点作起点，访问该顶点，继续过程2，直至所有顶点被访问完毕。

例



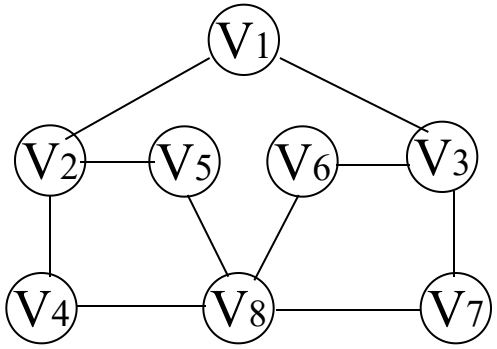
广度遍历：V1⇒ V2 ⇒V3 ⇒ V4 ⇒V5 ⇒V6 ⇒V7 ⇒V8

例



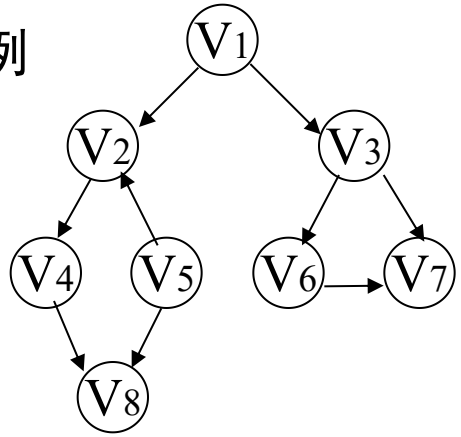
广度遍历：V1⇒ V2 ⇒V3 ⇒ V4 ⇒V5 ⇒V6 ⇒V7 ⇒V8

例



广度遍历：V1⇒ V2 ⇒V3 ⇒ V4 ⇒V5 ⇒V6 ⇒V7 ⇒V8

例



广度遍历：V1⇒ V2 ⇒V3 ⇒ V4 ⇒V6 ⇒V7 ⇒V8 ⇒V5

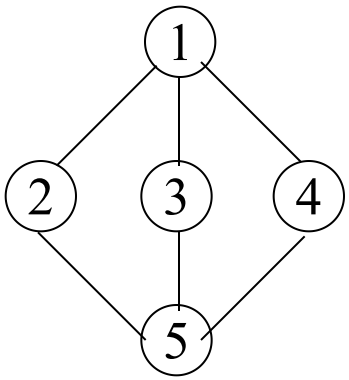
2 算法实现

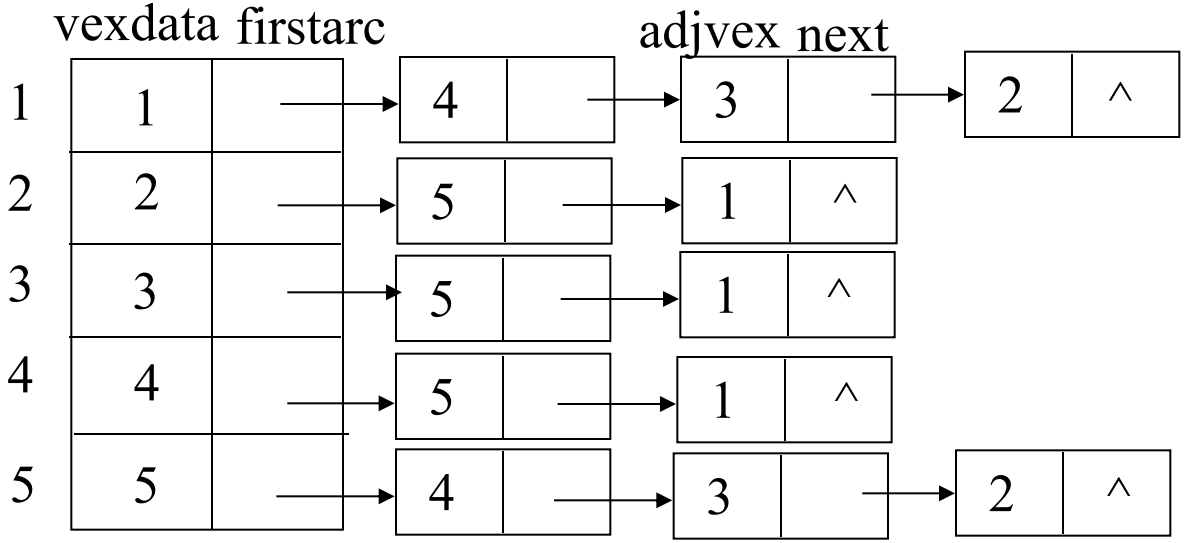
为了标记图中顶点是否被访问过，同样需要一个访问标记数组；其次，为了依次访问与v_i相邻接的各个顶点，需要附加一个队列来保存访问v_i的相邻接的顶点。

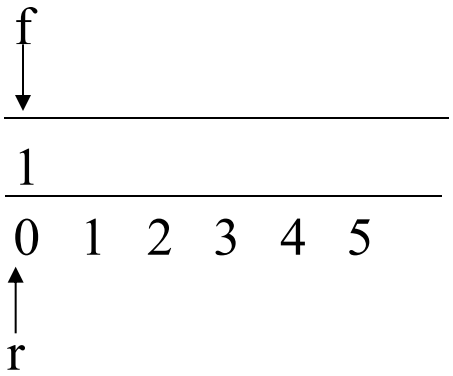
```
void BFS_traverse_Graph(ALGraph &G)
  {  int k ,v , w ;
     ArcNode  *p
     InitQueue(Q)  /*  建立空队列并初始化  */
     for (k=0 ; k<G.vexnum ; k++)
        Visited[k]=FALSE ;     /*  访问标志初始化  */
     for (k=0 ; k<G.vexnum ; k++)
        {     if (!Visited[k])              /*   顶点v尚未访问   */
              {  v=G.vertices[k].data;   printf(v);
                 EnQueue(Q,k);     /*   顶点v的下标k入对   */
                  while (!QueueEmpty(Q))
```

```
        {  DeQueue(Q,u);
           p=G.vertices[u].firstarc ;
            while (p!=NULL)
                if (!Visited[p->adjvex])
                  { v=G.vertices[p->adjvex].data;
                    printf(v);
                    EnQueue(Q,p->adjvex);
                    p=p->nextarc ;
                   }
            }   /*  end  while  */
     }    /*  end  if  */
  }  /*  end for  */
}
```

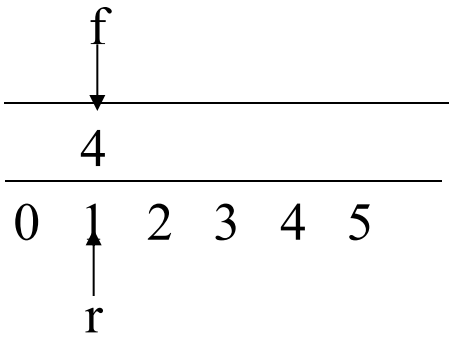
例



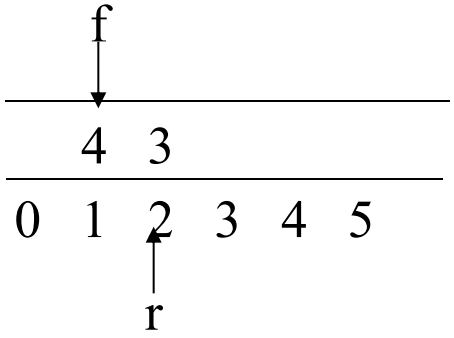




遍历序列：1

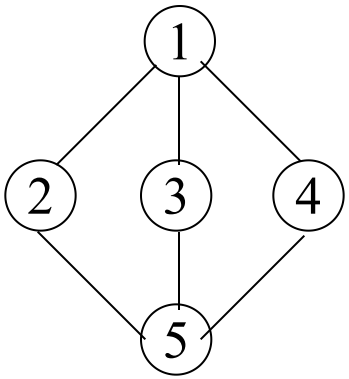


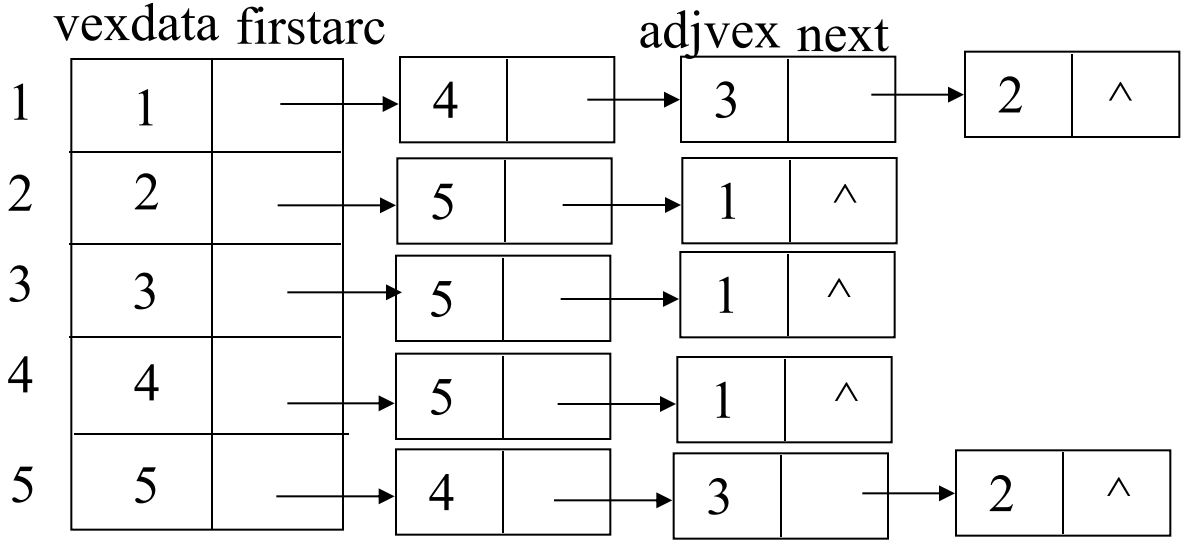
遍历序列：1 4

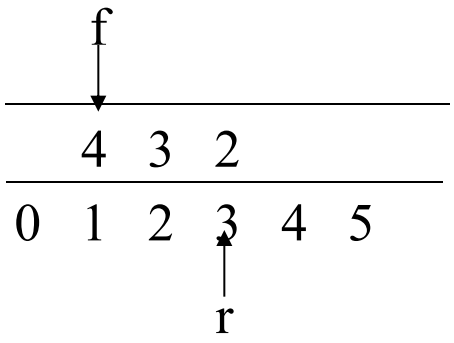


遍历序列：1 4 3

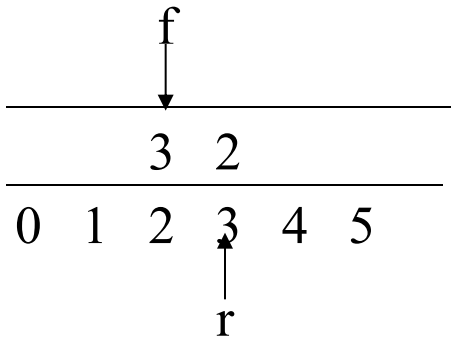
例



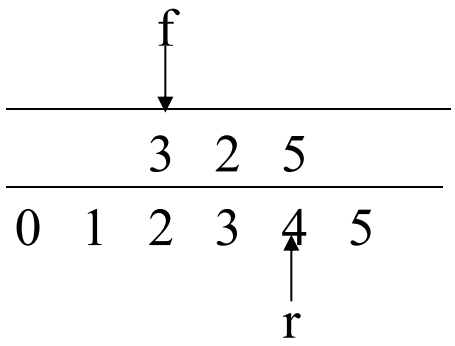




遍历序列：1 4 3 2

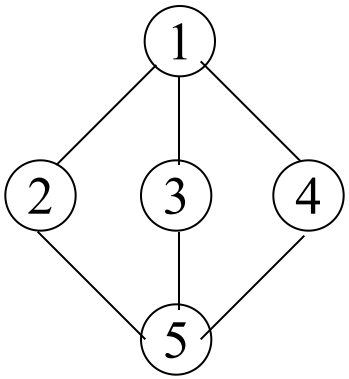


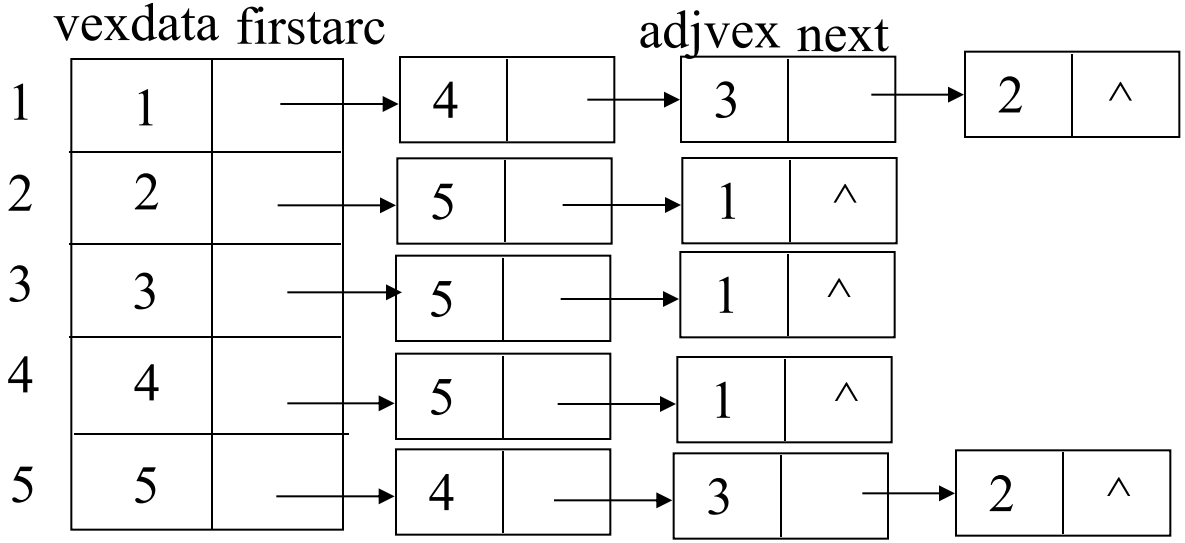
遍历序列：1 4 3 2

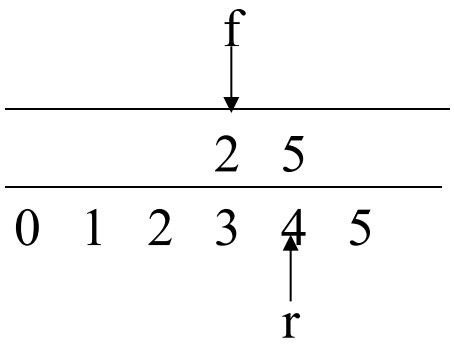


遍历序列：1 4 3 2 5

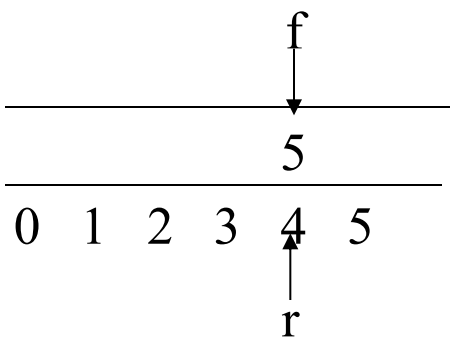
例



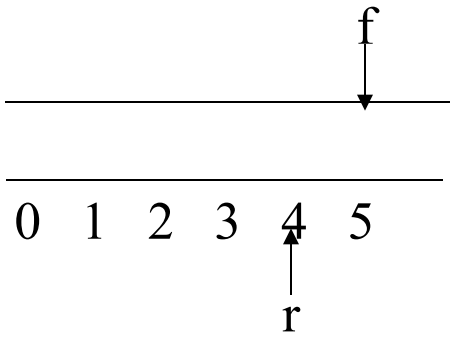




遍历序列：1 4 3 2 5



遍历序列：1 4 3 2 5



遍历序列：1 4 3 2 5

已知邻接矩阵求遍历序列

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

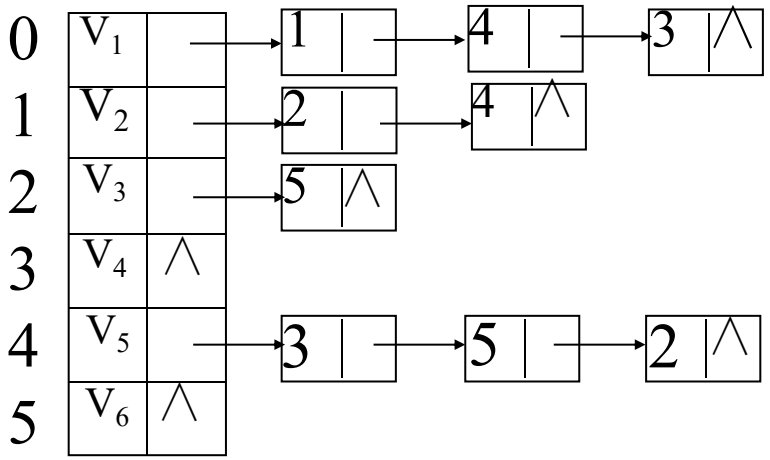
深度优先搜索：

V1->V2->V3->V4->V5

广度优先搜索：

V1->V2->V3->V4->V5

已知邻接表求遍历序列



深度优先遍历：V1->V2->V3->V6->V5->V4

广度优先遍历：V1->V2->V5->V4->V3->V6

图的遍历的时间复杂度

- 邻接矩阵为存储结构: $O(n^2)$
- 邻接表为存储结构: $O(n+e)$
- 空间复杂度均为: $O(n)$

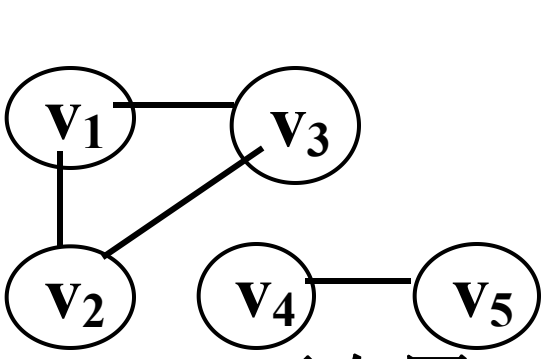
7.4 图的连通性问题

7.4.1 无向图的连通分量与生成树

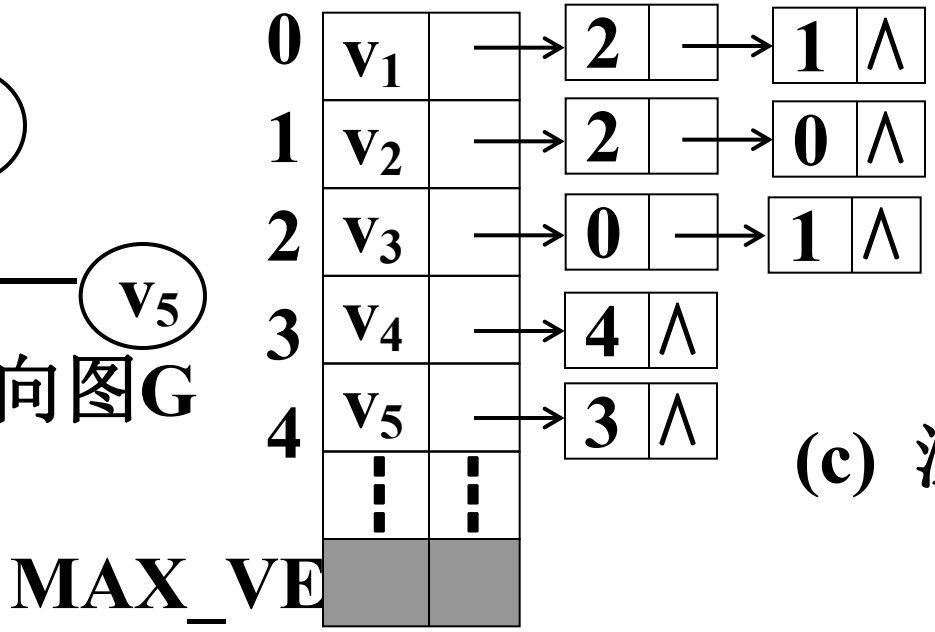
对于无向图，对其进行遍历时：

- ◆ 若是连通图：仅需从图中任一顶点出发，就能访问图中的所有顶点；
- ◆ 若是非连通图：需从图中多个顶点出发。每次从一个新顶点出发所访问的顶点集序列恰好是各个连通分量的顶点集；

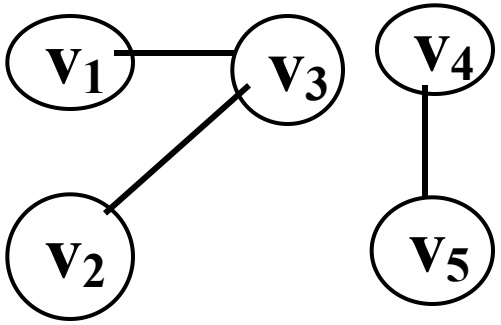
下图所示的无向图是非连通图，按图中给定的邻接表进行深度优先搜索遍历，2次调用DFS所得到的顶点访问序列集是：{ v1 ,v3 ,v2}和{ v4 ,v5 }



(a) 无向图G



(b) G的邻接链表



(c) 深度优先生成森林

无向图及深度优先生成森林

（1）若G=(V,E)是无向连通图， 顶点集和边集分别是V(G)，E(G)。若从G中任意点出发遍历时， E(G)被分成两个互不相交的集合：

T(G)：遍历过程中所经过的边的集合；

B(G)：遍历过程中未经过的边的集合；

显然： $E(G)=T(G)\cup B(G)$，$T(G)\cap B(G)=\varnothing$

显然，图G'=(V, T(G))是G的极小连通子图，且G'是一棵树。G'称为图G的一棵生成树。

从任意点出发按DFS算法得到生成树G'称为深度优先生成树；按BFS算法得到的G'称为广度优先生成树。

⑵ 若**G=(V,E)**是无向非连通图，对图进行遍历时得到若干个连通分量的顶点集：$V_1(G)$,$V_2(G)$,…,$V_n(G)$和相应所经过的边集：$T_1(G)$,$T_2(G)$, …,$T_n(G)$ 。

则对应的顶点集和边集的二元组：$G_i=(V_i(G),T_i(G))$ $(1\leqq i\leqq n)$是对应分量的生成树，所有这些生成树构成了原来非连通图的生成森林。

说明：当给定无向图要求画出其对应的生成树或生成森林时，若给出了相应的邻接表，则必须根据邻接表画出其对应的生成树或生成森林。

画一个图，然后画出不同的邻接表形式，给出DFS树和BFS树。

7.4.2 最小生成树

如果连通图是一个带权图，则其生成树中的边也带权，生成树中所有边的权值之和称为生成树的代价。

最小生成树：带权连通图中代价最小的生成树称为最小生成树。

最小生成树在实际中具有重要用途，如设计通信网。设图的顶点表示城市，边表示两个城市之间的通信线路，边的权值表示建造通信线路的费用。n个城市之间最多可以建$n\times(n-1)/2$条线路，如何选择其中的n-1条，使总的建造费用最低?

构造最小生成树的算法有许多，基本原则是：

- 尽可能选取权值最小的边，但不能构成回路；
- 选择n-1条边构成最小生成树。

以上的基本原则是基于MST性质：

设G=(V，E)是一个带权连通图，U是顶点集V的一个非空子集。若u∈U，v∈V-U，且(u, v)是U中顶点到V-U中顶点之间权值最小的边，则必存在一棵包含边(u, v)的最小生成树。

普里姆(Prim)算法

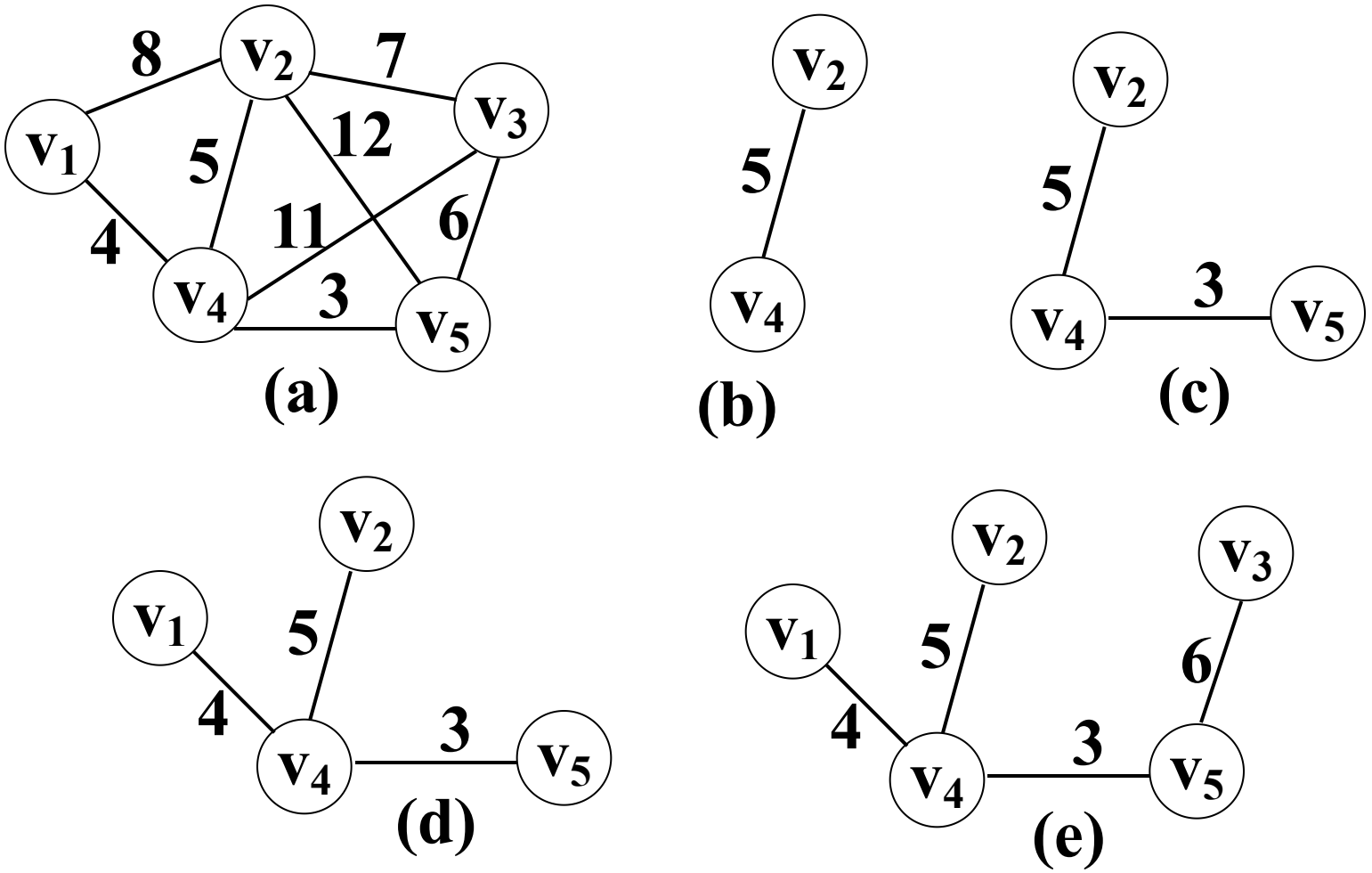
从连通网N=(V, E)中找最小生成树T=(U, TE) 。

1 算法思想

(1) 若从顶点v_0出发构造，U={v_0}，TE={};

(2) 先找权值最小的边(u, v)，其中u∈U且v∈V-U，则U= U∪{v}，TE=TE∪{(u, v)} ；

(3) 重复(2)，直到U=V为止。则TE中必有n-1条边，T=(U, TE)就是最小生成树。



按prim算法从v2出发构造最小生成树的过程

2 算法实现说明

设用邻接矩阵(二维数组)表示图，两个顶点之间不存在边的权值为机内允许的最大值。

为便于算法实现，设置一个一维数组**closedge[n]**，用来保存**V-U**中各顶点到**U**中顶点具有权值最小的边。数组元素的类型定义是：

```
struct
  {  int  adjvex ;      /*  边所依附于U中的顶点  */
     int  lowcost ;   /*  该边的权值  */
  }closedge[MAX_EDGE] ;
```

例如： **closedge[j].adjvex=k**，表明边(v_j, v_k)是**V-U**中顶点v_j到**U**中权值最小的边，而顶点v_k是该边所依附的**U**中的顶点。**closedge[j].lowcost**存放该边的权值。

假设从顶点v_s开始构造最小生成树。初始时令：

Closedge[s].lowcost=0：表明顶点v_s首先加入到**U**中；
Closedge[k].adjvex=s，
Closedge[k].lowcost=cost(k, s)

表示**V-U**中的各顶点到**U**中权值最小的边($k \neq s$)，**cost(k, s)**表示边(v_k, v_s)权值。

3 算法步骤

(1) 从closedge中选择lowcost域值最小的顶点Vk，将Vk加入U中，然后做：

① 置closedge[k].lowcost为0，表示v_k已加入到U中。

② 根据新加入的v_k更新closedge中每个元素：

$\forall v_i \in V-U$，若cost(i,k)≦colsedge[i].lowcost，表明在U中新加入顶点v_k后，(v_i, v_k)成为v_i到U中权值最小的边，置：

$$\begin{cases} Closedge[i].lowcost=cost(i, k) \\ Closedge[i].adjvex=k \end{cases}$$

(2) 重复(1)n-1次就得到最小生成树。

在Prim算法中，图采用邻接矩阵存储，所构造的最小生成树用一维数组存储其n-1条边，每条边的存储结构描述：

```
typedef struct MSTEdge
   {  int  vex1, vex2 ;    /*  边所依附的图中两个顶点 */
      WeightType  weight ;     /*  边的权值  */
   }MSTEdge ;
#define INFINITY  MAX_VAL     /* 最大值 */
void Prim_MST(MGraph  &G , int u, MSTEdge * &TE)
   /*   从第u个顶点开始构造图G的最小生成树   */
   {
```

```
int j , k , v , min ;
for (j=0; j<G.vexnum; j++)
   {  closedge[j].adjvex=u  ;
      closedge[j].lowcost=G.arcs[j][u].adj  ;
   }  /*  初始化数组closedge[n]  */
closedge[u].lowcost=0 ;    /*  初始时置U={u}  */
TE=(MSTEdge *)malloc((G.vexnum-
1)*sizeof(MSTEdge)) ;
for (j=0; j<G.vexnum-1; j++)
   { min= INFINITY ;
      for (v=0; v<G.vexnum; v++)
         if  (closedge[v].lowcost!=0&&
      closedge[v].Lowcost<min)
```

```
        { min=closedge[v].lowcost ; k=v ; }
      TE[j].vex1=closedge[k].adjvex ;
      TE[j].vex2=k ;
      TE[j].weight=closedge[k].lowcost ;
      closedge[k].lowcost=0 ;    /*  将顶点k并入U中  */
      for (v=0; v<G.vexnum; v++)
        if (G.arcs[v][k].adj<closedge[v]. lowcost)
          { closedge[v].lowcost= G.arcs[v][k].adj ;
            closedge[v].adjvex=k ;
          } /*  修改数组closedge[n]的各个元素的值  */
    }
} /*  求最小生成树的Prim算法  */
```

构造过程中辅组数组closedge中各分量的值的变化情况

i / closedge	0	1	2	3	4	U	V-U	K
adjvex	v_2		v_2	v_2	v_2	$\{v_2\}$	$\{v_1, v_3, v_4, v_5\}$	3
lwcost	8		7	5	12			
adjvex	v_4		v_2	v_2	v_4	$\{v_2, v_4\}$	$\{v_1, v_3, v_5\}$	4
lwcost	4		7	0	3			
adjvex	v_4		v_5	v_2	v_4	$\{v_2, v_4, v_5\}$	$\{v_1, v_3\}$	0
lwcost	4		6	0	0			
adjvex	v_4		v_5	v_2	v_4	$\{v_2, v_4, v_5, v_1\}$	$\{v_3\}$	2
lwcost	0		6	0	0			
adjvex	v_4		v_5	v_2	v_4	$\{v_2, v_4, v_5, v_1, v_3\}$	$\{\}$	
lwcost	0		0	0	0			

算法分析： 设带权连通图有n个顶点，则算法的主要执行是二重循环： 求closedge中权值最小的边，频度为n-1； 修改closedge数组，频度为n 。因此，整个算法的时间复杂度是$O(n^2)$，与边的数目无关。

适用于求边稠密的网的最小生成树

克鲁斯卡尔(Kruskal)算法

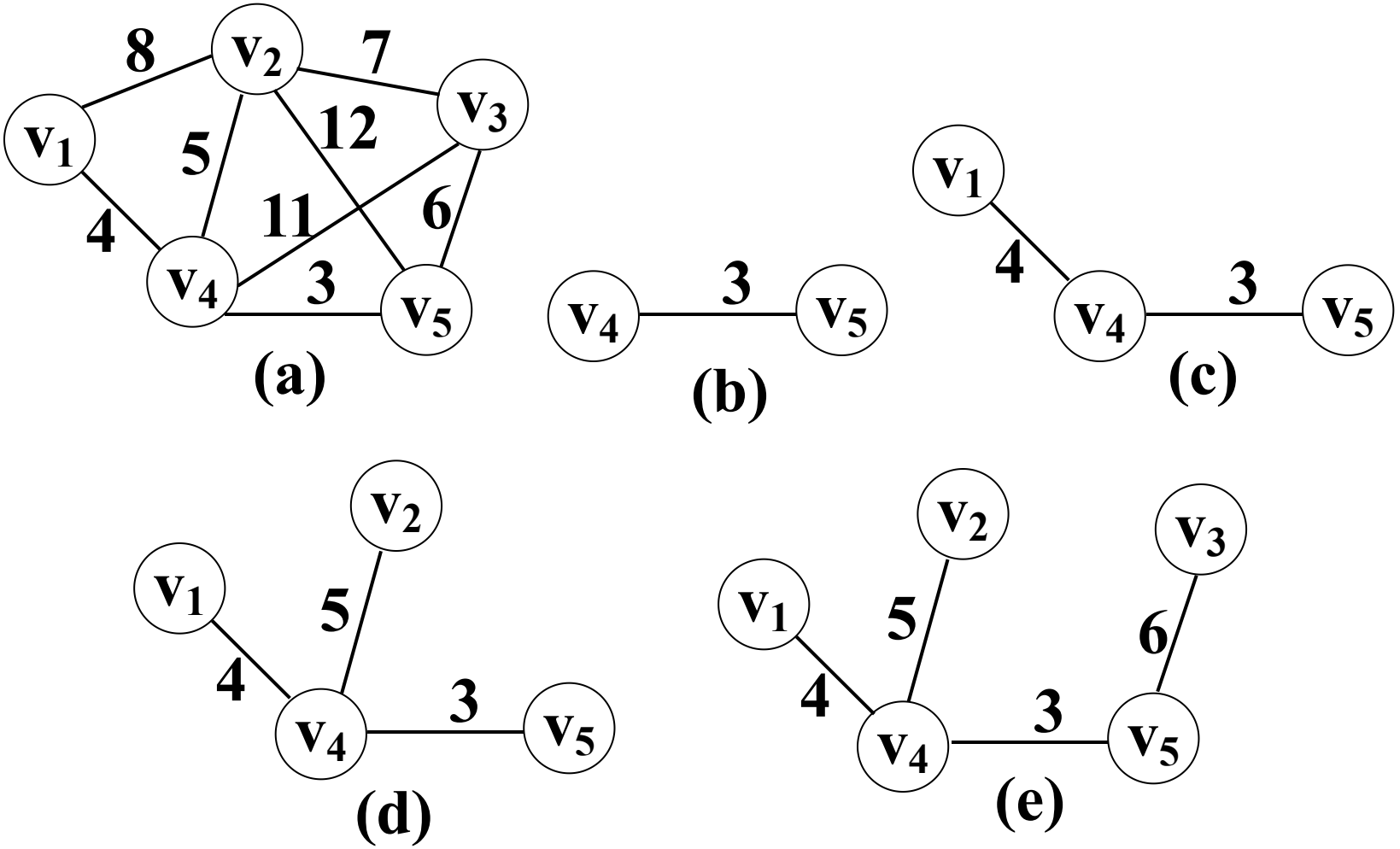
1 算法思想

设G=(V, E)是具有n个顶点的连通网，T=(U, TE)是其最小生成树。初值：U=V，TE={}。

对G中的边按权值大小从小到大依次选取。

(1) 选取权值最小的边(v_i，v_j)，若边(v_i，v_j)加入到TE后形成回路，则舍弃该边(边(v_i，v_j)；否则，将该边并入到TE中，即TE=TE∪{(v_i，v_j)}。

(2) 重复(1)，直到TE中包含有n-1条边为止。



按kruskal算法构造最小生成树的过程

克鲁斯卡尔算法的时间复杂度是$O(e\log e)$，适用于求稀疏的网的最小生成树。

普里姆算法的时间复杂度是$O(n^2)$，与边的数目无关，适用于求边稠密的网的最小生成树

7.5有向无环图及其应用

7.5.1 拓扑排序

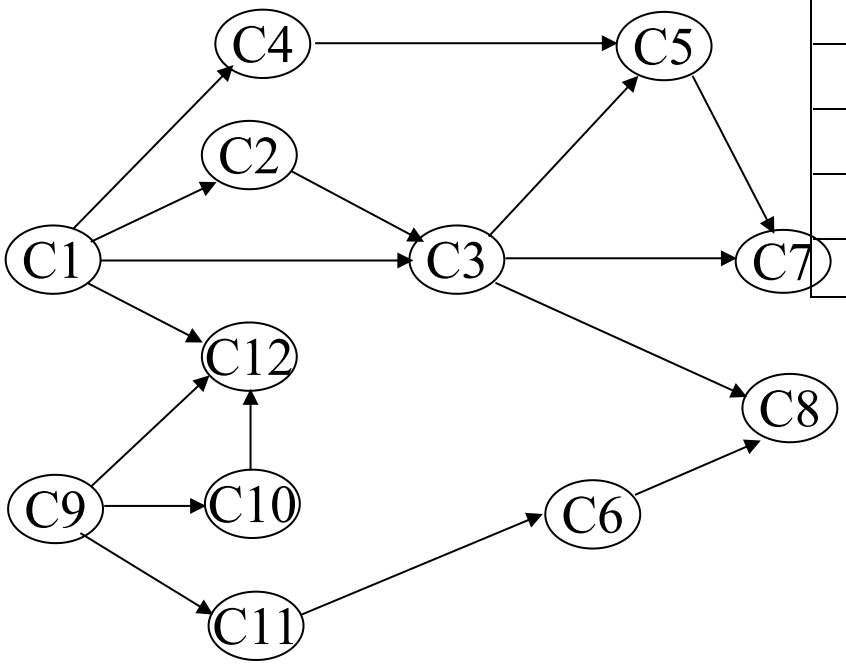
- 问题提出：学生选修课程问题
  - 顶点——表示课程
  - 有向弧——表示先决条件，若课程i是课程j的先决条件，则图中有弧<i,j>
  - 学生应按怎样的顺序学习这些课程，才能无矛盾、顺利地完成学业——拓扑排序
- 定义
  - AOV网——用顶点表示活动，用弧表示活动间优先关系的有向图称为顶点表示活动的网(Activity On Vertex network)，简称AOV网
    - 若<vi,vj>是图中有向边，则vi是vj的直接前驱；vj是vi的直接后继
    - AOV网中不允许有回路，否则意味着某项活动以自己为先决条件

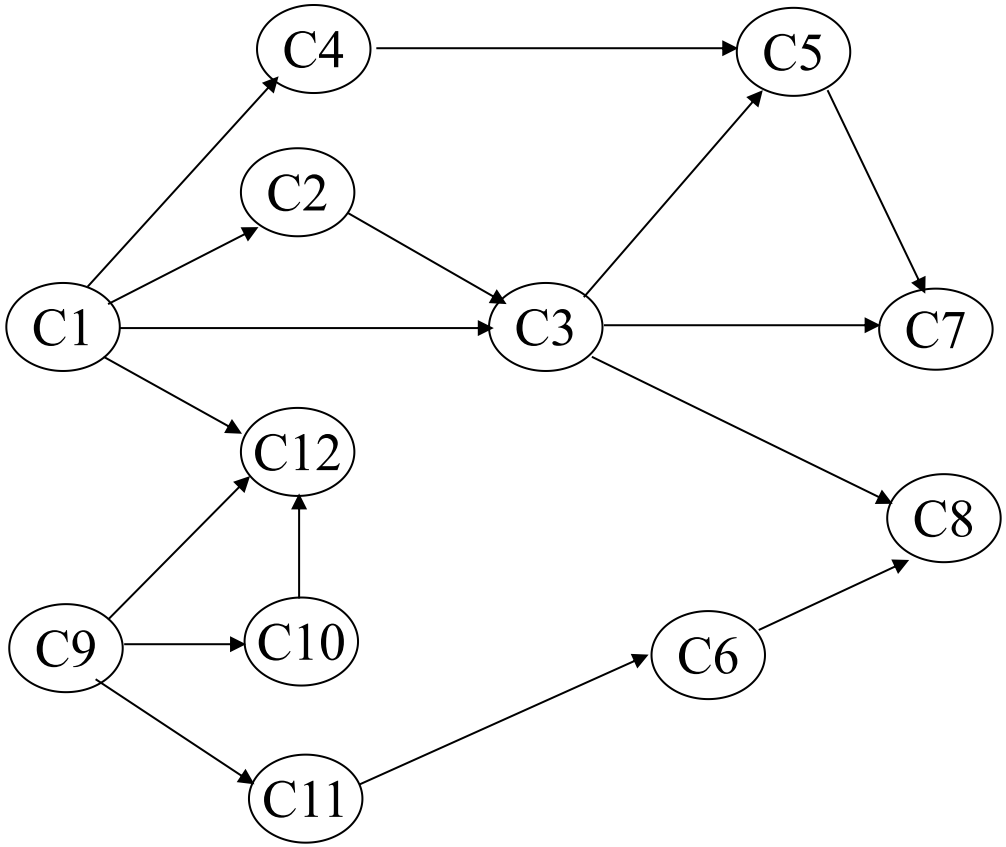
拓扑排序：

- 定义：把AOV网络中各顶点按照它们相互之间的优先关系排列成一个线性序列的过程叫~
- 作用：检测AOV网中是否存在环。对有向图构造其顶点的拓扑有序序列，若网中所有顶点都在它的拓扑有序序列中，则该AOV网必定不存在环
- 拓扑排序的方法
  - 在有向图中选一个没有前驱（入度为0）的顶点且输出之
  - 从图中删除该顶点和所有以它为尾的弧
  - 重复上述两步，直至全部顶点均已输出；或者当图中不存在无前驱的顶点为止

例

课程代号	课程名称	先修棵
C1	程序设计基础	无
C2	离散数学	C1
C3	数据结构	C1,C2
C4	汇编语言	C1
C5	语言的设计和分析	C3,C4
C6	计算机原理	C11
C7	编译原理	C3.C5
C8	操作系统	C3,C6
C9	高等数学	无
C10	线性代数	C9
C11	普通物理	C9
C12	数值分析	C1,C9,C10

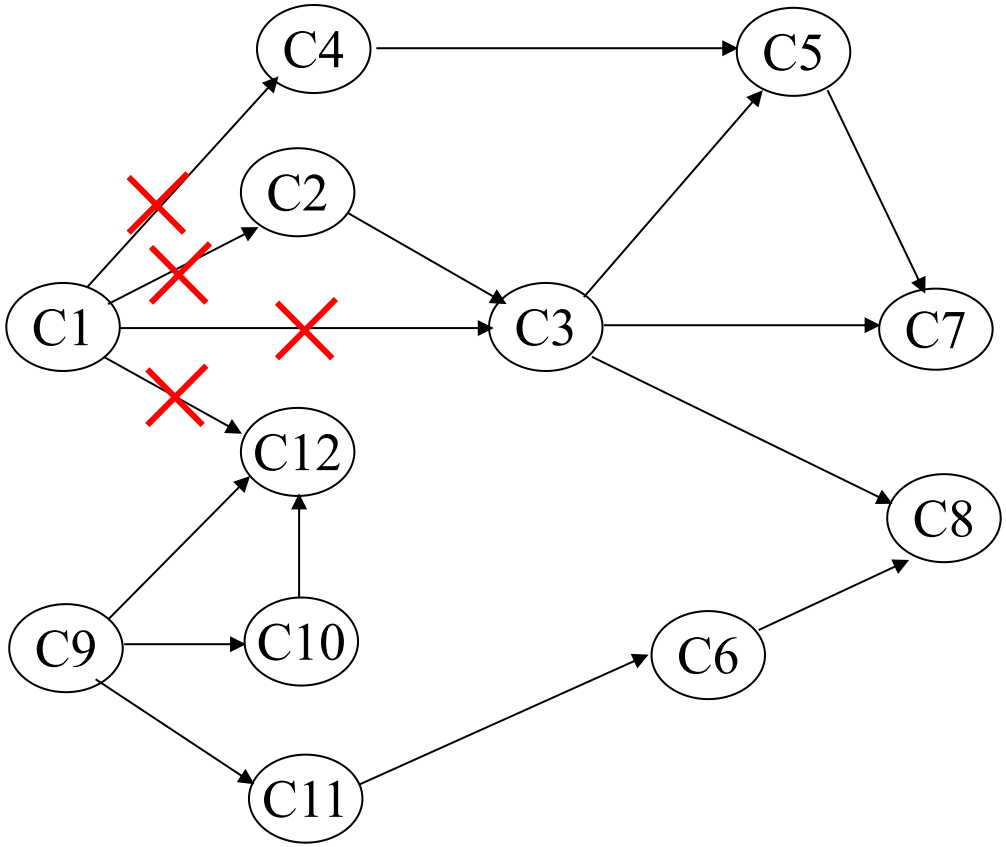


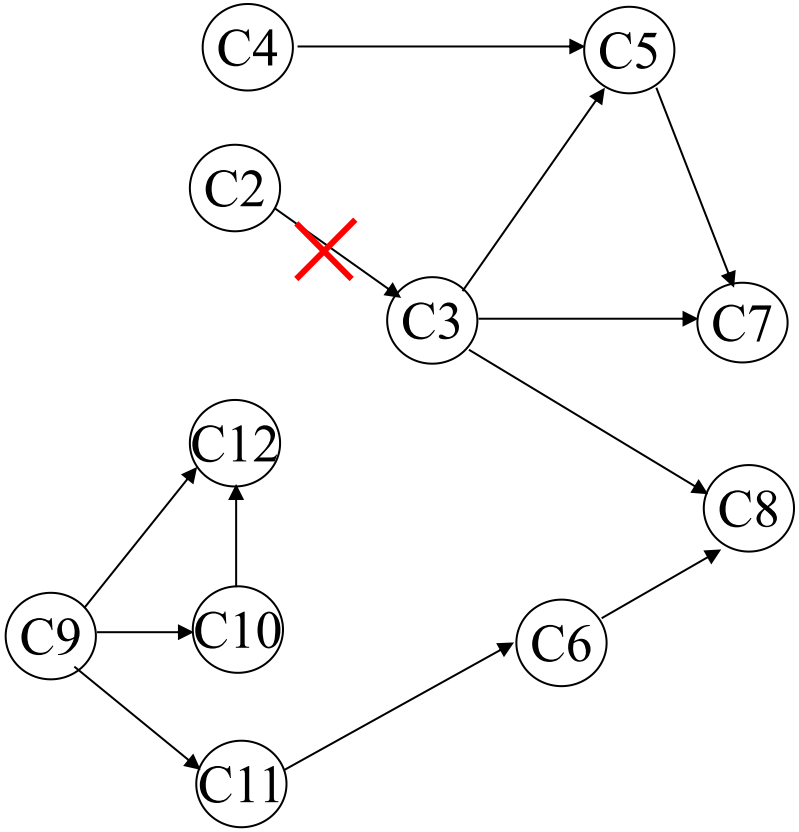


拓扑序列：C1--C2--C3--C4--C5--C7--C9--C10--C11--C6--C12--C8

或　　：C9--C10--C11--C6--C1--C12--C4--C2--C3--C5--C7--C8

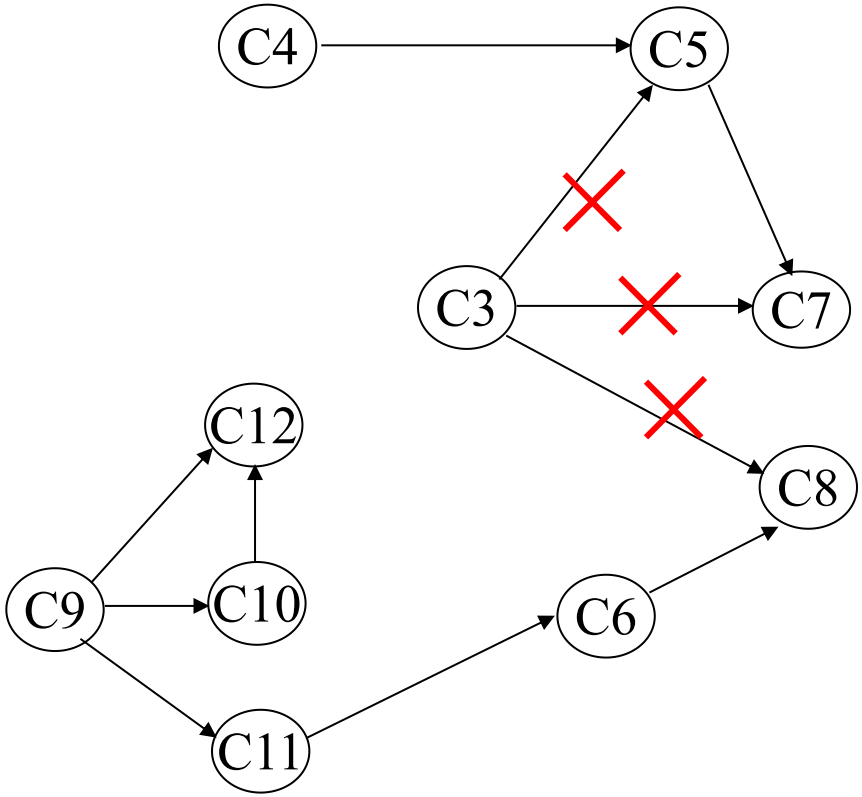
一个AOV网的拓扑序列不是唯一的


C4
C5
C2
C1
C3
C7
C12
C8
C9
C10
C6
C11



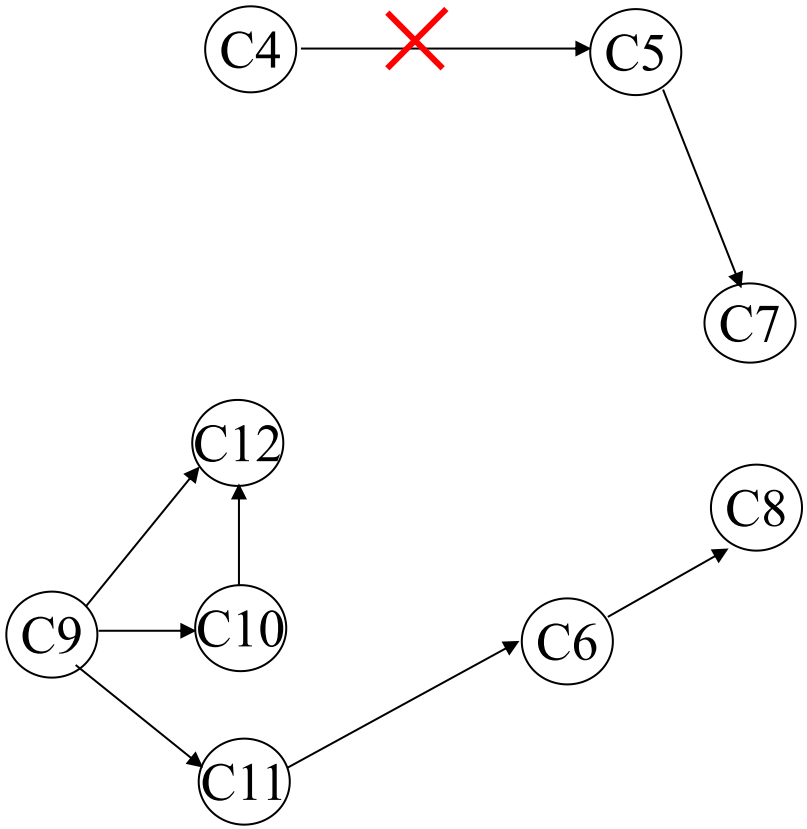
拓扑序列：C1

（1）



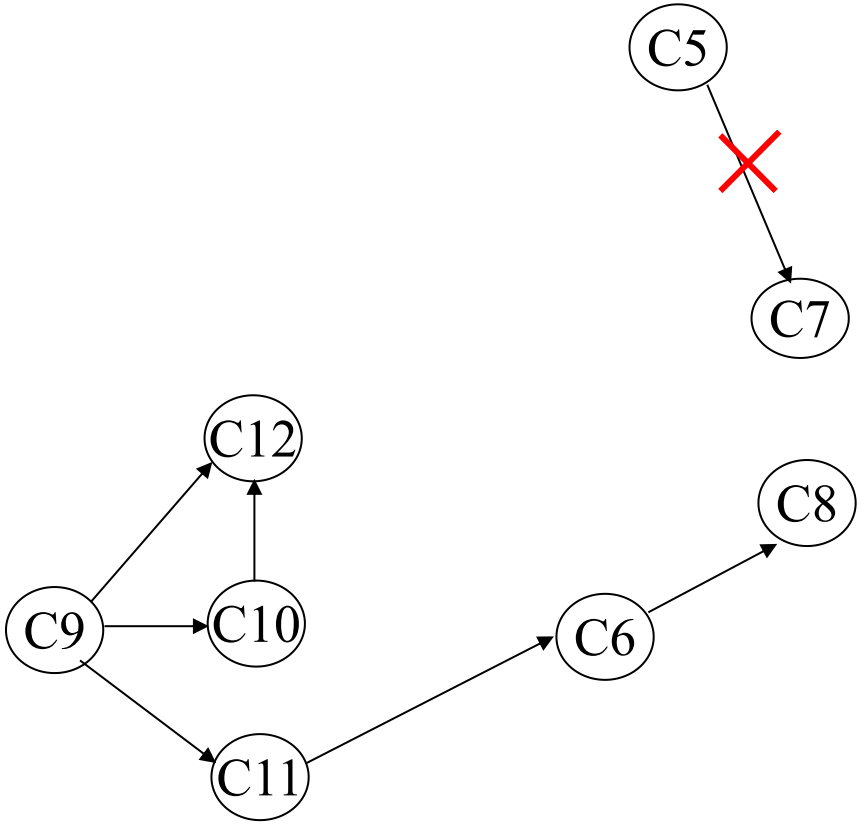
拓扑序列：C1--C2

（2）



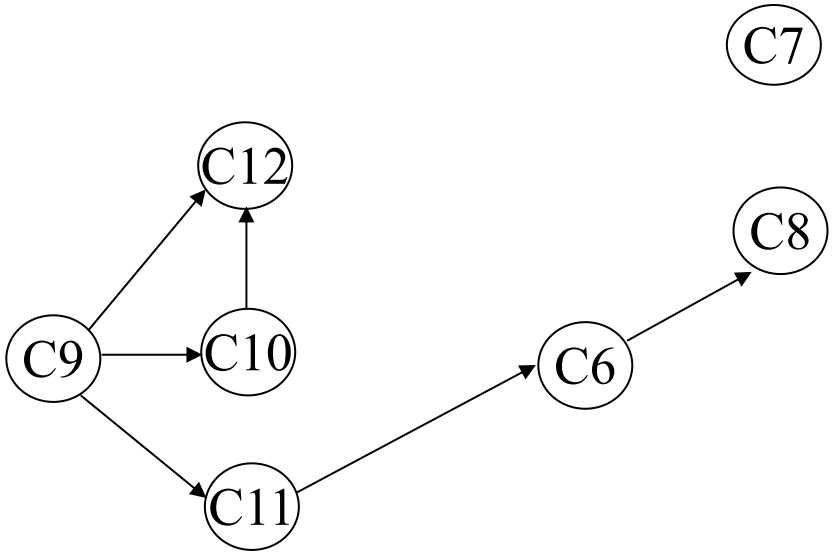
拓扑序列：C1--C2--C3

（3）



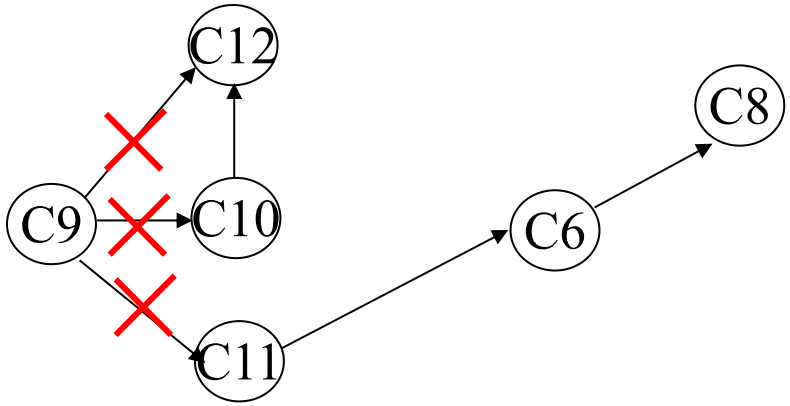
拓扑序列：C1--C2--C3--C4

（4）



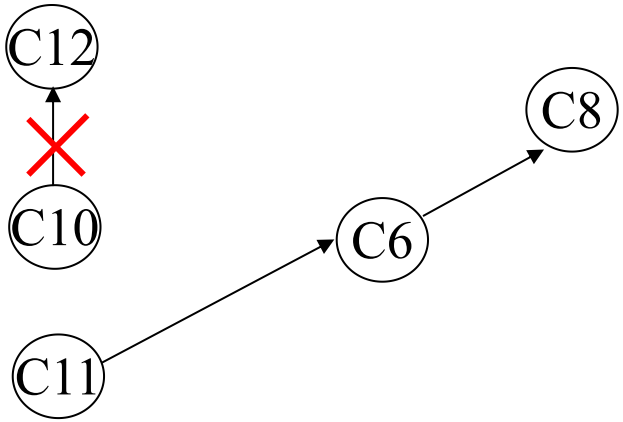
拓扑序列：C1--C2--C3--C4--C5

（5）

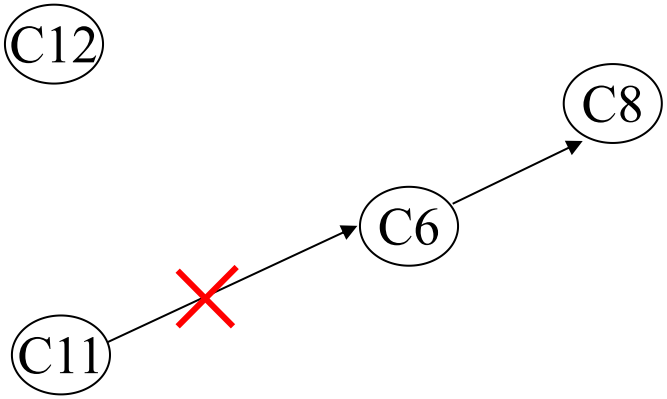


拓扑序列：C1--C2--C3--C4--C5--C7

（6）



拓扑序列：C1--C2--C3--C4--C5--C7--C9

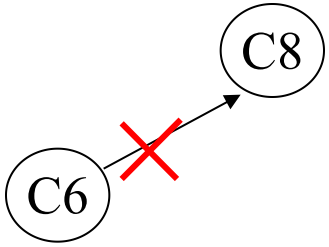


拓扑序列：C1--C2--C3--C4--C5--C7--C9

--C10

（8）

C12



（9）

拓扑序列：C1--C2--C3--C4--C5--C7--C9
--C10--C11

C12

C8

（10）

拓扑序列：C1--C2--C3--C4--C5--C7--C9
--C10--C11--C6

C8

（11）

拓扑序列：C1--C2--C3--C4--C5--C7--C9
--C10--C11--C6--C12

拓扑序列：C1--C2--C3--C4--C5--C7--C9
--C10--C11--C6--C12--C8

（12）

- 算法实现
  - 以邻接表作存储结构
  - 把邻接表中所有入度为0的顶点进栈（栈里存放顶点下标）
  - 栈非空时，输出栈顶元素vj并退栈；在邻接表中查找Vj的直接后继Vk，把Vk的入度减1；若Vk的入度为0则进栈
  - 重复上述操作直至栈空为止。若栈空时输出的顶点个数不是n，则有向图有环；否则，拓扑排序完毕

算法描述

```
int  Topologic_Sort(ALGraph G, int indegree[])
    {   FindInDegree(G, indegree) ;   /*  统计各顶点的入度  */
        InitStack(S);
        for (k=0; k<G.vexnum; k++)
            if  (indegree[k]==0)
                push(s,k);
      count=0;
      while(!StackEmpty(S))  {
        pop(s,i);  printf(i, G.vertices[i].data); count++;
       for(p=G.vertices[i].firstarc; p; p=p->nextarc) {
          k=p->adjvex;   if(!—indegree[k])  push(s,k);}
      }
   if(count<G.vexnum)  return 0
   else return 1;}
```

(2) 统计各顶点入度的函数

```
void FindInDegree(ALGraph &G, int indegree[])
   {  int k ; ArcNode *p ;
      for (k=0; k<G.vexnum; k++)
            indegree[k]=0 ;     /*  顶点入度初始化  */
      for (k=0; k<G.vexnum; k++)
         {  p=G.vertices[k].firstarc ;
            while (p!=NULL)     /*  顶点入度统计  */
               {  t=G.vertices[p->adjvex]
                  indegree[t]++ ;
                   p=p->nextarc ;   }
         }
   }
```

算法分析：设AOV网有n个顶点，e条边，则算法的主要执行是：

- 统计各顶点的入度：时间复杂度是O(n+e)；
- 入度为0的顶点入栈：时间复杂度是O(n)；
- 排序过程：顶点入栈和出栈操作执行n次，入度减1的操作共执行e次，时间复杂度是O(n+e)；

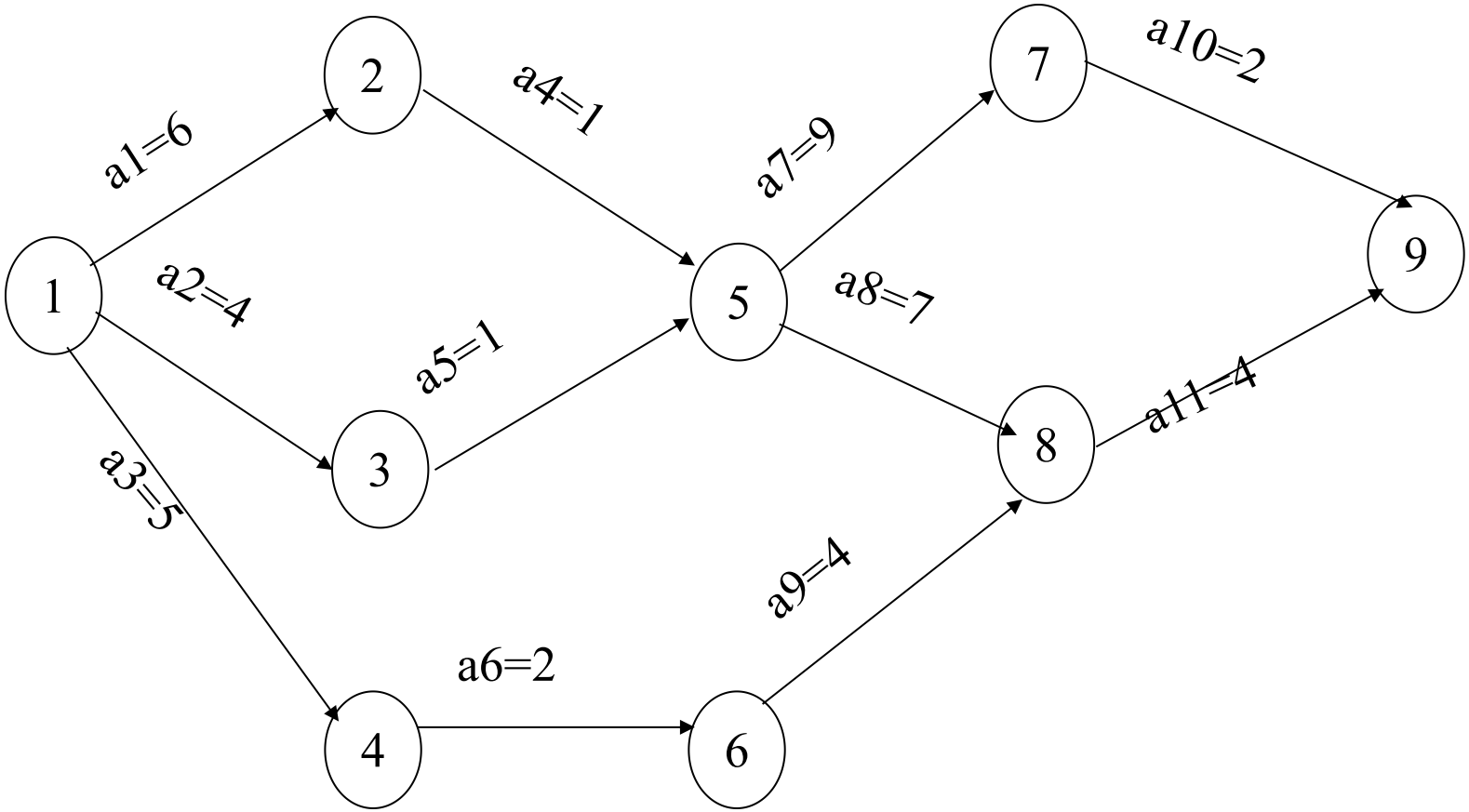
因此，整个算法的时间复杂度是O(n+e)。

7.5.2 关键路径

- 1956年，美国杜邦公司提出关键路径法，并于1957年首先用于1000万美元进行化工厂建设，工期比原计划缩短了4个月。杜邦公司在采用关键路径法的一年中，节省了100万美元。

关键路径

- 问题提出：
  - 把工程计划表示为有向图，用顶点表示事件，弧表示活动，弧上的权值表示活动持续的时间；
  - 每个事件表示在它之前的活动已完成，在它之后的活动可以开始
- 例 设一个工程有11项活动，9个事件
  - 事件 V1——表示整个工程开始
  - 事件V9——表示整个工程结束



- 问题：
  - （1）完成整项工程至少需要多少时间？
  - （2）哪些活动是影响工程进度的关键活动？

若干定义

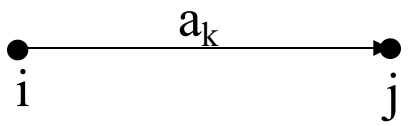
- AOE网(Activity On Edge)：也叫边表示活动的网。AOE网是一个带权的有向无环图，其中顶点表示事件，弧表示活动，权表示活动持续时间
- 路径长度：路径上各活动持续时间之和
- 关键路径：从开始顶点到完成顶点的路径中长度最长的叫~
- 关键活动：关键路径上的活动叫~。

几个变量：

- ve(j)：表示事件Vj的最早发生时间
- vl(j)：表示事件Vj的最迟发生时间
- e(k)：表示活动a_k<vi,vj>的最早开始时间=ve(i)
- l(k)：表示活动a_k<vi,vj>的最迟开始时间
- l(k)-e(k)：表示完成活动a_k的时间余量
- 当l(k)=e(k)时，ak是关键活动。

问题分析

- 如何找e(k)=l(k)的关键活动？
  - 设活动a_k用弧<i,j>表示，其持续时间记为：dut(<i,j>)则有：
    - （1）$e(a_k)=ve(i)$
    - （2）$l(a_k)=vl(j)-dut(<i,j>)$
  - 如何求ve(i)和vl(j)？



- (1)从ve(1)=0开始向前递推

$$ve(j)=\underset{i}{Max}\{ve(i)+dut(<i,j>)\},<i,j>\in S,2\le j\le n$$

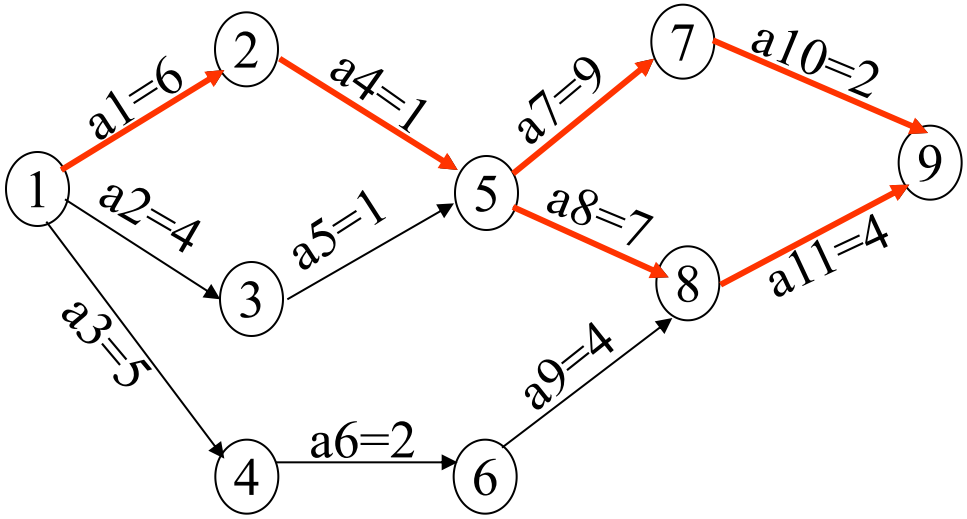
其中 S是所有以 j为头的弧的集合

- (2)从vl(n)=ve(n)开始向后递推

$$vl(i)=\underset{j}{Min}\{vl(j)-dut(<i,j>)\},<i,j>\in S,1\le i\le n-1$$

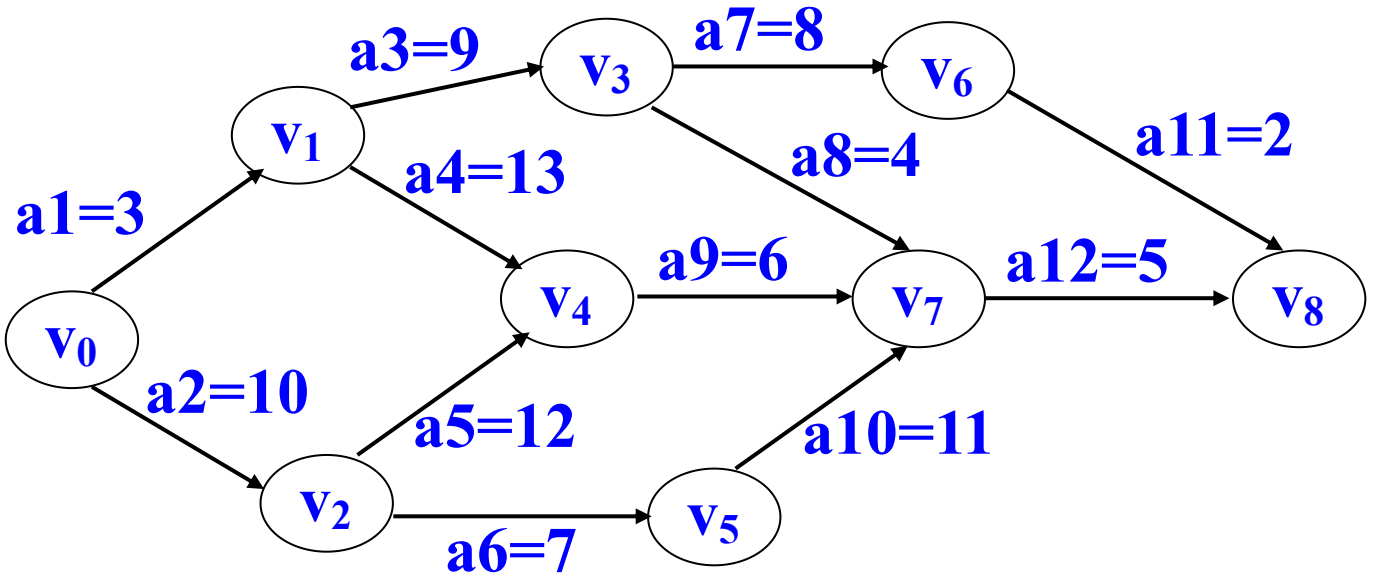
其中 S是所有以 i为尾的弧的集合

- 求关键路径步骤
  - 求ve(i)
  - 求vl(j)
  - 求e(k)
  - 求l(k)
  - 计算l(k)-e(k)



顶点	ve	vl
V1	0	0
V2	6	6
V3	4	6
V4	5	8
V5	7	7
V6	7	10
V7	16	16
V8	14	14
V9	18	18

活动	e	l	l-e
a1	0	0	0 ✓
a2	0	2	2
a3	0	3	3
a4	6	6	0 ✓
a5	4	6	2
a6	5	8	3
a7	7	7	0 ✓
a8	7	7	0 ✓
a9	7	10	3
a10	16	16	0 ✓
a11	14	14	0 ✓



顶点	v_0	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8
ve(i)	0	3	10	12	22	17	20	28	33
v*l*(i)	0	9	10	23	22	17	31	28	33

活动	a1	a2	a3	a4	a5	a6	a7	a8	a9	a10	a11	a12
e(i)	0	0	3	3	10	10	12	12	22	17	20	28
l(i)	6	0	14	9	10	10	23	24	22	17	31	28

2 求AOE中关键路径和关键活动

(1) 算法思想

① 利用拓扑排序求出AOE网的一个拓扑序列；

② 从拓扑排序的序列的第一个顶点(源点)开始，按拓扑顺序依次计算每个事件的最早发生时间ve(i)；

③ 从拓扑排序的序列的最后一个顶点(汇点)开始，按逆拓扑顺序依次计算每个事件的最晚发生时间vl(i)；

(2) 拓扑排序求事件最早开始时间算法

```
int  TopologicOrder(ALGraph G, stack &T)
   {   FindInDegree(G, indegree) ;   /*  统计各顶点的入度  */
       InitStack(S); count=0; ve[0…G.vexnum-1]=0;
       for (k=0; k<G.vexnum; k++)
           if  (indegree[k]==0)
                push(s,k);
      count=0;
      while(!StackEmpty(S))  {
        pop(s, j); push(T, j); count++;
       for(p=G.vertices[j].firstarc; p; p=p->nextarc) {
          k=p->adjvex;   if(!—indegree[k])  push(s,k);
           if(ve[j]+*(p->info)>ve[k]) ve[k]=ve[j]+*(p->info); }
      }     if(count<G.vexnum)  return error
    else return ok;}
```

（3）关键路径算法实现

```
void criticalpath(ALGraph *G)
   {  int j, k, m ; ArcNode *p ;
      if  (TopologicOrder(G, T)==ERROR) return -1;
      Vl[0..G.vexnum-1]=ve[G.vexnum-1];
      While(!StackEmpty(T))
For(pop(T,j), p=G.vertices[j].firtarc; p; p=p->nextarc){
          K=p->adjvex; dut=*(p->info);
          If(vl[k]-dut<vl[j])  vl[j]=vl[k]-dut;}
          For(j=0;j<G.vexnum;j++)
            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);
          }}
```

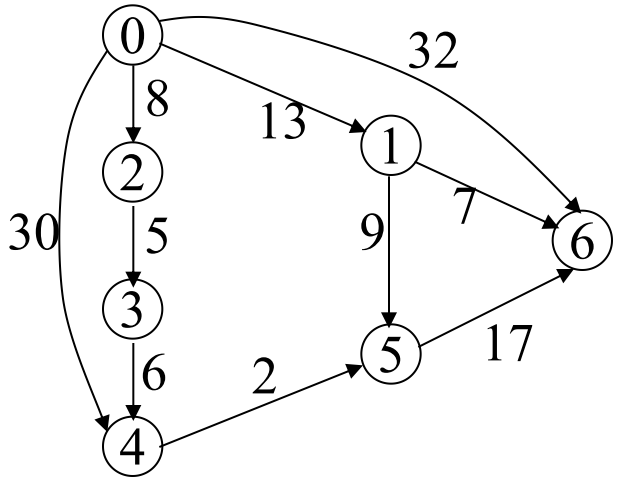
7.7 最短路径

- 问题提出
  - 用带权的有向图表示一个交通运输网，图中：
    - 顶点——表示城市
    - 边——表示城市间的交通联系
    - 权——表示此线路的长度或沿此线路运输所花的时间或费用等
  - 问题：从某顶点出发，沿图的边到达另一顶点所经过的路径中，各边上权值之和最小的一条路径——最短路径
  - 第一个顶点为源点，最后一个顶点为终点。

7.7.1 从某个源点到其余各顶点的最短路径

定义：对于给定的有向图G=(V，E)及单个源点V_0，求V_0到G的其余各顶点的最短路径。

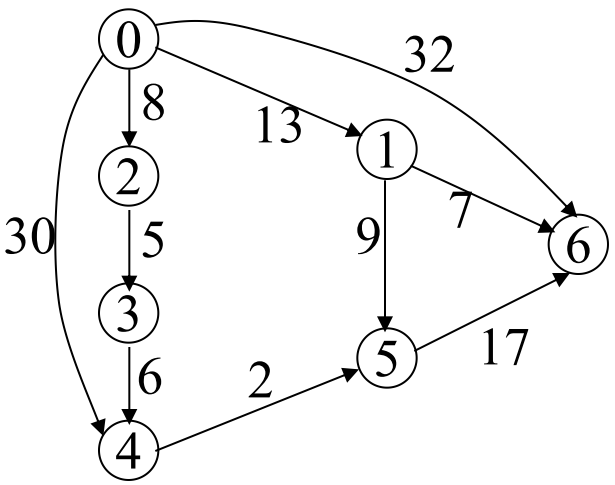
迪杰斯特拉(Dijkstra)算法：按路径长度递增的次序产生最短路径的算法。



最短路径	长度
<V0,V1>	13
<V0,V2>	8
<V0,V2,V3>	13
<V0,V2,V3,V4>	19
<V0,V2,V3,V4,V5>	21
<V0,V1,V6>	20

第一条长度最短的最短路径

- 计算从源点v0到其他所有顶点的直接路径，存入辅助向量D中。
- D[i]:表示从V0到vi的最短路径的长度。
- D[i]的初值：若vs到vi有弧，则D[i]为弧上的权值；否则为∞。



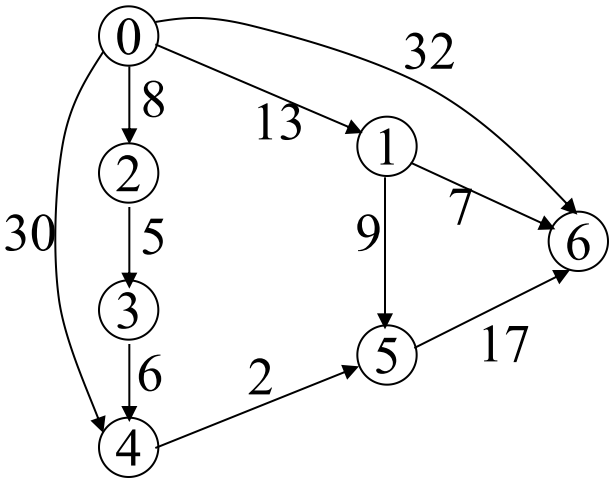
最短路径	长度
<V0,V1>	13
<V0,V2>	8
<V0,V2,V3>	13
<V0,V2,V3,V4>	19
<V0,V2,V3,V4,V5>	21
<V0,V1,V6>	20

D[1]=13; D[2]=8; D[3]= ∞; D[4]=30; D[5]= ∞; D[6]=32;
则最小的值D[2]=8是v0到v2的最短路径，也是第一条长度最短的最短路径。

一般化：D[j]=Min{D[i]| vi∈V} 的路径就是从v0出发的长度最短的一条最短路径。此路径为(v0, vj)。

下一条长度次短的最短路径是哪一条呢？

- 因为第一条长度最短的最短路径为(v0,v2)，则下一条长度次短的最短路径要么是从v0到某顶点的直接路径，要不是经过v2的路径。例如(v0,v1)或(v0,v2,v3)



最短路径	长度
<V0,V1>	13
<V0,V2>	8
<V0,V2,V3>	13
<V0,V2,V3,V4>	19
<V0,V2,V3,V4,V5>	21
<V0,V1,V6>	20

- 一般化，假设该次短路径的终点是vk，则该路径或者是(v0,vk)或者是(v0, vj, vk)。它的长度或者是从v0到vk的弧上的权值，或者是D[j]和从vj到vk的弧上的权值之和。

下一条最短路径

- 假设S为已求得最短路径的终点的集合，则可证明，下一条最短路径（设其终点为x）或者是弧（v0,x)，或者是中间只经过S种的顶点而最后到达顶点x的路径。
- 反证法：假设此路径上有一个顶点不在S种，则说明存在一条终点不在S而长度比此路径短的路径。但是，这是不可能的。因为我们是按照路径长度递增的次序来产生各最短路径的，故长度比此路径的所有路径均已产生，它们的终点必定在S中，即假设不成立。

- 一般化，下一条长度次短的最短路径的长度为：

$D[j]=Min\{D[i]|vi \in V-S\}$

其中，D[i]或者是弧(v0,vi)上的权值，

或者是$D[k](v_k \in S)$和弧(vk,vi)上的权值之和。

算法步骤

（1）令S={Vs}，用带权的邻接矩阵表示有向图，对图中每个顶点Vi按以下原则置初值：

$$D[i]=\begin{cases} 0 & i=s \\ W_{si} & i\neq s\text{且}<v_s,v_i>\in E,\ w_{si}\text{为弧上的权值} \\ \infty & i\neq s\text{且}<v_s,v_i>\text{不属于}E \end{cases}$$

（2）选择一个顶点V_j，使得：

$$D[j]=Min\{D[i]\mid V_i\in V-S\}$$

V_j就是求得的下一条最短路径终点，令$S=S\cup\{V_j\}$。

（3）对V-S中的每个顶点V_k，修改D[k]，方法是：

若$D[j]+W_{jk}<D[k]$，则修改为：

$$D[k]=D[j]+W_{jk}\ (\forall V_k\in V-S)$$

（4）重复操作(2)，(3)共n-1次，直到S=V为止。

算法实现

用带权的邻接矩阵表示有向图，

- 设数组D[i]保存从vs到顶点vi的当前最短路径。
- 设数组P[n]保存从Vs到其它顶点的最短路径。若p[i]=k，表示从Vs到Vi的最短路径中，Vi的前一个顶点是Vk，即最短路径序列是(Vs , …, Vk , Vi) 。
- 设数组final[n]，标识一个顶点是否已加入S中。

```
BOOLEAN  final[MAX_VEX] ;
int  P[MAX_VEX] , D[MAX_VEX] ;
void Dijkstra_path (MGraph G, int v0)
 /*   从图G中的顶点v0出发到其余各顶点的最短路径    */
    {  int j, k, m, min ;
        for ( j=0; j<G.vexnum; j++)
        { P[j]=v0 ;  final[j]=0 ;D[j]=G->adj[v][j] ;}  /*  各数组的初始化  */
          D[v0]=0 ; final[v0]=1 ;      /*  设置S={v}  */
        for ( j=1; j<G.vexnum; j++) /*  其余n-1个顶点  */
            {
```

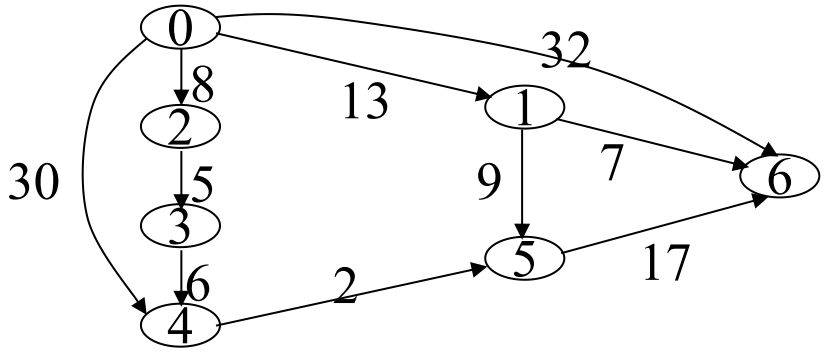
```
        min=INFINITY ;
        for ( w=0; w<G.vexnum; w++)
           {  if  (!final[w]&&D[w]<min)
                  {   min=D[w] ;  v=w; }
           }     /* 求出当前最小的D[v]值  */
        final[v]=1;      /*  将第v个顶点并入S中  */
        for ( w=0; w<G.vexnum; w++)
           { if (!final[w]&&(D[v]+G.arcs[v][w]<D[w]))
                 {   D[w]=D[v]+G.arcs[v][w] ;
                     P[w]=v;
                 }
            }     /* 修改D和P数组的值  */
    }       /* 找到最短路径  */
}
```

算法分析

Dijkstra算法的主要执行是：

- **数组变量的初始化：时间复杂度是O(n)；**
- **二重循环：时间复杂度是$O(n^2)$；**

因此，整个算法的时间复杂度是$O(n^2)$。

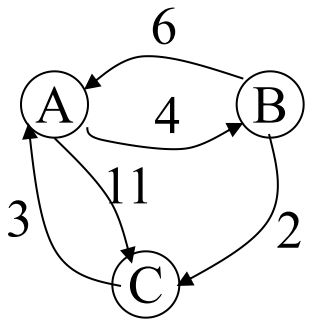


终点	从V0到各终点的最短路径及其长度				
V1	13 <V0,V1>	13 <V0,V1>	-------	-------	-------
V2	8 <V0,V2>	-------	-------	-------	-------
V3	∝	13 <V0,V2,V3>	13 <V0,V2,V3>	-------	-------
V4	30 <V0,V4>	30 <V0,V4>	30 <V0,V4>	19 <V0,V2,V3,V4>	-------
V5	∝	∝	22 <V0,V1,V5>	22 <V0,V1,V5>	21 <V0,V2,V3,V4,V5>
V6	32 <V0,V6>	32 <V0,V6>	20 <V0,V1,V6>	20 <V0,V1,V6>	20 <V0,V1,V6>
Vj	V2:8 <V0,V2>	V1:13 <V0,V1>	V3:13 <V0,V2,V3>	V4:19 <V0,V2,V3,V4>	V6:20 <V0, V1,V6>

每一对顶点之间的最短路径

- 方法一：每次以一个顶点为源点，重复执行Dijkstra算法n次—— $T(n)=O(n^3)$
- 方法二：弗洛伊德(Floyd)算法
  - 算法思想：逐个顶点试探法
  - 求最短路径步骤
    - 初始时设置一个n阶方阵，令其对角线元素为0，若存在弧<Vi,Vj>，则对应元素为权值；否则为∞
    - 逐步试着在原直接路径中增加中间顶点，若加入中间点后路径变短，则修改之；否则，维持原值
    - 所有顶点试探完毕，算法结束

例



初始：$\begin{pmatrix} 0 & 4 & 11 \\ 6 & 0 & 2 \\ 3 & \infty & 0 \end{pmatrix}$ 路径：

	AB	AC
BA		BC
CA		

加入A：$\begin{pmatrix} 0 & 4 & 11 \\ 6 & 0 & 2 \\ 3 & 7 & 0 \end{pmatrix}$ 路径：

	AB	AC
BA		BC
CA	CAB	

加入B：$\begin{pmatrix} 0 & 4 & 6 \\ 6 & 0 & 2 \\ 3 & 7 & 0 \end{pmatrix}$ 路径：

	AB	ABC
BA		BC
CA	CAB	

加入C：$\begin{pmatrix} 0 & 4 & 6 \\ 5 & 0 & 2 \\ 3 & 7 & 0 \end{pmatrix}$ 路径：

	AB	ABC
BCA		BC
CA	CAB	

- 算法实现
  - 图用邻接矩阵存储
  - 定义一个n阶方阵序列

$D^{(-1)}, D^{(0)}, D^{(1)}, \ldots, D^{(k)}, \ldots, D^{(n-1)}$

其中$D^{(-1)}[i][j]=G.arcs[i][j]$

$D^{(k)}[i][j]=Min\{D^{(k-1)}[i][j], D^{(k-1)}[i][k]+D^{(k-1)}[k][j]\}$　　$k\in[0,n-1]$

$D^{(1)}[i][j]$是从vi到vj的中间顶点序号不大于1的最短路径长度。

$D^{(k)}[i][j]$是从vi到vj的中间顶点序号不大于k的最短路径长度。

$D^{(n-1)}[i][j]$是从vi到vj的最短路径长度。

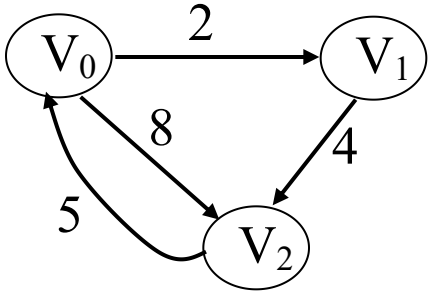
◆ 定义二维数组Path[n][n](n为图的顶点数)，元素Path[i][j]保存从V_i到V_j的最短路径所经过的顶点。

◆ 若Path[i][j]=k：从V_i到V_j经过V_k，最短路径序列是(V_i，…，V_k，…，V_j)，则路径子序列：(V_i，…，V_k)和(V_k，…，V_j)一定是从V_i到V_k和从V_k到V_j的最短路径。从而可以根据Path[i][k]和Path[k][j]的值再找到该路径上所经过的其它顶点，…依此类推。

◆初始化为Path[i][j]=-1，表示从V_i到V_j不经过任何。当某个顶点V_k加入后使D[i][j]变小时，令Path[i][j]=k。

用Floyd算法求任意一对顶点间最短路径

步骤	初态	k=0	K=1	K=2
D	$\begin{pmatrix} 0 & 2 & 8 \\ \infty & 0 & 4 \\ 5 & \infty & 0 \end{pmatrix}$	$\begin{pmatrix} 0 & 2 & 8 \\ \infty & 0 & 4 \\ 5 & 7 & 0 \end{pmatrix}$	$\begin{pmatrix} 0 & 2 & 6 \\ \infty & 0 & 4 \\ 5 & 7 & 0 \end{pmatrix}$	$\begin{pmatrix} 0 & 2 & 6 \\ 9 & 0 & 4 \\ 5 & 7 & 0 \end{pmatrix}$
Path	$\begin{pmatrix} -1 & -1 & -1 \\ -1 & -1 & -1 \\ -1 & -1 & -1 \end{pmatrix}$	$\begin{pmatrix} -1 & -1 & -1 \\ -1 & -1 & -1 \\ -1 & 0 & -1 \end{pmatrix}$	$\begin{pmatrix} -1 & -1 & 1 \\ -1 & -1 & -1 \\ -1 & 0 & -1 \end{pmatrix}$	$\begin{pmatrix} -1 & -1 & 1 \\ 2 & -1 & -1 \\ -1 & 0 & -1 \end{pmatrix}$
S	{ }	{ 0 }	{ 0, 1 }	{ 0, 1, 2 }



$$\begin{pmatrix} 0 & 2 & 8 \\ \infty & 0 & 4 \\ 5 & \infty & 0 \end{pmatrix}$$

带权有向图及其邻接矩阵

根据上述过程中Path[i][j]数组，得出：

V_0到V_1：最短路径是{ 0, 1 }，路径长度是2；

V_0到V_2：最短路径是{ 0, 1, 2 }，路径长度是6；

V_1到V_0：最短路径是{ 1, 2, 0 }，路径长度是9；

V_1到V_2：最短路径是{ 1, 2 }，路径长度是4；

V_2到V_0：最短路径是{ 2, 0 }，路径长度是5；

V_2到V_1：最短路径是{ 2, 0, 1 }，路径长度是7；

算法实现

```
int  A[MAX_VEX][MAX_VEX] ;
int  Path[MAX_VEX][MAX_VEX] ;
void Floyd_path (MGraph G)
   {  int j, k, m ;
      for ( j=0; j<G->vexnum; j++)
         for ( k=0; k<G->vexnum; k++)
            {  D[j][k]=G.arcs[j][k] ; Path[j][k]=-1 ;  }
                  /*  各数组的初始化  */
```

```
for ( m=0; m<G.vexnum; m++)
    for ( j=0; j<G.vexnum; j++)
        for ( k=0; k<G.vexnum; k++)
          if ((D[j][m]+D[m][k])<D[j][k])
              {  D[j][k]=D[j][m]+D[m][k] ;
                 Path[j][k]=m;
              }   /*  修改数组D和Path的元素值  */
for ( j=0; j<G.vexnum; j++)
    for ( k=0; k<G.vexnum; k++)
        if (j!=k)
           {  printf("%d到%d的最短路径为:\n", j, k) ;
              printf("%d  ", j) ; prn_pass(j, k) ;
              printf("%d  ", k) ;
```

```
                printf("最短路径长度为: %d\n",D[j][k]) ;
            }
    }       /*   end of Floyd  */


void  prn_pass(int j , int k)
    {   if (Path[j][k]!=-1)
            {  prn_pass(j, Path[j][k]) ;
               printf(", %d" , Path[j][k]) ;
               prn_pass(Path[j][k],  k) ;
            }
    }
```