第八章 图

现实世界中，事物之间的关系是错综复杂的，最简单的是线性关系，稍微复杂一些的是树关系。

本章学习更为复杂的一种数据结构——图结构。特征是：每个元素可以有多个前驱、多个后继。

- 图结构ADT的定义
- 图结构ADT的实现
  - 图的存储
  - 图的重要操作
- 图的典型应用
  - 最小连通代价问题
  - 最短路径问题
  - 图的工程应用问题（AOV、AOE）

LOGO

8.1 图结构ADT的定义

8.1.1 图结构的逻辑定义

1. 图逻辑结构

[图 Graph]：图是一种数据结构，它由顶点(Vertex)集合（即数据元素）及顶点间的边(Edge) 集合（即元素之间的关系）组成。

Graph＝(D,R)=(*V*, *E*)

其中　$V = \{ x \mid x \in D_0 \}$ ，即V是顶点的有穷非空集合；

$E = \{(x, y) \mid x, y \in V \}$

或　$E = \{<x, y> \mid x, y \in V \ \&\& \ Path\ (x, y)\}$，

即E是顶点之间关系的有穷集合，也叫做边集合。

Path (*x*, *y*)表示从 *x* 到 *y* 的一条单向通路, 它是有方向的。

8.1 图结构ADT的定义

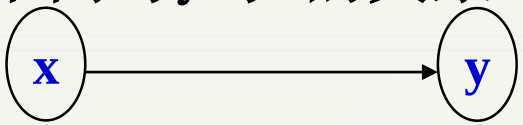
8.1.1 图结构的逻辑定义

1. 图逻辑结构

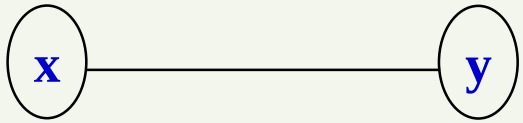
[顶点 Vertex] 数据元素;

[边 Edge] 数据元素之间的关系;

[有向边] 数据元素之间的关系有序，
即x与y的关系不同与y与x的关系，用尖括号表示 <x, y>≠<y, x>



[无向边] 数据元素之间的关无有序，
即x与y的关系等价与y与x的关系，(x, y)=(y, x)



[权 Weight] 图赋予了一种含义，边具有一个关联的数值，这个数值称为权。（带权图、不带权图）

8.1 图结构ADT的定义

8.1.1 图结构的逻辑定义

1. 图逻辑结构

(1)无向图(Undirected Graph or Undigraph)

(2)有向图(Directed Graph or Digraph)

(3)稀疏图(Sparse Graph)

(4)稠密图(Dense Graph)

(5)简单图：若图满足：任一边(u,v) 或<u,v> 有u≠v，即自己不能与自己有关系； 一条边不允许重复出现，即两个元素不能有相同的多个关系；

8.1 图结构ADT的定义

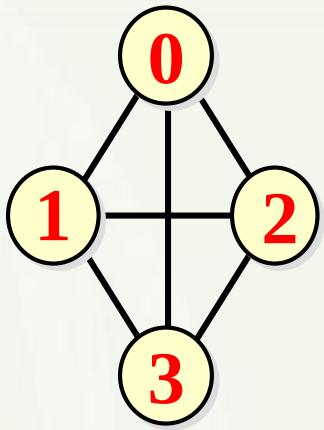
8.1.1 图结构的逻辑定义

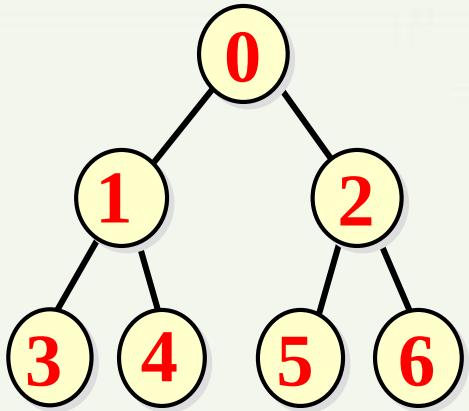
1. 图逻辑结构

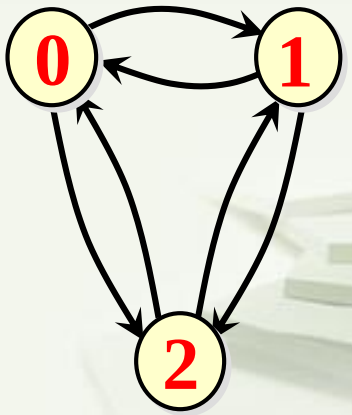
(6)完全图(Conplete Graph) ：包括所有可能边的简单图，即具有最大边数的简单图。

有 n 个顶点的无向完全图有 $n(n-1)/2$ 条边。

有 n 个顶点的有向完全图有 $n(n-1)$ 条边。







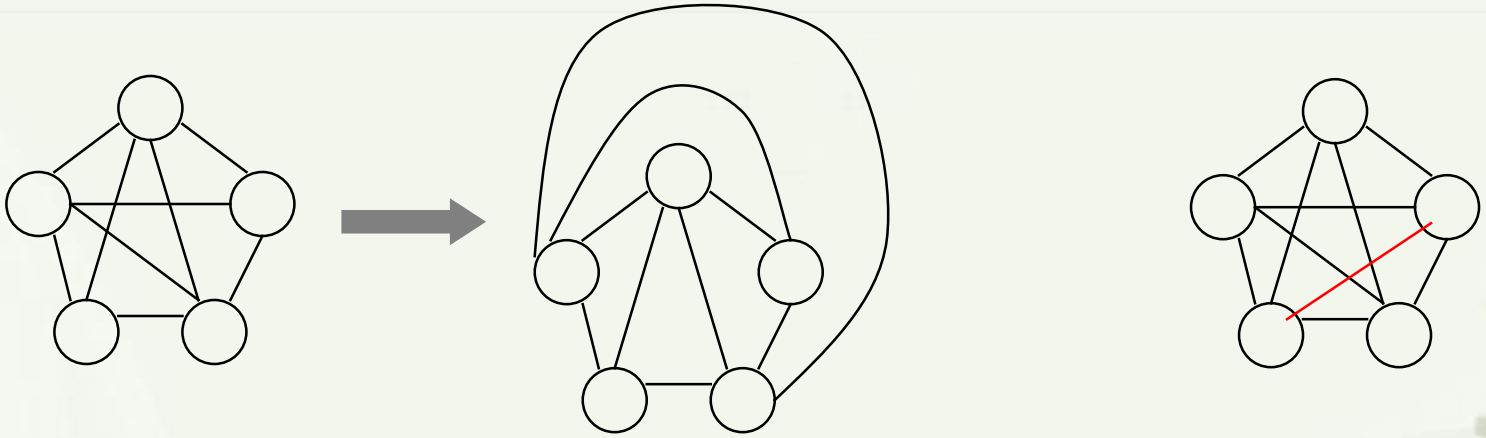
8.1 图结构ADT的定义

8.1.1 图结构的逻辑定义

1. 图逻辑结构

(7) 平面图：存在一种画法，使各条边仅在顶点处相交（关系能够表示清楚）；

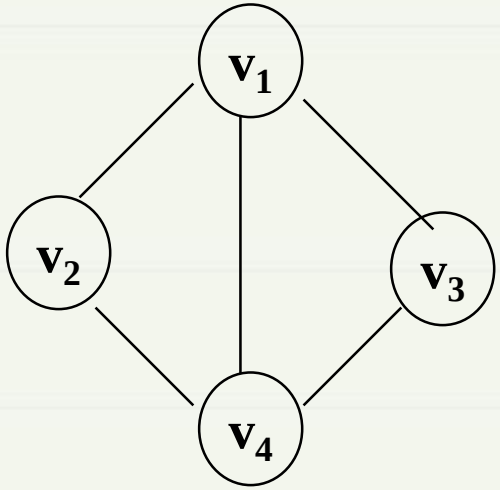
(8) 非平面图：无论怎样画，都有边在非顶点处相交。



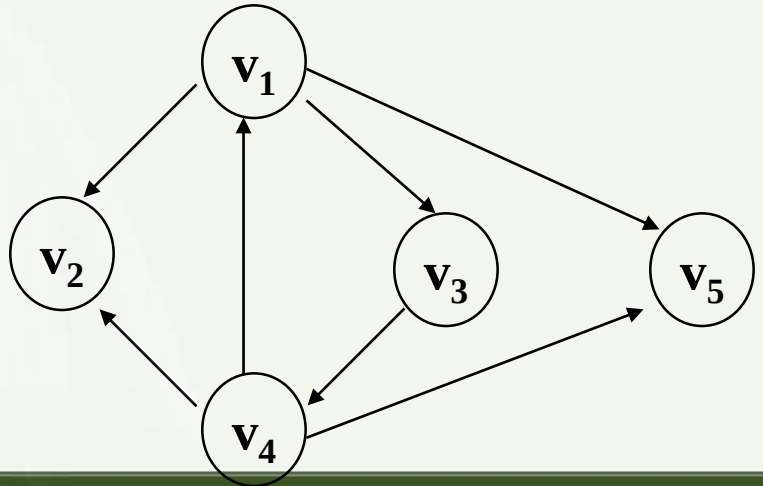
8.1 图结构ADT的定义

8.1.1 图结构的逻辑定义

1. 图逻辑结构



$G_1=(D,R)=(V,R)=(V,E)$
$V=\{ v_1,v_2,v_3,v_4 \}$
$R=\{ (v_1.v_2), (v_1,v_3), (v_1,v_4), (v_2,v_4), (v_3,v_4) \}$



$G_1=(D,R)=(V,R)=(V,E)$
$V=\{ v_1,v_2,v_3,v_4,v_5 \}$
$R=\{ <v_1,v_2>, <v_1,v_3>, <v_1,v_5>,$
$<v_3,v_4>, <v_4,v_1>, <v_4,v_2>,$
$<v_4,v_5> \}$

8.1 图结构ADT的定义

8.1.1 图结构的逻辑定义

2. 有关术语

（1）邻接：无向图，若(u, v)∈E，则称u, v相互邻接；
有向图，若<u, v> ∈E，则称u邻接到v，
或v邻接于u；

（2）关联（依附）：若(u, v)∈E 或 <u, v> ∈E ，则称边依附于顶点u, v 或顶点u, v与边相关联；

（3）顶点的度(Degree)：与顶点相关联的边的数目，
记作Td(v)；

入度：与顶点相关联的引入边的数目，记作Id(v)；
出度：与顶点相关联的出入边的数目，记作Od(v)；

8.1 图结构ADT的定义

8.1.1 图结构的逻辑定义

2. 有关术语

Td(v)=Id(v)+Od(v)

假设图的边数为e，顶点数为n ，则：

$$2e=\sum_{i=1}^{n} Td(v_i)$$

（4）路径：在图 G＝(*V*, *E*) 中, 若从顶点 v_i 出发, 沿一些边经过一些顶点 $v_{p1}, v_{p2}, \ldots, v_{pm}$， 到达顶点$v_j$。则称顶点序列 ($v_i$ v_{p1} v_{p2} ... v_{pm} v_j) 为从顶点v_i 到顶点 v_j 的路径。它经过的边(v_i, v_{p1})、(v_{p1}, v_{p2})、...、(v_{pm}, v_j) 应是属于*E*的边。（有向、无向）

8.1 图结构ADT的定义

8.1.1 图结构的逻辑定义

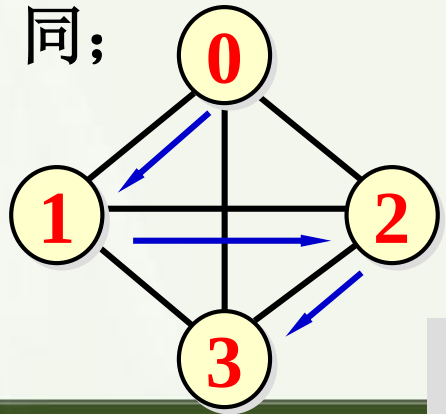
2. 有关术语

路径长度： 带权路径长度、非带权路径长度

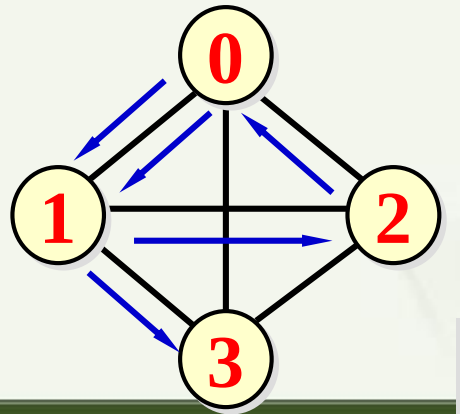
简单路径： 若路径上各顶点 $v_1, v_2, ..., v_m$ 均不 互相重复, 则称这样的路径为简单路径。

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： 路径上除起点与终点相同外，其余顶点都不相同；



0-1-2-3



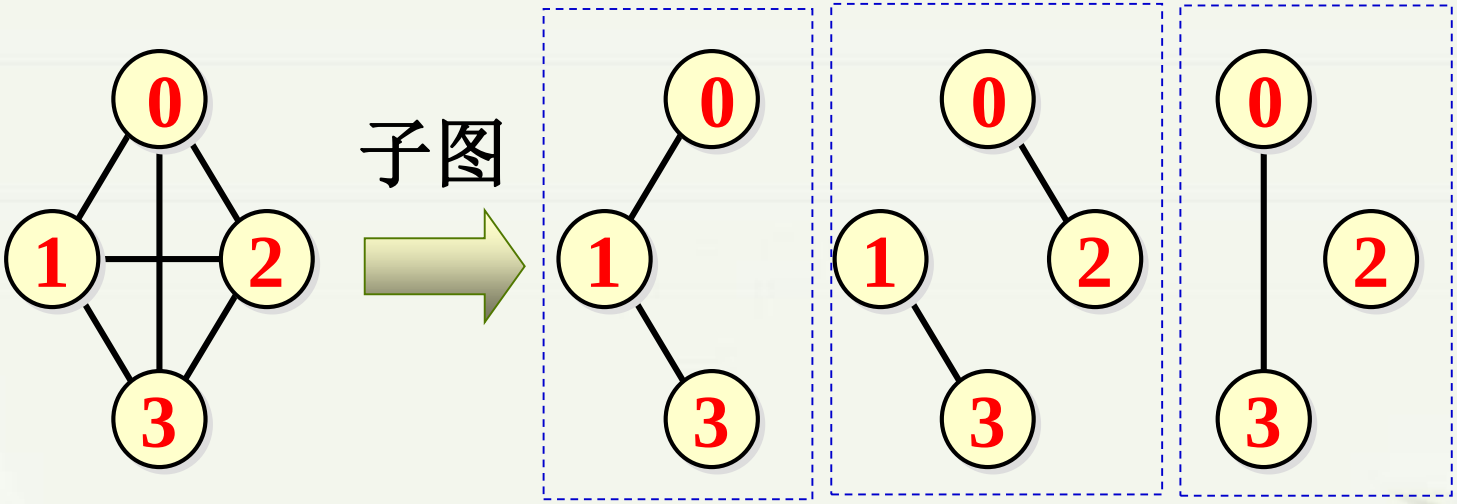
0-1-2-0-1-3

8.1 图结构ADT的定义

8.1.1 图结构的逻辑定义

2. 有关术语

（5）子图：设有两个图G＝(*V*, *E*) 和G'＝(*V*', *E*')。若*V* '⊆ *V* 且*E*'⊆*E*, 则称图G'是图G的子图。



一个图的子图包括其自身！

8.1 图结构ADT的定义

8.1.1 图结构的逻辑定义

2. 有关术语

（6）顶点连通：（无向图）两个顶点v_i，v_j之间有路径；

顶点强连通：（有向图）两个顶点v_i到v_j，v_j到v_i之间都有有向路径；

连通图：如果图中任意一对顶点都是连通的，则称此图是连通图。

连通分量：非连通图的极大连通子图。

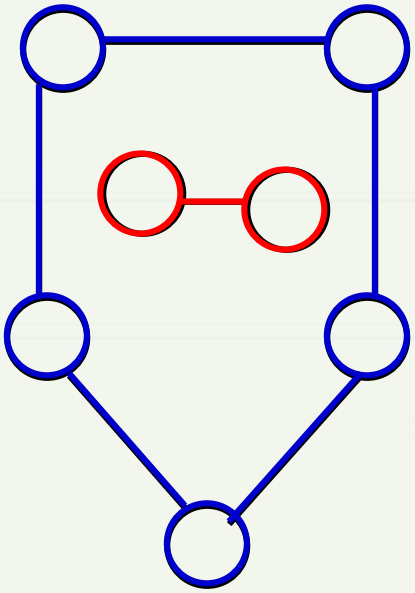
强连通图：在有向图中，若对于每一对顶点v_i和v_j，都存在一条从v_i到v_j和从v_j到v_i的路径，则称此图是强连通图。

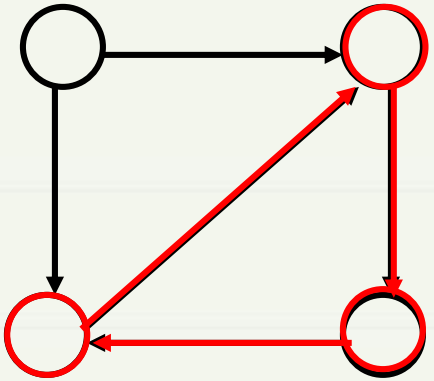
强连通分量：非强连通图的极大强连通子图。

8.1 图结构ADT的定义

8.1.1 图结构的逻辑定义

2. 有关术语





连通图只有一个连通分量；
强连通图只有一个强连通分量；

8.1 图结构ADT的定义

8.1.1 图结构的逻辑定义

2. 有关术语

（7）生成树：连通图的极小连通子图，它包含图的所有n个顶点，n-1条边；

有向树：一个有向图恰有一个入度0为的顶点，其余顶点的入度均为1。

生成树森林：无向图，各个连通分量的生成树；
有向图，由若干有向树组成，含有图中全部顶点，但只有足以构成若干棵不相交的有向树的边；

8.1 图结构ADT的定义

8.1.2 图结构上定义的操作

图初始化	Init_Graph(g)
求顶点在图中的位置	**Loc_vertex(g,v)**
访问图的顶点	**Get_vertex(g,i)**
求图中**v**的第一个邻接点	**First_adj(g,v)**
求图中**v**的**w**后的下一个邻接点	**Next_adj(g,v,w)**
插入顶点	**Ins_vertex(g,u)**
插入边	**Ins_edge(g,u1,u2)**
删除顶点	**Del_vertex(g,u)**
删除边	**Del_edge(g,u1,u2)**
图的遍历	**Traversal_g(g,u)**
……	……

8.1 图结构ADT的定义

8.1.3 图的ADT定义

```
ADT Graph
{  数据结构：Graph=(D,R)
              D= {具有相同性质的数据元素的集合};
              R= {<u,v>|(u,v∈D)}。
   操作：
      Init_Graph(g)
      Loc_vertex(g,v)
      Get_vertex(g,i)
      First_adj(g,v)
      Next_adj(g,v,w)
      Ins_vertex(g,u)
      Ins_edge(g,u1,u2)
      Del_vertex(g,u)
      Del_edge(g,u1,u2)
      IsAdjacent(g,u,v)
      DFS_Traverse(g,v;
      BFS_Traverse(g,v);
}
```

8.2 图结构ADT的实现

随着数据结构的复杂，其关系的存储（表示）也越来越麻烦，特别是顺序存储方式是靠物理上的相邻来表示逻辑关系的，表示关系的能力很弱。因此，面对图这种复杂逻辑结构，顺序存储方式已经无能为力了！

所以，图结构没有顺序存储方式。

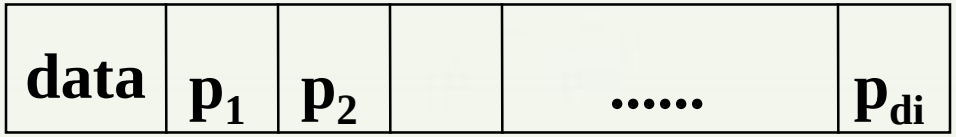
而链式存储方式是靠存储相关元素的地址（指针）来表示关系的，表示关系的能力很强。应该说可以存储任意复杂的逻辑关系。所以，对图结构来说，最容易想到的就是链式存储结构，即在存储数据元素的同时，用指针表示它们之间的关系！

8.2 图结构ADT的实现

定长结点结构:

d是图的度，即顶点度的最大值；

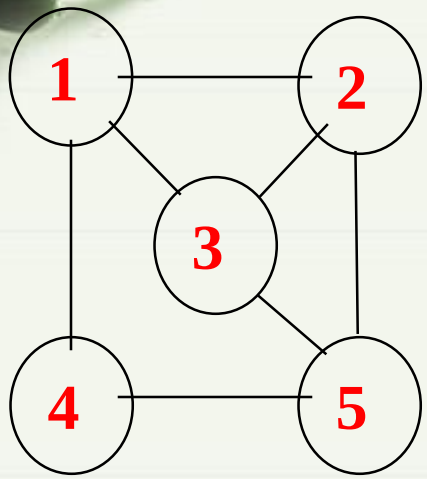
不定长结点结构:

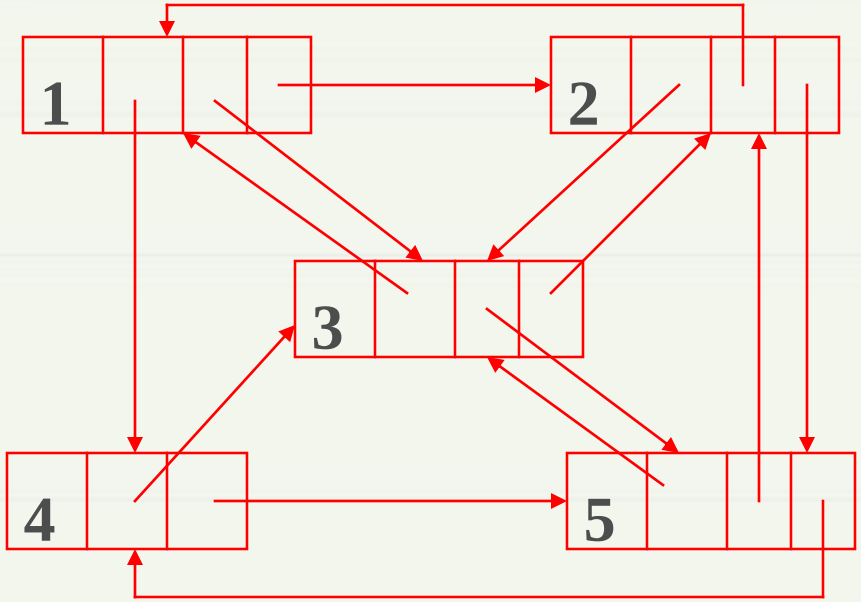


d_i是顶点的度；

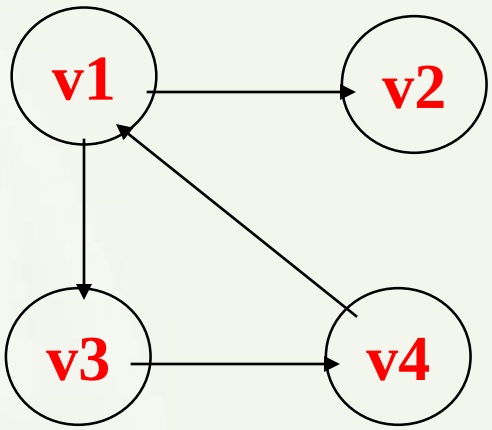
注意：有向图一般是指的出度！

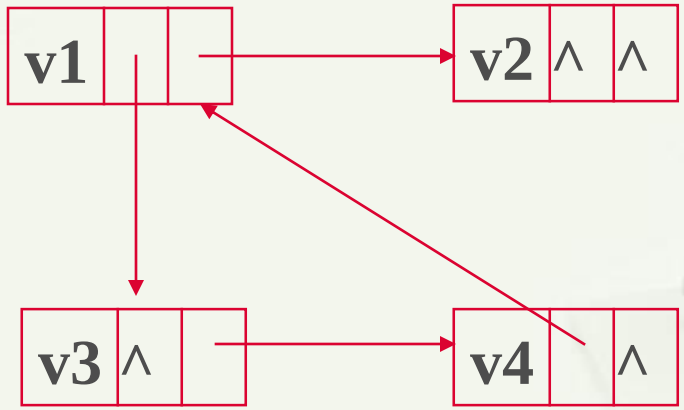
8.2 图结构ADT的实现





不定长结点





定长结点

8.2 图结构ADT的实现

由于图结构的复杂，其顺序存储方式不存在，而一般的链式存储又存在一些缺点。为此，提出了专门针对图结构的一些存储方式。这些存储方式的基本原则是：

（1） 存储数据元素（一般采用顺序存储方式）

（2） 存储（表示）数据之间的关系

邻接矩阵
关联矩阵
} 用二维表表示关系

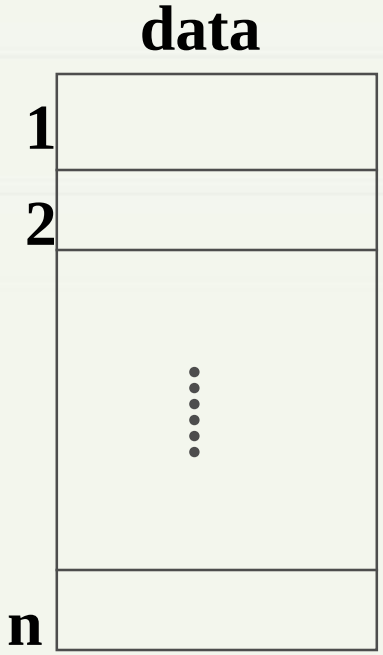
邻接表
十字链表
邻接多重表
} 用指针表示关系

8.2 图结构ADT的实现

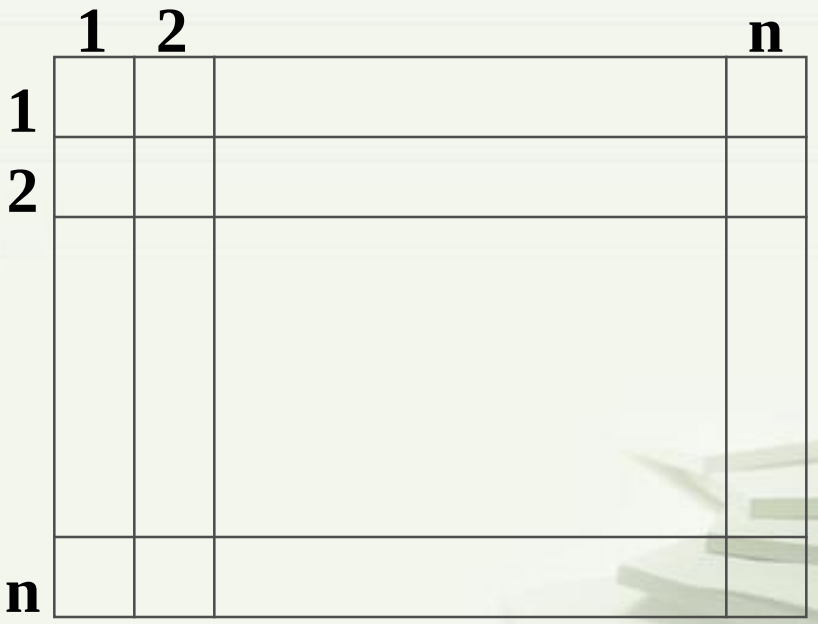
8.2.1 基于邻接矩阵存储结构的实现

8.2.1.1 存储结构

存储方式：用连续的地址空间存储图的数据元素及元素之间的关系(邻接关系)；



存储元素



表示邻接关系

8.2 图结构ADT的实现

8.2.1 基于邻接矩阵存储结构的实现

8.2.1.1 存储结构

VerticesList： 一维数组，存放数据元素（顶点）

Edge： 二维数组，存放数据之间的关系

$$Edge[i][j]=\begin{cases}1, & \text{如果} <v_i,v_j> \in E \text{ 或者 } (v_i,v_j)\in E \\ 0, & \text{否则}\end{cases}$$

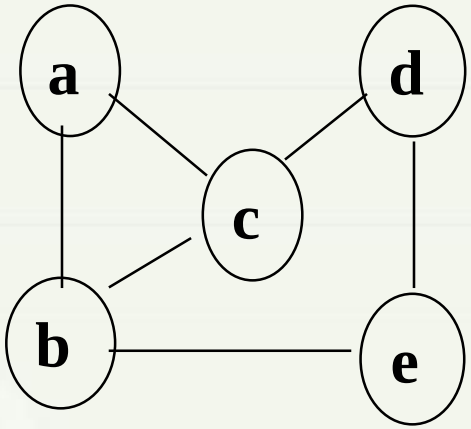
$$Edge[i][j]=\begin{cases}W_{ij}, & \text{若} v_i \neq v_j \text{且} <v_i,v_j> \in E \text{或} (v_i,v_j)\in E \\ \infty, & \text{若} v_i \neq v_j \text{且} <v_i,v_j> \notin E \text{或} (v_i,v_j)\notin E \\ 0, & \text{若} v_i == v_j\end{cases}$$

8.2 图结构ADT的实现

8.2.1 基于邻接矩阵存储结构的实现

8.2.1.1 存储结构

举例：



[无向图]

1	a
2	b
3	c
4	d
5	e

存储数据元素

	1	2	3	4	5
1	0	1	1	0	0
2	1	0	1	0	1
3	1	1	0	1	0
4	0	0	1	0	1
5	0	1	0	1	0

存储数据的邻接关系

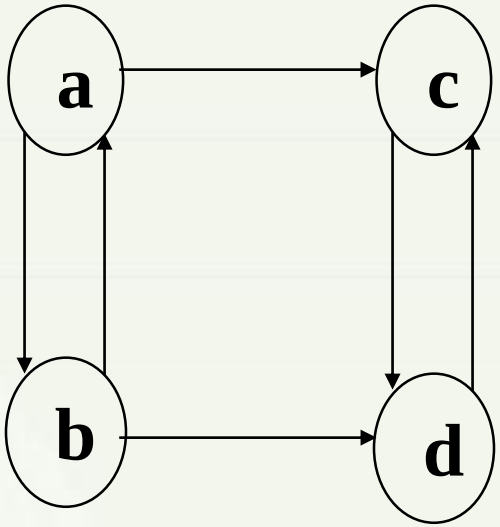
1	d
2	e
3	a
4	c
5	b

	1	2	3	4	5
1	0	1	0	1	0
2	1	0	0	0	1
3	0	0	0	1	1
4	1	0	1	0	1
5	0	1	1	1	0

8.2 图结构ADT的实现

8.2.1 基于邻接矩阵存储结构的实现

8.2.1.1 存储结构



[有向图]

1	a
2	c
3	b
4	d

存储数据元素

	1	2	3	4
1	0	1	1	0
2	0	0	0	1
3	1	0	0	1
4	0	1	0	0

存储数据的邻接关系

8.2 图结构ADT的实现

8.2.1 基于邻接矩阵存储结构的实现

8.2.1.1 存储结构

对于无向图（非加权）：

（1）矩阵是对称的;

（2）第i行或第i 列1的个数为顶点v_i 的度;

（3）矩阵中1的个数的一半为图中边的数目;

（4）很容易判断顶点v_i 和顶点v_j之间是否有边相连;

特点：

加权图如何？

对于有向图（非加权）：

（1） 矩阵不一定是对称的;

（2） 第i 行中1的个数为顶点v_i 的出度;

（3） 第i列中1的个数为顶点 v_i的入度;

（4） 矩阵中1的个数为图中弧的数目;

（5） 很容易判断顶点v_i 和顶点v_j 是否有弧相连;

8.2 图结构ADT的实现

8.2.1 基于邻接矩阵存储结构的实现

8.2.1.1 存储结构

高级语言实现：

```
#define  Max_VertexNum  100   //允许图的顶点个数的最大值
typedef  数据元素类型  VertexType;     //例如  char
typedef  边类型  Adjmatrix;    //例如 int
typedef enum{DG,DN,AG,AN} GraphKind;
struct Graph
 {  GraphKind  Kind;     //图的类型
    int VexNum;    //顶点（元素）个数
    VertexType VerticesList[Max_VertexNum ];  //存储元素
    Adjmatrix  Edges[Max_VertexNum, Max_VertexNum ];  //表示关系
 }
```

8.2 图结构ADT的实现

8.2.1 基于邻接矩阵存储结构的实现

8.2.1.2 类定义及部分成员函数（操作）实现

类定义：

```
typedef  边类型  E;          //边的类型（权值）
typedef  元素类型  T;        //元素类型（顶点）
class Graphmtx
{  friend istream& operator >> ( istream& in,   Graphmtx & G);   //输入
   friend ostream& operator << (ostream& out, Graphmtx & G);  //输出
   private:
      int maxVertices;     //允许的图的顶点个数的最大值
      int numVertices;    //图的顶点数
      int numEdges;       //图的边数
      T *VerticesList;                 //一维数组，存放元素（顶点）
      E **Edge;                        //二维数组，存放邻接矩阵（关系）
```

8.2 图结构ADT的实现

8.2.1 基于邻接矩阵存储结构的实现

8.2.1.2 类定义及部分成员函数（操作）实现

```
    int getVertexPos (T vertex)  //确定元素在图中的存储位置
        { for (int i = 0; i < numVertices; i++)
              if (VerticesList[i] == vertex) return i;
          return -1;  };
public:
    Graphmtx (int sz = DefaultVertices);    //构造函数
    ～Graphmtx ()                                    //析构函数
        { delete [ ]VerticesList;  delete [ ]Edge; }
    T getValue (int i)  //取顶点 i 的值, i 不合理返回0
    { if (i >= 0 && i < numVertices )  return VerticesList[i];
      else { cout<<“位置错！”<<endl;exit(1);}    }
     E getWeight (T v1, T v2)  //取边(v1,v2)上权值
     {  i= getVertexPos (v1);   j= getVertexPos (v2);
        return (i != -1 && j != -1 )? Edge[i][j] : 0;  }
```

8.2 图结构ADT的实现

8.2.1 基于邻接矩阵存储结构的实现

8.2.1.2 类定义及部分成员函数（操作）实现

```
    T getFirstNeighbor(T v);          //取顶点 v 的第一个邻接顶点
    T getNextNeighbor(T v, T w); //取 v 的邻接顶点 w 的下一邻接顶点
    bool insertVertex(const T vertex); //插入顶点vertex
    bool insertEdge(T v1, T v2, E cost); //插入边(v1, v2),权值为cost
    bool removeVertex(T v);  //删去顶点 v 和所有与它相关联的边
    bool removeEdge(T v1, T v2);      //在图中删去边(v1,v2)
    BFS_traversal(T v);  //图的广度遍历
    DFS_traversal(T v);  //图的深度遍历
};
```

8.2 图结构ADT的实现

8.2.1 基于邻接矩阵存储结构的实现

8.2.1.2 类定义及部分成员函数（操作）实现

部分成员函数的实现：

（1）构造函数：

```
Graphmtx::Graphmtx (int sz)   //构造函数
{  maxVertices = sz;
   numVertices = 0;  numEdges = 0;
    int i, j;
    VerticesList = new T[maxVertices];    //开辟元素存储空间
    Edge = (int **) new int *[maxVertices]; //开辟关系存储空间—邻接矩阵
    for (i = 0; i < maxVertices; i++)
       Edge[i] = new int[maxVertices];
    for (i = 0; i < maxVertices; i++)        //邻接矩阵初始化
       for (j = 0; j < maxVertices; j++)
            Edge[i][j] = (i == j) ?  0 : maxWeight;
};
```

8.2 图结构ADT的实现

8.2.1 基于邻接矩阵存储结构的实现

8.2.1.2 类定义及部分成员函数（操作）实现

（2）求顶点v的第一个邻接点：

```
T  Graphmtx::getFirstNeighbor (T v)
//有，则返回邻接点，//如果没有，则返回空值
{  i=getVertexPos(v);  //顶点v的存储位置
   if (i != –1)
   {  for (int col = 0; col < numVertices; col++)
         if (Edge[i][col] && Edge[i][col] < maxWeight)
            return getValue(col);   //由存储位置得到元素
    }
    return NULL;    //空元素
};
```

8.2 图结构ADT的实现

8.2.1 基于邻接矩阵存储结构的实现

8.2.1.2 类定义及部分成员函数（操作）实现

（3）求顶点v的w后的下一个邻接点：

```
T  Graphmtx::getNextNeighbor (T v, T w)
{ //如果有，则返回该邻接点，如果没有就返回空值
   i=getVertexPos(v);     //顶点v的存储位置
   j=getVertexPos(w);     //顶点w的存储位置
   if (i!= -1 && j!= -1)
   {   for (int col = j+1; col < numVertices; col++)
         if (Edge[i][col] && Edge[j][col] < maxWeight)
            return getValue(col);  //由存储位置得到元素
    }
    return NULL;
};
```

8.2 图结构ADT的实现

8.2.1 基于邻接矩阵存储结构的实现

8.2.1.2 类定义及部分成员函数（操作）实现

（4）在图中插入顶点（元素）：此时不能把关系带进来

```
bool Graphmtx::insertVertex (const T vertex )
{ //插入顶点 vertex， 存储在一维数组的最后

      if (numVertices==maxVertices)  return false;
      VerticesList[numVertices++] =vertex;
      return ture;
};
```

8.2 图结构ADT的实现

8.2.1 基于邻接矩阵存储结构的实现

8.2.1.2 类定义及部分成员函数（操作）实现

（5）删除顶点（元素）：此时，该顶点关联的边也应该被删除

```
bool Graphmtx::removeVertex (T v )
{ //删顶点及其相关联的边
    k=getVertexPos(v);
   if (k<0||k>=numVertices)
       { cout << "参数v越界出错!" << endl;return false; }
   for(int i = 0; i < numVertices; i++)
      for(int j = 0; j < numVertices; j++)
       if((i == k || j == k) && i != j && Edge[i][j] <maxWeight )
        {   Edge[i][j] =maxWeight;              //删除该边
            Edge[j][i] = maxWeight;
            numEdges --;     }              //边的个数减1
```

8.2 图结构ADT的实现

8.2.1 基于邻接矩阵存储结构的实现

8.2.1.2 类定义及部分成员函数（操作）实现

```
    for (i=k; i<numVertices-1; i++)   //移动，删除邻接矩阵的第k行
        for (j=0; j<numVertices; j++)
          Edge[i][j]=Edge[i+1][j];
    for (i=0;i<numVertices-1; i++)   //移动，删除邻接矩阵的第k列
      for (j=k;j<numVertices-1; j ++)
        Edge[i][j]=Edge[i][j+1];
    for (i=k;i<numVertices-1; i++)    //移动，在顶点数组中删除顶点v
      Verticeslist[i]=Verticeslist[i + 1];
    numVertices --;
    return true;
};
```

8.2 图结构ADT的实现

8.2.1 基于邻接矩阵存储结构的实现

8.2.1.2 类定义及部分成员函数（操作）实现

（6）插入边

```
bool Graphmtx::insertEdge (T v1,T v2,E cost )
{  //插入一条起始顶点为v1、终止顶点为 v2的边
      i=getVertexPos(v1);
      j=getVertexPos(v2);
     if(i<0||i>=numVertices||j<0||j>=numVertices)
     { cout << "参数v1或v2出错!" << endl;return false; }
     Edge[i][j]=cost;
     Edge[j][i]=cost;
     numEdges++;    //边数加1
     return true;
}
```

8.2 图结构ADT的实现

8.2.1 基于邻接矩阵存储结构的实现

8.2.1.2 类定义及部分成员函数（操作）实现

（7）删除边

```
bool Graphmtx::removeEdge (T v1,T v2 )
{  //删除顶点v1与v2之间的边
    i=getVertexPos(v1);
    j=getVertexPos(v2);
    if(i<0||i> =numVertices||j<0||j>=numVertices)
    { cout << "参数v1或v2出错!" << endl; return false;}
    if(Edge[i][j]==maxWeight||i==j)
      { cout << "该边不存在!" << endl;       return false; }
    Edge[i][j] = maxWeight;      Edge[j][i] = maxWeight;
    numEdges--;                          //边的个数减1
    return true;
}
```

8.2 图结构ADT的实现

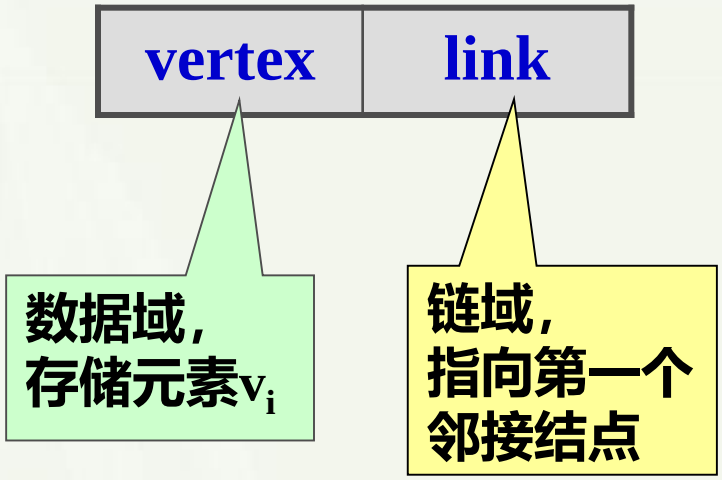
8.2.2 基于邻接表存储结构的实现

8.2.2.1 存储结构

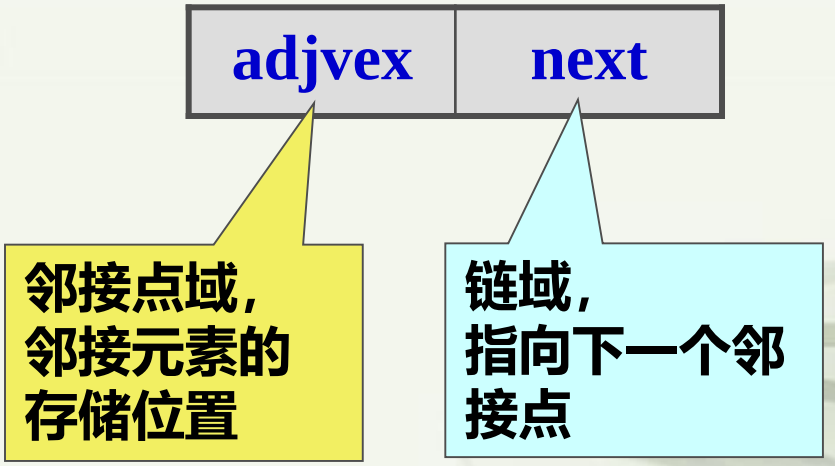
存储方式：

用连续的地址空间存储图的数据元素，用单链表（非顺序空间）存储（表示）元素之间的邻接关系；即把元素与哪些元素有关系通过链表表示出来。

存储元素结点



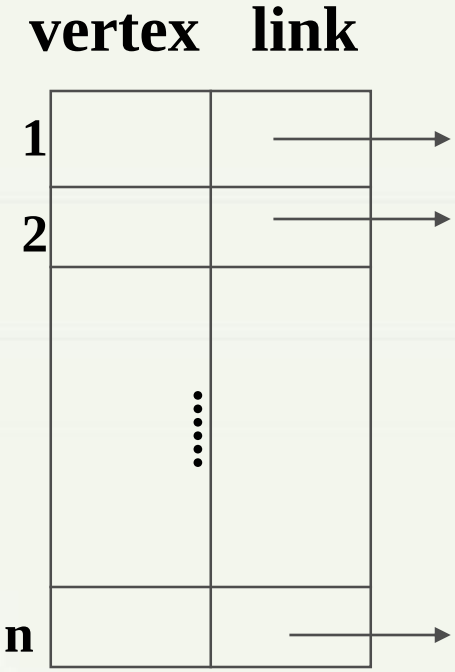
存储邻接关系



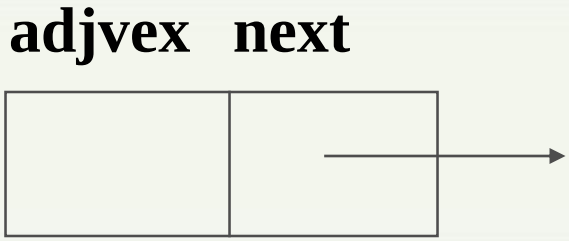
8.2 图结构ADT的实现

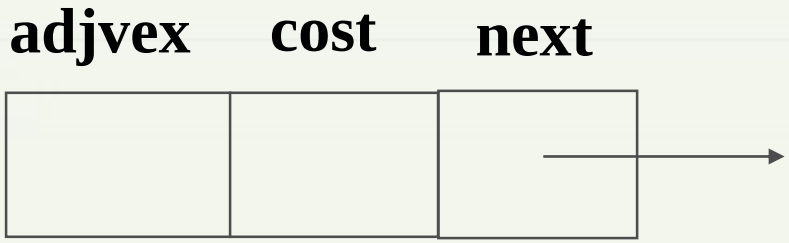
8.2.2 基于邻接表存储结构的实现

8.2.2.1 存储结构



存储元素





存储邻接点信息

8.2 图结构ADT的实现

8.2.2 基于邻接表存储结构的实现

8.2.2.1 存储结构

高级语言实现：

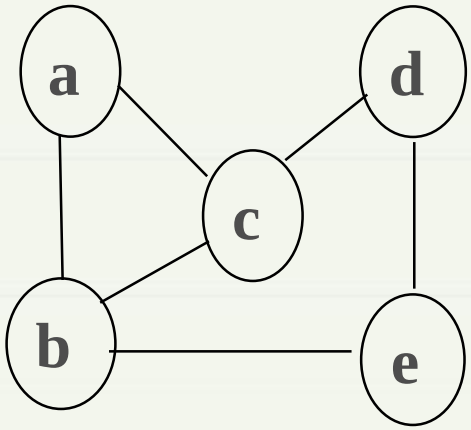
```
const   int   n =maxn;      // maxn表示图中最大顶点数

typedef enum{DG,DN,AG,AN} GraphKind;
struct  Edgenode
  {    int adjvex;                 //邻接点的存储位置
       Edgenode  *next; }      //下邻接点
struct Vertexnode
  {    ElemType vertex;              //数据元素
       Edgenode *link;  }         //第一个邻接点
struct Graph
{  GraphKind  tp;                 //图的类型
   int   vertexnum;               //图的顶点个数
   Vertexnode Nodetable[n];   //一维数组-邻接表
}
```

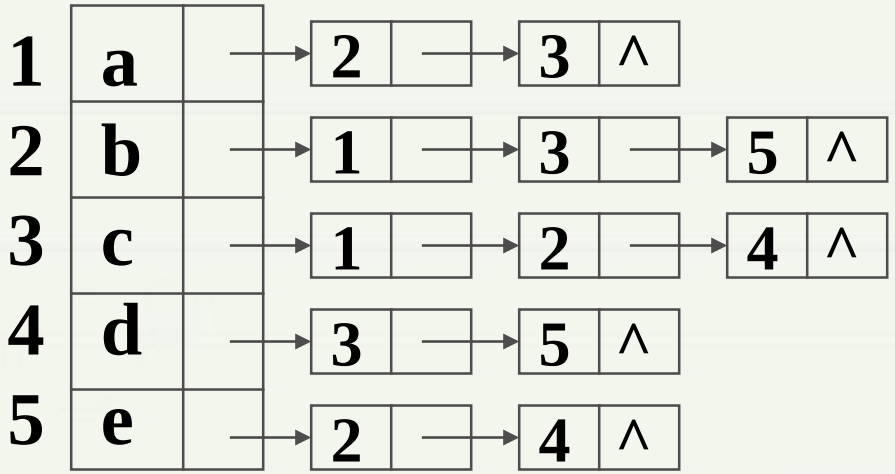
8.2 图结构ADT的实现

8.2.2 基于邻接表存储结构的实现

8.2.2.1 存储结构



[无向图]

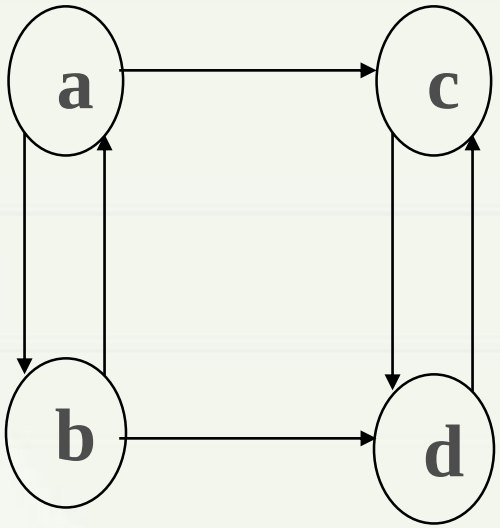


存储元素

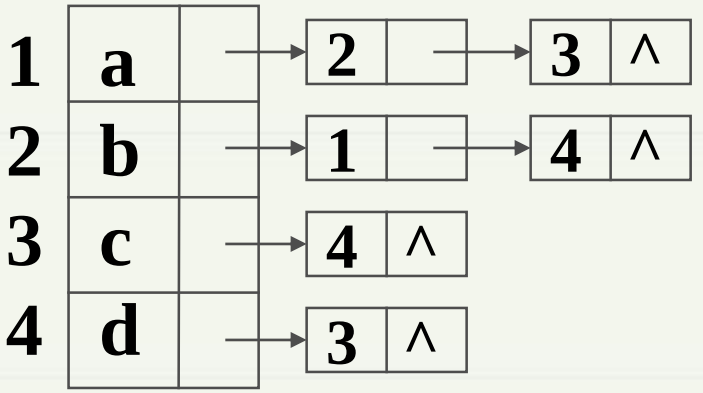
8.2 图结构ADT的实现

8.2.2 基于邻接表存储结构的实现

8.2.2.1 存储结构

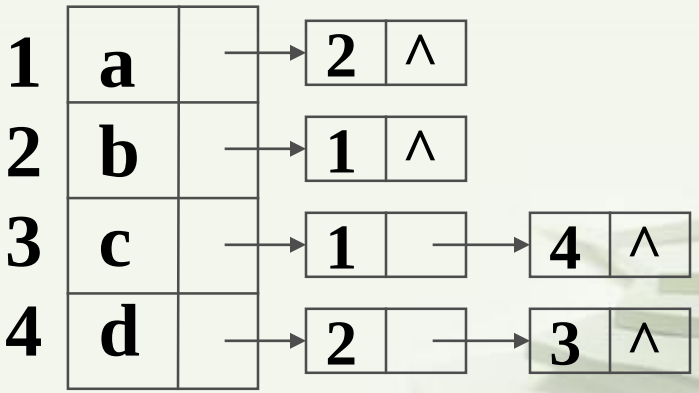


[有向图]



存储元素

逆邻接表

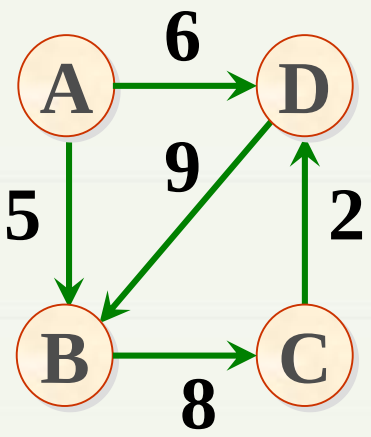


存储元素

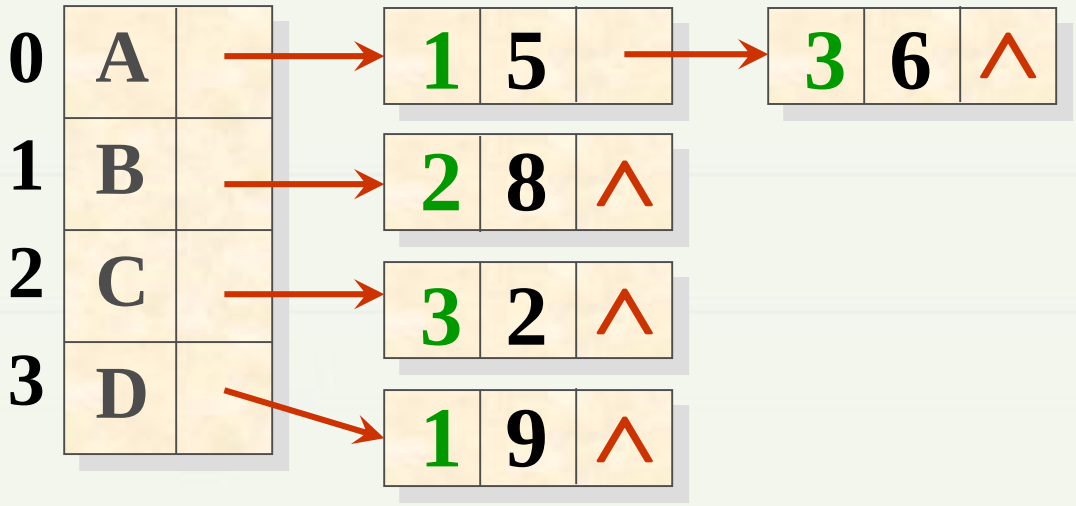
8.2 图结构ADT的实现

8.2.2 基于邻接表存储结构的实现

8.2.2.1 存储结构



有向加权图



8.2 图结构ADT的实现

8.2.2 基于邻接表存储结构的实现

8.2.2.1 存储结构

从无向图的邻接表可以得到如下结论：

（1）第i 个链表中结点数目为顶点v_i的度；
（2）所有链表中结点数目的一半为图中边数；
（3）占用的存储单元数目为n+2e 。

特点：

从有向图的邻接表可以得到如下结论：

（1）第i 个链表中结点数目为顶点v_i的出度；
（2）所有链表中结点数目为图中弧数；
（3）占用的存储单元数目为n+e 。

有时，也建立有向图的逆邻接表，即链接邻接于的顶点！

8.2 图结构ADT的实现

8.2.2 基于邻接矩阵存储结构的实现

8.2.2.2 类定义及部分成员函数（操作）实现

类定义：

```
struct Edge //边结点（邻接点）的定义
{
    int dest;    //边的另一顶点位置
    E cost;              //边上的权值
    Edge *link;                          //下一条边链指针
    Edge () {}                          //构造函数
    Edge (int num, E cost)                  //构造函数
      : dest (num), weight (cost), link (NULL) { }
    bool operator != (Edge& R) const
       { return dest != R.dest; }                        //判边等否
};
```

8.2 图结构ADT的实现

8.2.2 基于邻接矩阵存储结构的实现

8.2.2.2 类定义及部分成员函数（操作）实现

```
struct Vertex  //顶点的定义
{
    T data;            //顶点元素
    Edge  *adj;        //边链表的头指针
};


class Graphlnk  //图的类定义
{
   friend istream& operator >> (istream& in, Graphlnk& G);          //输入
   friend ostream& operator << (ostream& out,Graphlnk & G);  //输出
```

8.2 图结构ADT的实现

8.2.2 基于邻接矩阵存储结构的实现

8.2.2.2 类定义及部分成员函数（操作）实现

```
private:
    Vertex *NodeTable;              //顶点表 (各边链表的头结点)
     int getVertexPos (const T vertx) //给出顶点vertex在图中的位置
    {    for (int i = 0; i < numVertices; i++)
             if (NodeTable[i].data == vertx) return i;
         return -1; }
public:
    Graphlnk (int sz = DefaultVertices);  //构造函数
     ~Graphlnk();
    T getValue (int i)   //第i个存储位置的元素值
    {    return (i >= 0 && i < NumVertices) ?
             NodeTable[i].data : NULL;
    }
```

8.2 图结构ADT的实现

8.2.2 基于邻接矩阵存储结构的实现

8.2.2.2 类定义及部分成员函数（操作）实现

```
    E getWeight (T v1, T v2);              //取边(v1,v2)权值
    bool insertVertex (const T& vertex);
    bool removeVertex (T v);
    bool insertEdge (T v1, T v2, E cost);
    bool removeEdge (T v1, T v2);
    int getFirstNeighbor (T v);
    int getNextNeighbor (T v, T w);
    BFS_traversal(T v);
    DFS_traversal (T v);
}
```

8.2 图结构ADT的实现

8.2.2 基于邻接矩阵存储结构的实现

8.2.2.2 类定义及部分成员函数（操作）实现

部分成员函数的实现：

（1）构造函数：

```
Graphlnk::Graphlnk (int sz) //构造函数：建立一个空的邻接表
{
   maxVertices = sz;
   numVertices = 0;  numEdges = 0;
   NodeTable = new Vertex[maxVertices];  //创建顶点表数组
     if (NodeTable == NULL)
      { cerr << "存储分配错！ " << endl;  exit(1); }
   for (int i = 0; i < maxVertices; i++)
       NodeTable[i].adj = NULL;
};
```

8.2 图结构ADT的实现

8.2.2 基于邻接矩阵存储结构的实现

8.2.2.2 类定义及部分成员函数（操作）实现

（2）析构函数：

```
Graphlnk::～Graphlnk()  //析构函数：删除一个邻接表
{
   for (int i = 0; i < numVertices; i++ )   //把每个单链表空间释放
  {   Edge *p = NodeTable[i].adj;
      while (p != NULL)
      {  NodeTable[i].adj = p->link;
         delete p;  p = NodeTable[i].adj;
      }
    }
    delete [ ]NodeTable;                    //删除顶点表数组
};
```

8.2 图结构ADT的实现

8.2.2 基于邻接矩阵存储结构的实现

8.2.2.2 类定义及部分成员函数（操作）实现

（3）求第一个邻接点：

```
T Graphlnk::getFirstNeighbor (T v)
//给出顶点位置为 v 的第一个邻接顶点的位置,
//如果找不到, 则函数返回-1
{  i= getVertexPos(v);
   if (i!= -1)   //顶点v存在
   {   Edge *p = NodeTable[i].adj; //对应边链表第一个边结点
       if (p != NULL) return getValue(p->dest);
                                   //存在, 返回第一个邻接顶点
    }
    return NULL;
};
```

8.2 图结构ADT的实现

8.2.2 基于邻接矩阵存储结构的实现

8.2.2.2 类定义及部分成员函数（操作）实现

（4）求下一个邻接点：

```
T Graphlnk::getNextNeighbor (T v,  T w)
{//给出顶点v的邻接顶点w的下一个邻接顶点的位置,
 //若没有下一个邻接顶点, 则函数返回-1
   i= getVertexPos(v);
   j= getVertexPos(w);
   if (i!= -1)   //顶点v存在
   {  Edge *p = NodeTable[i].adj;    //指向第一个邻接点
      while (p != NULL && p->dest != j)
         p = p->link;
       if (p != NULL && p->link != NULL)
         return getValue(p->link->dest);    //返回下一个邻接顶点
    }
    return NULL;
}
```

8.3 图的重要操作—遍历

遍历：从已给的连通图中某一顶点出发，沿着一些边访遍图中所有的顶点，且使每个顶点仅被访问一次，就叫做图的遍历，它是图的最重要的基本运算。

遍历图的实质：找每个顶点的邻接点并访问的过程。

注意：图的遍历比树更复杂，因为元素之间的关系复杂。

要考虑两种情况：

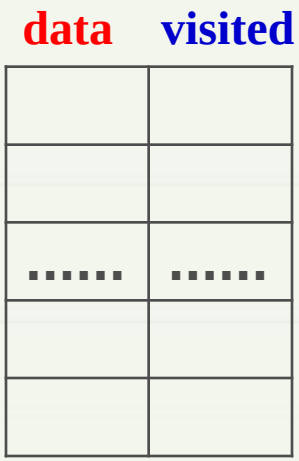
其一：可能会陷入死循环（如图中存在环）

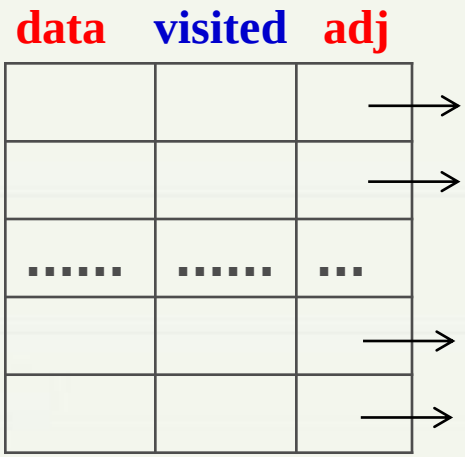
其二：可能有的顶点不能从出发点访问到（如非连通图）

处理的方法是：对每个顶点作一个访问标志！

8.3 图的重要操作—遍历

可设置一个**辅助数组** *visited* [n]，用来标记每个被访问过的顶点。它的初始状态为0，在图的遍历过程中，一旦某一个顶点v_i 被访问，visited[i]置为1，防止它被多次访问。





图的遍历方式：

- 深度优先搜索（Depth_First Search：DFS）
- 广度优先搜索（Breadth_First Search：BFS）

8.3 图的重要操作—遍历

8.3.1 图的深度遍历：DFS

1、遍历方式：假设图为G=（V，E），从v_0开始深度优先遍历图。

访问v_0，作已访问标志；

选择一个与v_0邻接但未访问的顶点u（如果没有，则从v_0开始的深度优先遍历结束；如果有多个，选其中一个）；

从顶点u开始深度优先遍历图；

递归！

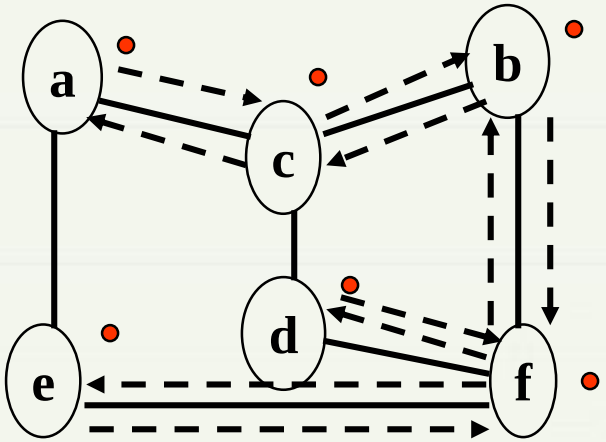
2、特点：类似与树的前序遍历。

沿着图的某一分支访问，直到它的末端，然后回溯。

8.3 图的重要操作—遍历

8.3.1 图的深度遍历：DFS

3、举例：

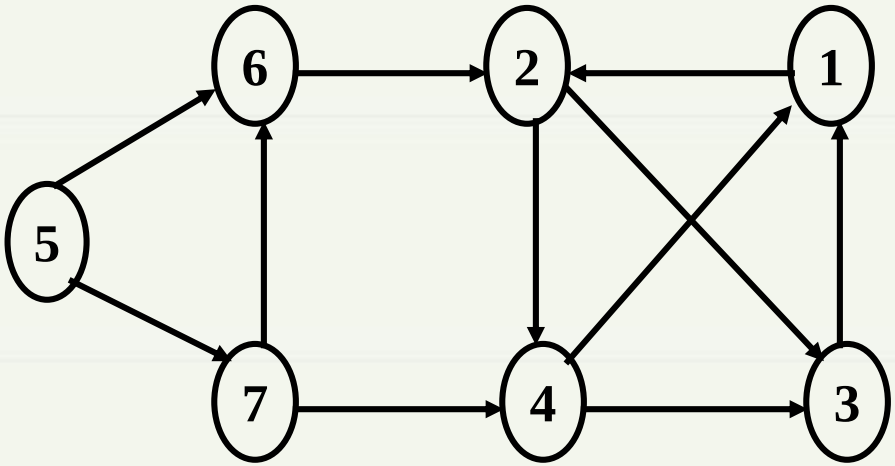


从a开始：a c b f d e

从c开始：c a e f d b

8.3 图的重要操作—遍历

8.3.1 图的深度遍历：DFS



从1开始：1　2　3　4

从5开始：5　6　7

从5开始：5　7　6　2　4　3　1

8.3 图的重要操作—遍历

8.3.1 图的深度遍历：DFS

4、算法：

从某一顶点v深度遍历图G:

```
void DFS (Graph& G, T v, bool &visited[])
{ //从顶点v开始深度优先遍历图G
    visite(v);        //访问顶点v
    i= G.getVertexPos(v);   //顶点v的存储位置
    visited[i] = true;                  //作访问标记
    T w = G.getFirstNeighbor(v);     //v的第一个邻接顶点
    k=G.getVertexPos(w);
     while (k != -1) {                        //若邻接顶点w存在
        if ( !visited[k] ) DFS(G, w, visited);  //若w未访问过, 递归访问顶点w
        w = G.getNextNeighbor (v, w); //下一个邻接顶点
        k=G.getVertexPos(w);  }
};
```

8.3 图的重要操作—遍历

8.3.1 图的深度遍历：DFS

深度优先遍历图G：

```
void DFS_Graph (Graph& G)
{ //深度优先遍历图（图可能不连通或从一个顶点不能遍历）
    int i, loc, n = G.NumVertices();    //顶点个数
    bool *visited = new bool[n];         //创建辅助数组
     for (i = 0; i < n; i++)
         visited [i] = false;       //辅助数组visited初始化
     for(i=0;i<n;i++)   //从一个未访问的顶点开始深度优先遍历
        if(!visited[i]) DFS(G, G.NodeTable[i].data, visited);
     delete [] visited;             //释放visited
};
```

8.3 图的重要操作—遍历

8.3.2 图的广度遍历：BFS

1、遍历方式：假设图为G=（V，E），从v0开始广度优先遍历图。

访问v0，作已访问标志；

依次访问与v0邻接但未访问的各个顶点；

再依次访问这些顶点的未被访问的邻接点，.....

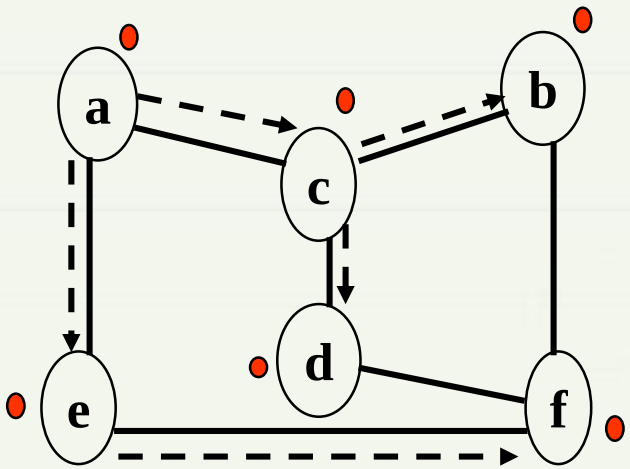
2、特点：类似与树的层次遍历。

尽可能先在横向上访问邻接点，即由近及远，依次访问和出发点有路径相通，且路径长度为1，2...的顶点。

8.3 图的重要操作—遍历

8.3.2 图的广度遍历：BFS

3、举例：

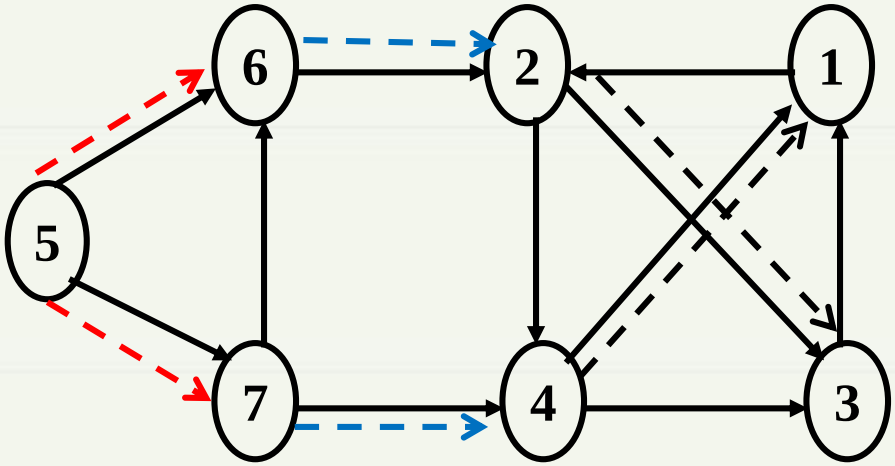


从a开始：a c e b d f

从c开始： c a b d e f

8.3 图的重要操作—遍历

8.3.2 图的广度遍历：BFS



从5开始： 5 6 7 2 4 3 1

从1开始：1 2 3 4

从5开始： 5 6 7

8.3 图的重要操作—遍历

8.3.2 图的广度遍历：BFS

4、算法：

从某一顶点v广度遍历图G:

```
void BFS (Graph& G, const T& v)
{ //从顶点v开始广度优先遍历图G
    int i, w, n = G.NumVertices();       //图中顶点个数
    bool *visited = new bool[n];
    for (i = 0; i < n; i++) visited[i] = false;
    int loc = G.getVertexPos (v);              //取顶点号
    visite(v);       //访问顶点v
    visited[loc] = true;                       //做已访问标记
      Queue  Q;
      Q.EnQueue (v);        //顶点进队列, 实现分层访问
```

8.3 图的重要操作—遍历

8.3.2 图的广度遍历：BFS

```
    while (!Q.IsEmpty() ) {                    //循环, 访问所有结点
        Q.DeQueue (v);
        k = G.getFirstNeighbor (v);      //第一个邻接顶点
        while (k != –1) {                        //若邻接顶点w存在
            if (!visited[k]) {                   //若未访问过
                visite(w);                  //访问
                visited[k] = true;
                Q.EnQueue (w); } //顶点w进队列
            k = G.getNextNeighbor (v, w); //找顶点loc的下一个邻接顶点
        }
    }      //外层循环，判队列空否
  delete [] visited;
}
```

8.3 图的重要操作—遍历

注意：

（1） 在没有给出图的存储时，图的某种遍方式得到的遍历序列可能是不唯一的。（为什么？）

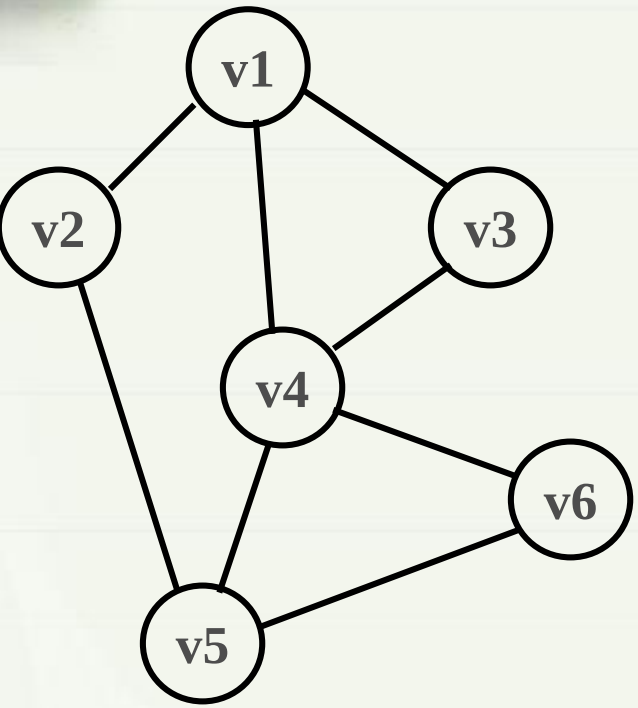
（2） 从某顶点开始遍历图，不一定能访问到图中的所有元素。（为什么？）。

只能访问到该顶点所在最大连通子图（连通分量）的所有顶点。

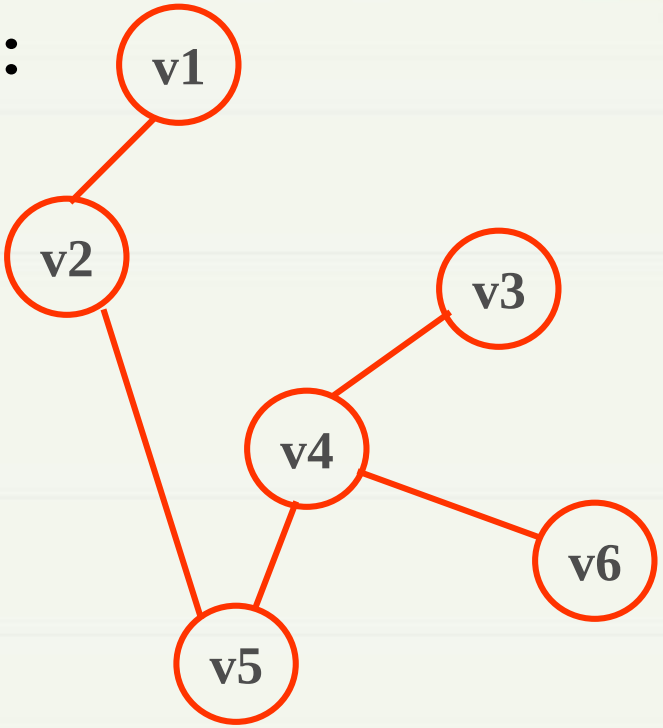
（3） 对于连通无向图，遍历访问了n个元素，经过了n-1条边，它们组成的子图就是生成树。（深度生成树、广度生成树）。

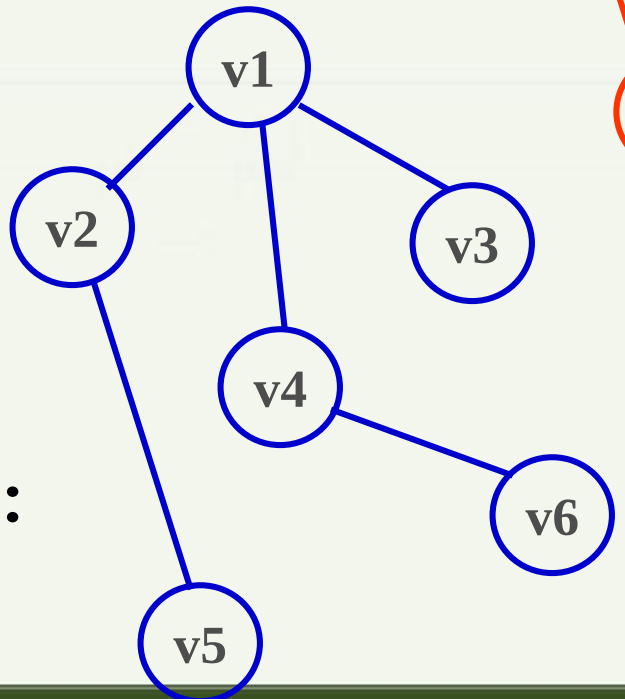
对于非连通无向图，遍历可得到连通分量的生成树森林

8.3 图的重要操作—遍历



深度生成树：
从v1开始





广度生成树：
从v1开始

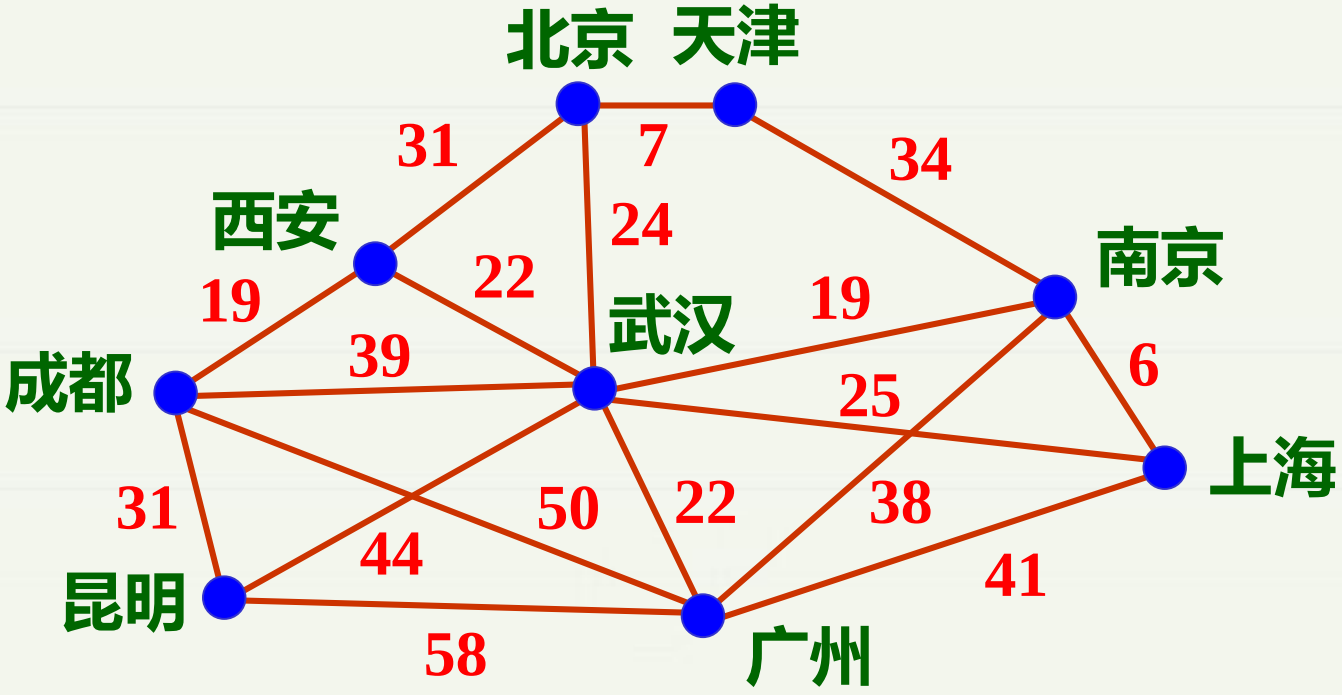
8.4 最小连通代价问题

无向连通图的生成树有很多，如果图的边具有权值，那么各个生成树的边的权值之和大小就不同。在所有生成树中，权值之和最小的一棵称为最小生成树

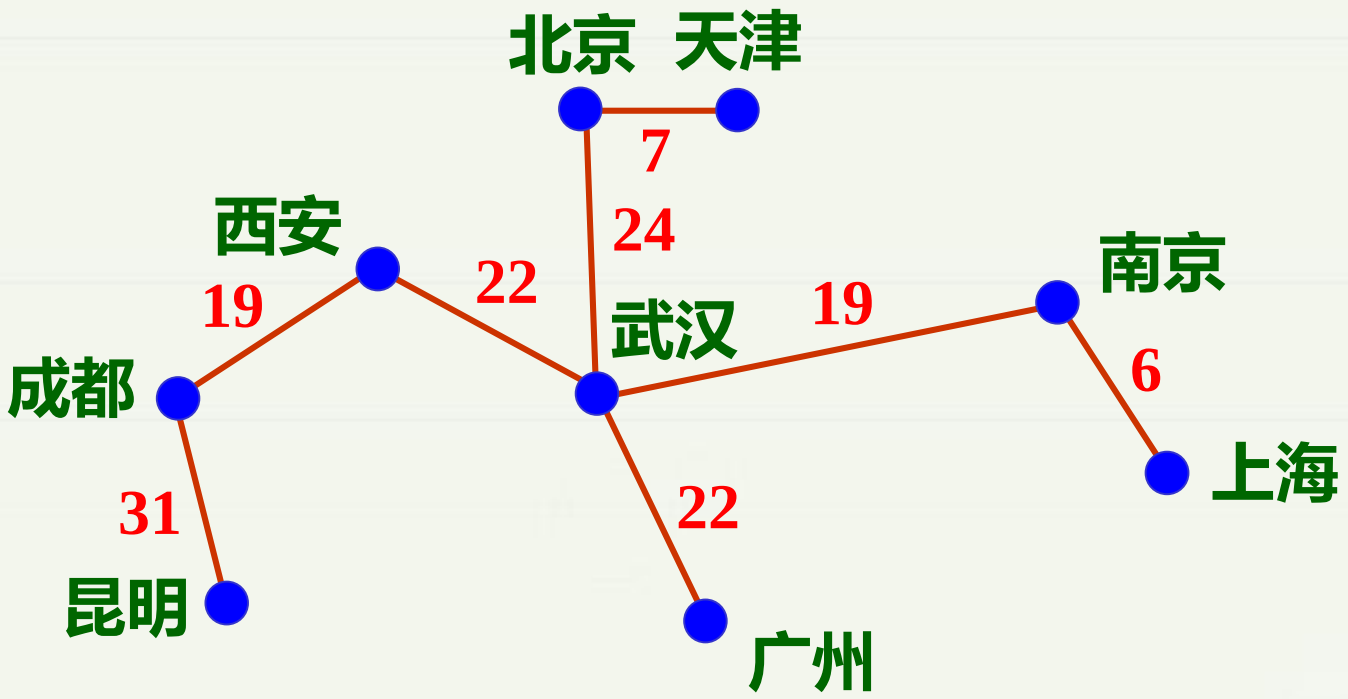
1. 最小生成树：假设图是一个加权连通图，具有最小权值之和的生成树称为最小代价生成树。
   Minimum Cost Spanning Tree (MST)

假设有一个网络，用以表示 n 个城市之间架设通信线路，边上的权值代表架设通信线路的成本。如何架设才能使线路架设的成本达到最小？

8.4 最小连通代价问题


北京
天津
西安
南京
武汉
成都
上海
昆明
广州
31
7
34
24
19
22
19
39
6
25
31
50
22
38
44
41
58

8.4 最小连通代价问题



8.4 最小连通代价问题

2. MST性质：

假设 G=(V,E) 是加权连通图，U是V的非空子集，若(u_0, v_0)是一条最小权值的边，其中$u_0 \in U$，$v_0 \in V-U$；则：(u_0, v_0)必在最小生成树上。

[反证法] 假设任何最小生成树均不包含边(u0,v0)，不妨设T是一棵最小生成树，将(u0,v0)加入T中，则必定存在回路！即一定存在另外一条边(u',v')跨越两个集合，即u' ∈ U，v' ∈ V-U，删除(u',v')，则得到另外一棵生成树T'，而T'的权值之和肯定不大于T，于是T'是一棵含有(u0,v0)的最小生成树，矛盾！

8.4 最小连通代价问题

3. 最小生成树的构造方法：

——有多种算法，但最常用的是以下两种：

- Kruskal（克鲁斯卡尔）算法——边归并
- Prim（普里姆）算法 ——顶点归并

这两种方法都是采用的贪心策略，都是基于MST性质的！

贪心准则： 选两个顶点不在同一连通分量上的边中权值最小的。

8.4 最小连通代价问题

■ 克鲁斯卡尔（Kruskal）方法:

设$N = \{ V, E \}$是有 n 个顶点的连通网，

步骤:

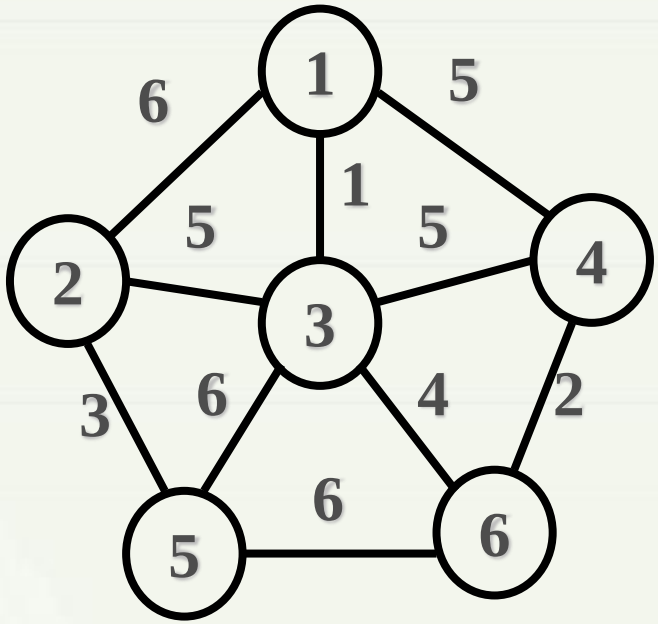
(1) 首先构造一个只有 n 个顶点但没有边的非连通图 $T = \{ V, \varnothing \}$, 图中每个顶点自成一个连通分量。

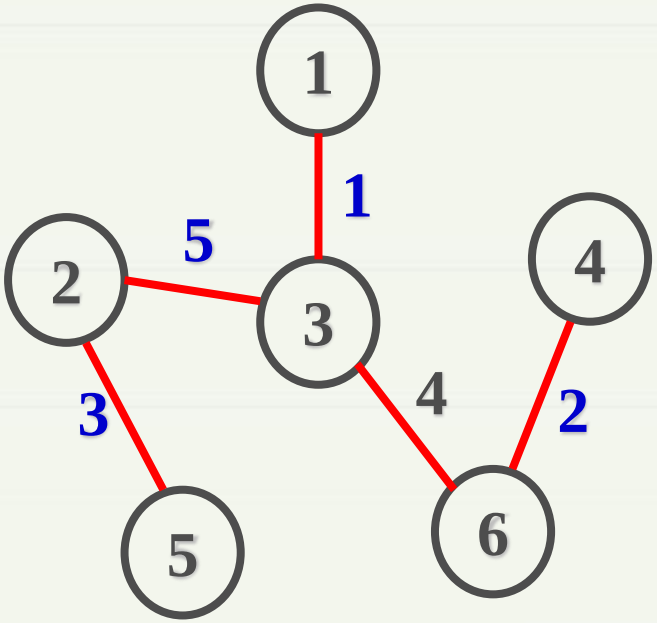
(2) 在边集E中选则具有最小权值的边,若该边的两个顶点落在T中不同的连通分量上，则将此边加入到生成树的边集合T中；否则将此边舍去，重新选择一条权值最小的边。

(3) 如此重复下去，直到所有顶点在同一个连通分量上为止。此时的T即为所求（最小生成树）。

8.4 最小连通代价问题

举例：





图采用邻接表存储，算法实现：略！

8.4 最小连通代价问题

■ 普利姆（Prim）方法：

设：N =（V , E）是个连通网，
另设U为最小生成树的顶点集，TE为最小生成树的边集。

步骤：

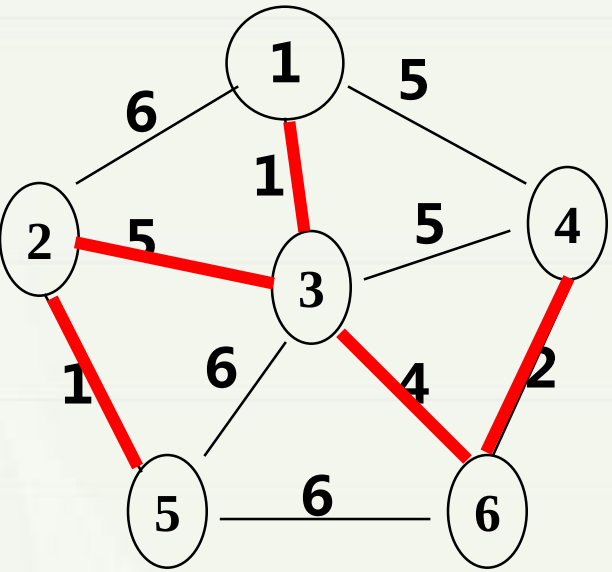
(1) 初始状态：令U={u0},u0∈V，TE={ }, T=(V,TE)

(2) 在所有u∈U, v∈V-U的边(u,v)中找一条权值最小的(u′,v′)，(u′,v′)并入TE，即TE = TE∪{(u′,v′)}，v′并入U，即U = U∪{v′}

(3) 重复第二步，直到U = V为止。此时TE中必有n-1条边，T = (U，TE) 就是最小生成树。

8.4 最小连通代价问题

举例：



U={1} V-U={2,3,4,5,6} TE={}

{(1,2) (1,3) (1,4)}

TE={(1,3)} U={1,3}

{(1,2) (1,4) (3,2) (3,4) (3,5) (3,6)}

TE={(1,3) (3,6)} U={1,3,6}

{(1,2) (1,4) (3,2) (3,4) (3,5) (6,4) (6,5)}

TE={(1,3) (3,6) (6,4)} U={1,3,6,4}

{(1,2) (3,2) (3,5) (6,5)}

TE={(1,3) (3,6) (6,4) (3,2)} U={1,3,6,4,2}

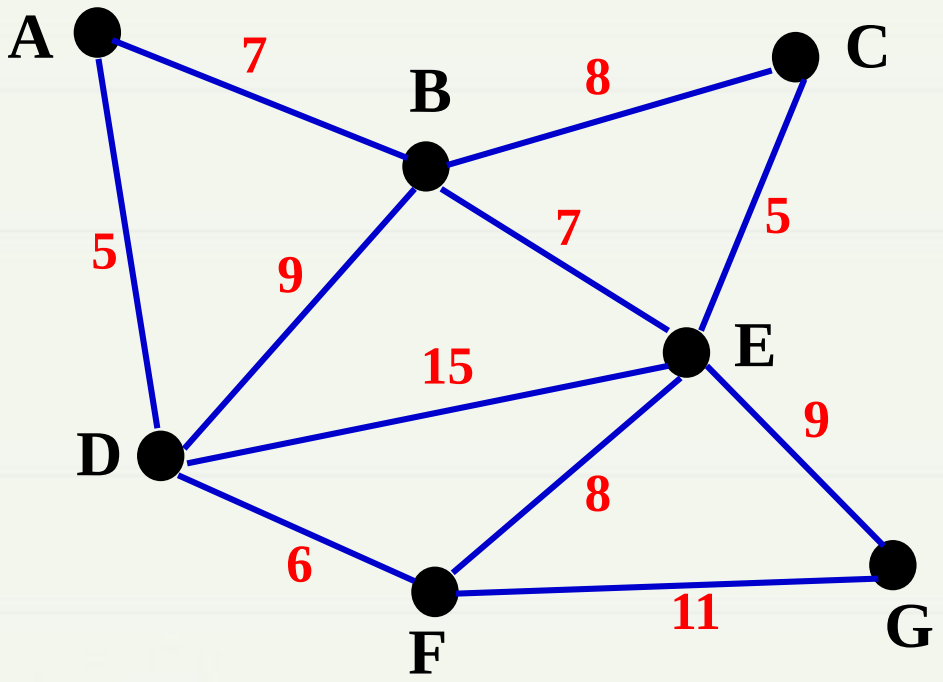
{ (2,5) (3,5) (6,5)}

TE={(1,3) (3,6) (6,4) (3,2) (2,5)} U={1,3,6,4,2,5}

图采用加权邻接矩阵存储，算法实现：略！

8.4 最小连通代价问题

练习：
分别用Kruskal、Prim 方法画出最小生成树。



？ 最小生成树唯一吗？什么情况下唯一？

8.5 最短路径问题

在现实中,有时要从甲地到乙地,有两种行路的方案:

其一：为了减少麻烦，选择中转次数最少的路线；

其二：为了节省时间，选择距离最短的路线；

采用广度优先方式遍历图

本节讨论的最短路径问题，根据图中各边的权值选择路径。

8.5 最短路径问题

最短路径（shortest paths）是一种重要的图算法，在日常生活中的需求也非常普遍。

例如：

- 两地之间有无路可通？在有几条路的情况下，哪一条路最短？城市交通导游、GPS导游等等；
- 邮政自动分拣机的路选装置，要选择最短的路并投递；
- 计算机网络的路由选择尽管很复杂，但是最基本的路由问题仍然是最短路径问题；

8.5 最短路径问题

给定一个带权重的图G=(V,E)，边的 权重函数为w:E→R，则一条路径p=$v_1 \to v_2 \to \cdots \to v_k$的权重为该路径上每条边的权重之和：

$$w(p)=\sum_{i=k}^{k-1} w(v_i, v_{i+1})$$

[最短路径] 无论是有向图还是无向图，从任意定点u到v的最短路径是从所有u到v的路径中权重最小的路径。最短路径权重定义为：

$$\delta(u,v)=\min\{\, w(p): p\text{是从}u\text{到}v\text{的路径}\}$$

最短路径问题：

- ◆单源单点最短路径问题：人们最容易想到的问题，但…
- ◆单源多点最短路径问题—Dijkstra（迪杰斯特拉）算法
- ◆多源多点最短路径问题—Floyd（弗洛伊德）算法

8.5 最短路径问题

■ 单源单点最短路径问题：

直观感觉上，寻找单源单点最短路径的时间复杂性将大大超过边的数目！！

时间复杂性与一对顶点之间的路径的条数成正比，那么一对顶点之间有多少条路径呢？最坏情况下至少是(n-2)!，至多是(n-1)!，n=|V|，即O(n!)。 因此，最坏时间复杂性是巨大的！

对于一般性的一对顶点(单源单点)之间的最短路径问题，目前还没有找到更好的办法。

8.5 最短路径问题

■ 单源多点最短路径问题：

尽管比较糟糕，但是人们发现，在找一对顶点之间的最短路径时，已经考察了源点到任意顶点的所有路径，既然如此，何不一次性地找出源点到所有顶点的最短路径呢？这样平摊下来，花费在一对顶点之间的最短路径上的成本就会大大降低！

更可喜的是，求一个顶点到其他所有顶点的最短路径问题可以使用贪心策略，贪心+摊销，从而使得最短路径的搜索成本大为下降。

1959年埃德加.沃波.狄杰斯特拉(Edsger.Wybe.Dijkstra)发明了著名的最短路径算法—Dijkstra算法。

8.5 最短路径问题

■ 单源多点最短路径问题：

1. 问题描述

设一有向图G=（V, E），已知各边的权值，以某指定点v_0为源点，求从v_0到图的其余各点的最短路径。限定各边上的权值大于或等于0。

2. Dijkstra 算法的基本思想

贪心策略：从源点向外延伸，越短的路径（终点）越早被求出！——思想与洪水泛滥类似！

即：找出目前离源点最近的顶点（它和源点构成了一条新的最短路径，这条路径应该在未求出的路径中最短！）

8.5 最短路径问题

■ 单源多点最短路径问题：

于是：
第一条最短路径是：从源点出发，不经过任何顶点可达的所有路径中最短的那条。

按照Dijkstra的思想，第二条肯定比第一条要长，但应该是从源点可达的路径中最短的那条。哪条？两种可能：
（1）从源点直达
（2）经过其他顶点，此时要经过只能经过第一条（的顶点）

第三条呢？ 类似！

“按路径长度的非递减次序逐一产生各条最短路径”
——贪心准则

8.5 最短路径问题

■ 单源多点最短路径问题：

3. Dijkstra 算法

设v_0是源点，S={v_1,…,v_k}是已经求得最短路径的顶点集合，V-S就是还未求出最短路径的顶点集合。一个顶点v_i属于 S当且仅当从源点v_0到v_i的最短路径已经求出。

引入一个辅助数组dist。它的每一个分量dist[i]记录着源点v_0到v_i的当前最短路径长度。

（1）初始化：dist的初始状态：对每个顶点有：

- 若从v_0到顶点v_i有边, 则dist[i]为该边的权值；
- 若从v_0到顶点v_i无边, 则dist[i]为∞

$$dist[i]=\begin{cases} w_{ki} \\ \infty \end{cases}$$

k是源点的存储位置，

$<v_0,v_i>\in E$,w_{kj}是其权值

$<v_0,v_i>\notin E$

8.5 最短路径问题

■ 单源多点最短路径问题：

(2)求一条最短路径：

（应用贪心准则选择求出每一条最短路径）

即在未求出最短路径的顶点中选取离源点距离最近的顶点。

在**V-S**中，选取具有最短的当前路径的顶点v_k，满足：

dist[k]=min{ dist[i] | v_i ∈V-S }

$S \leftarrow S \cup \{v_k\}$;

(3) 修正：对所有未求出最短路径且又与v_k邻接的顶点v_j ∈V-S按下式修正（新最短路径的出现是否使各个顶点的当前路径变短），即：

dist[j]=min{ dist[j],dist[k]+w(k,j) }

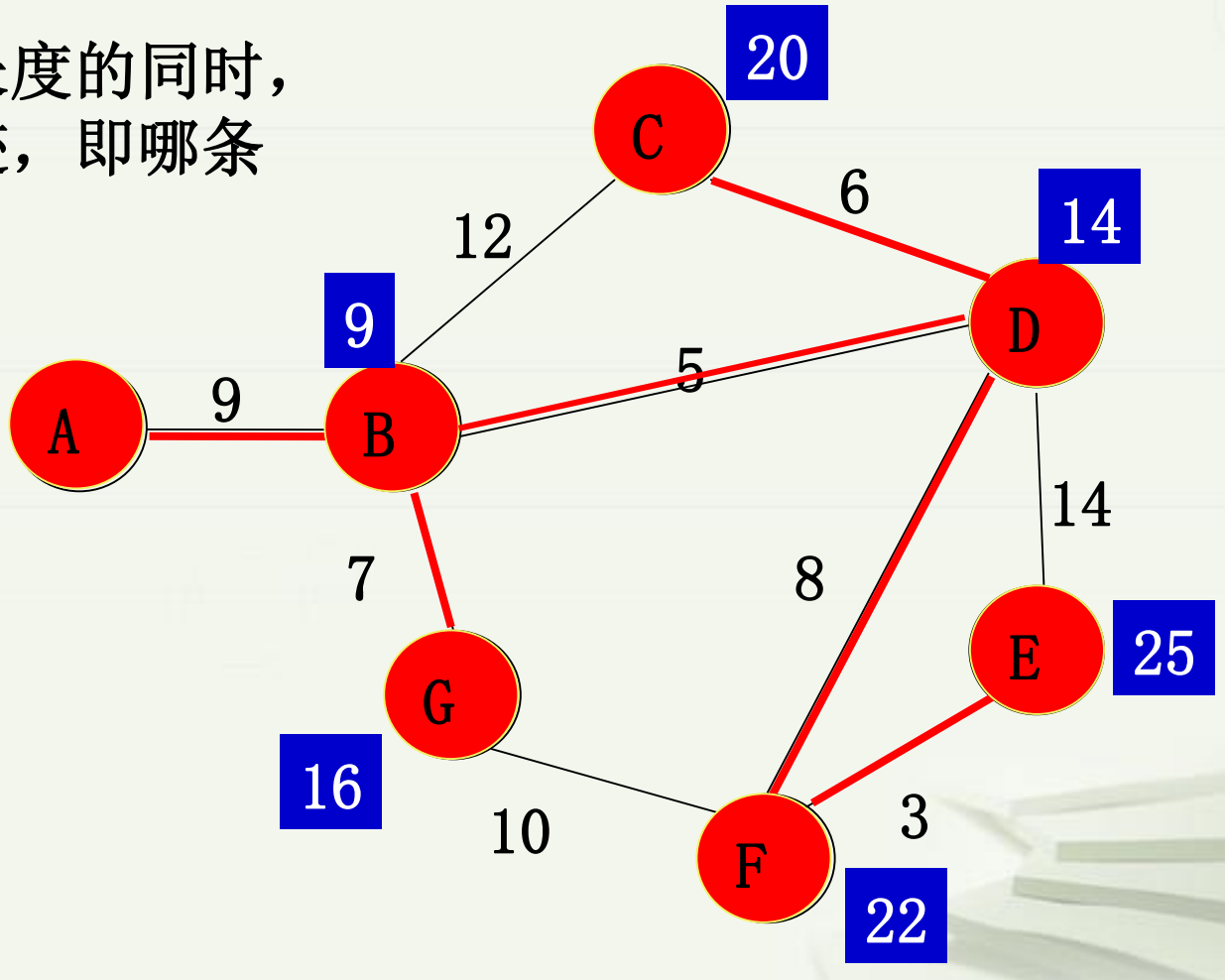
（4）重复（2）（3），直到**S=V,**

即，所有顶点最短路径都已经求出。

8.5 最短路径问题

■ 单源多点最短路径问题：

注意：在记录路径长度的同时，还应该记录路径轨迹，即哪条路径最短。

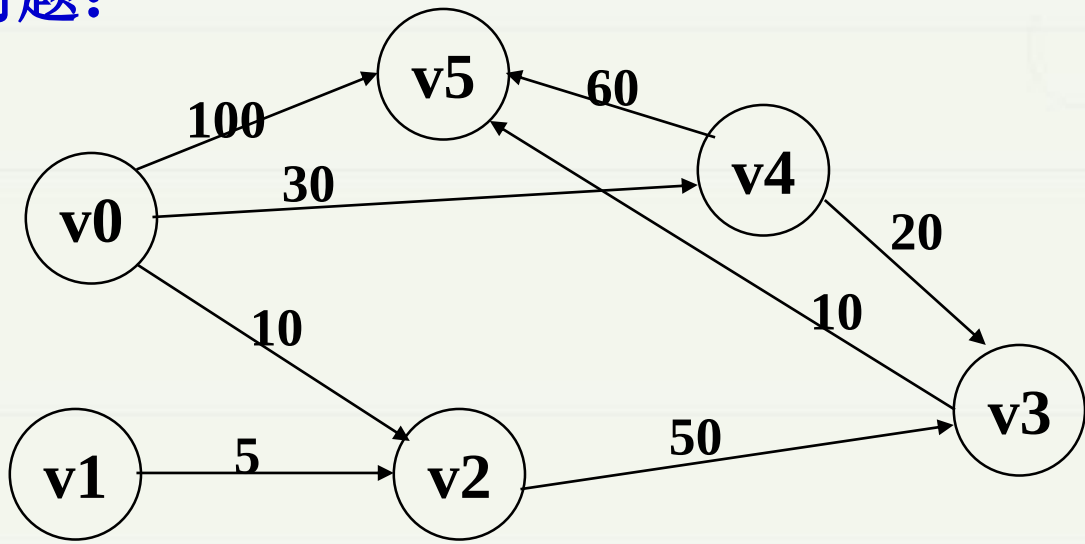


8.5 最短路径问题

■ 单源多点最短路径问题：

4. 举例：

求v_0到其余各顶点的最短路径。



存储结构：

加权邻接矩阵

0	v0
1	v1
2	v2
3	v3
4	v4
5	v5

	0	1	2	3	4	5
0	0	∞	10	∞	30	100
1	∞	0	5	∞	∞	∞
2	∞	∞	0	50	∞	∞
3	∞	∞	∞	0	∞	10
4	∞	∞	∞	20	0	60
5	∞	∞	∞	∞	∞	0

8.5 最短路径问题

■ 单源多点最短路径问题：

初始化：　　S={ v_0 }

dist[1]= ∞	path[1]=[ ]
dist[2]=10	path[2]=[v0,v2]
dist[3]= ∞	path[3]=[ ]
dist[4]=30	path[4]=[v0,v4]
dist[5]= 100	path[5]=[v0,v5]

第 1 条最短路径 v_0->v_2 ， 长度10；

S={v_0, v_2}　　path=[v_0, v_2]

8.5 最短路径问题

■ 单源多点最短路径问题：

dist[1]= ∞ ? dist[2]+<v_2,v_1>的权= 10+ ∞= ∞

dist[3]= ∞ ? dist[2]+<v_2,v_3>的权=10+20= 60

dist[4]=30 ? dist[2]+<v_2,v_4>的权=10+ ∞= ∞

dist[5]= 100 ? dist[2]+<v_2,v_5>的权=10+ ∞= ∞

path[3]=[v_0,v_2,v_3]

dist[1]= ∞

dist[3]= 60

dist[4]=30

dist[5]= 100

第2条最短路径 v_0->v_4 ， 长度30；

S={v_0, v_2, v_4}　　path=[v_0, v_4]

8.5 最短路径问题

■ 单源多点最短路径问题：

dist[1]= ∞ ? dist[4]+<v_4,v_1>的权= 30+ ∞= ∞

dist[3]= 60 ? dist[4]+<v_4,v_3>的权=30+20= 50

dist[5]= 100 ? dist[4]+<v_4,v_5>的权=30+60= 90

dist[1]= ∞

dist[3]= 50

dist[5]= 90

path[3]=[v_0,v_4,v_3]

path[5]=[v_0,v_4,v_5]

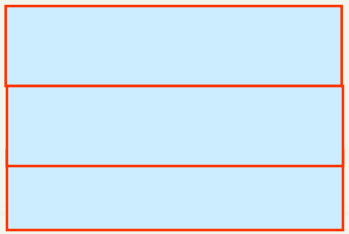
第3条最短路径 v_0->v_4->v_3 ， 长度50；

S={v_0, v_2, v_4, v_3}　　path=[v_0, v_4, v_3]

8.5 最短路径问题

■ 单源多点最短路径问题：

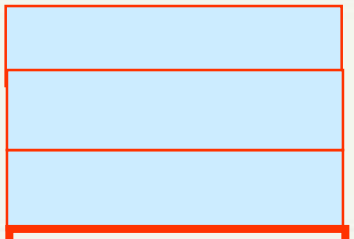
dist[1]= ∞　? dist[3]+<v_3,v_1>的权= 50+ ∞= ∞



dist[5]= 90　? dist[3]+<v_3,v_5>的权=50+10= 60

path[5]=[v_0,v_4,v_3,v_5]

dist[1]= ∞



dist[5]= 60

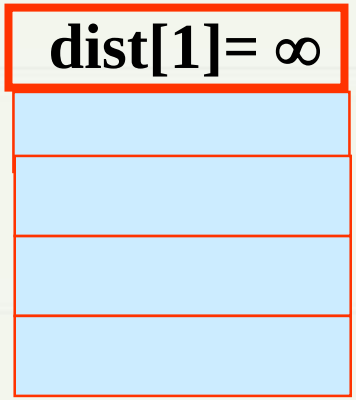
第4条最短路径 v_0->v_4->v_3->v_5 ， 长度60；

S={v_0, v_2, v_4, v_3, v_5}　path=[v_0, v_4, v_3, v_5]

8.5 最短路径问题

■ 单源多点最短路径问题：

dist[1]= ∞　？dist[5]+<v_5,v_1>的权= 60+ ∞= ∞



第5条最短路径 ： 无

5、算法： **参见教材P377-378**

8.5 最短路径问题

■ 多源多点最短路径问题：

1. 问题：对于有向加权图G，求每一对顶点之间的最短路径。

2. 方法：

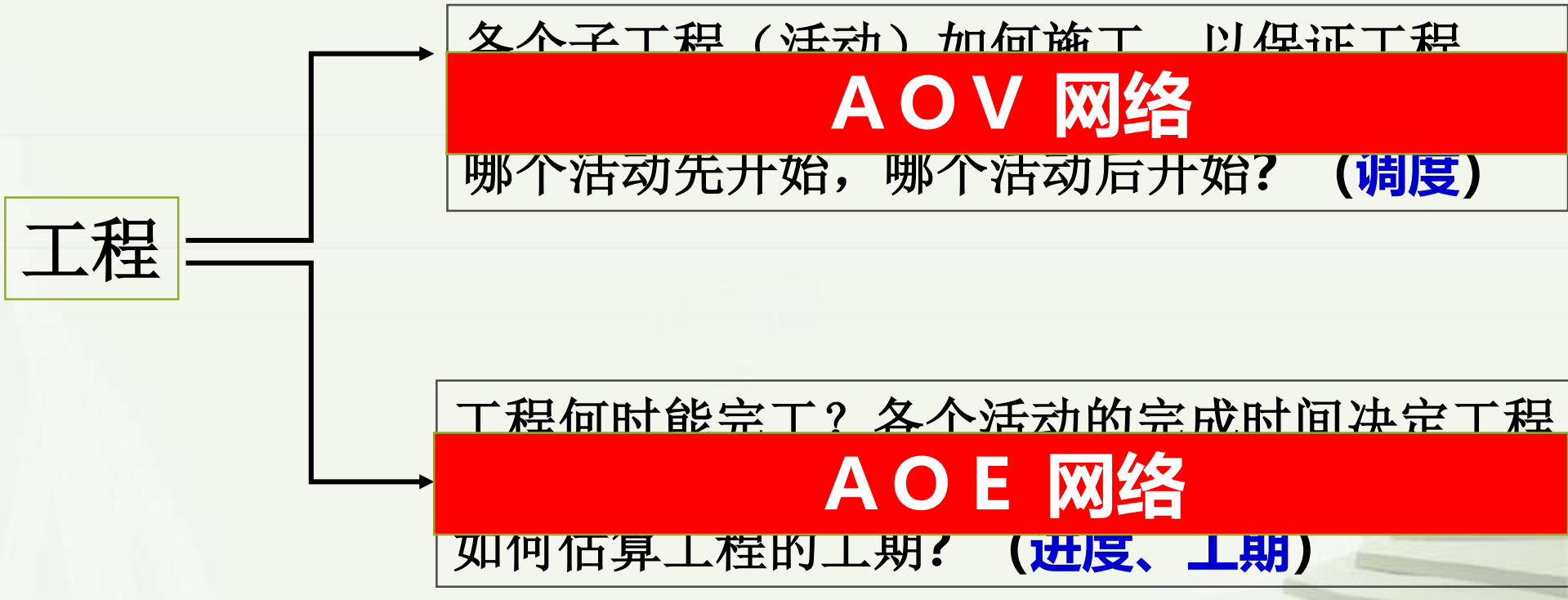
方法 1 ：调用上面的算法，以每个顶点作为一次源点。

方法 2 ：Floyd算法（略）——动态规划算法

8.6 有向图在工程中的应用

工程（Project）—有时又称为系统，一般指一项大的或复杂的任务

活动（Activity）—复杂的工程一般要分解为许多的子工程，每个子工程称为活动。



研究 AOV 是研究 AOE 的前提！！

8.6 有向图在工程中的应用

8.6.1 AOV网络及拓扑分类

AOV网络：用顶点表示工程的活动，有向边表示活动之间的优先关系(Activity On Vertex)的有向图。

前驱、后继：在AOV网络中，若从顶点V_i到V_j有一条有向路径，则称V_i是V_j的前驱，V_j是V_i的后继；若$<V_i,V_j>$是有向，则称V_i是V_j的直接前驱，V_j是V_i的直接后继。

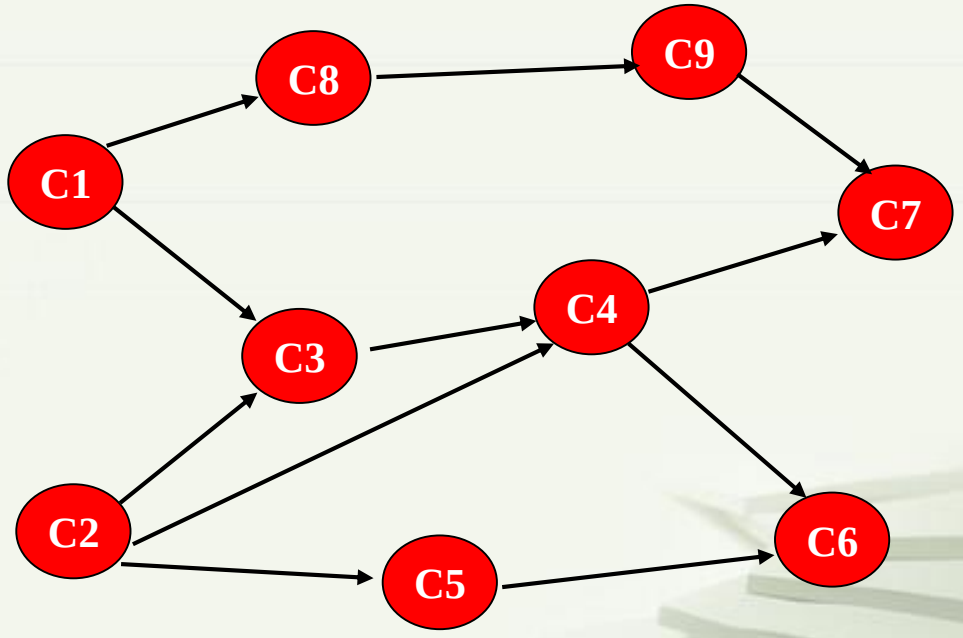
前驱后继关系表示一个活动是另一个活动的先决条件，即前驱活动完成，后继活动才可以开始。

8.6 有向图在工程中的应用

8.6.1 AOV网络及拓扑分类

例如，计算机专业学生必须完成一系列规定的基础课和专业课才能毕业，这就是一个工程。学习一门课程就是一个活动。先学哪些课，后学哪些课是有要求的，即活动之间有优先关系。

课程号	课程名称	先修课程
C1	高等数学	
C2	程序设计基础	
C3	离散数学	C1，C2
C4	数据结构	C3，C2
C5	高级语言程序设计	C2
C6	编译原理	C4，C5
C7	操作系统	C4，C9
C8	普通物理	C1
C9	计算机原理	C8



教学计划的AOV

8.6 有向图在工程中的应用

8.6.1 AOV网络及拓扑分类

拓扑序列：对于有向图，其顶点$V_1,V_2,...V_n$的一种排列，如果满足：V_i到V_j有一条有向路径，则V_i排在V_j前面。则称顶点是拓扑有序的，亦称为顶点的一个拓扑序列。

如果顶点集合（工程的活动）能排为拓扑序列，则说明工程就得到了一个合理的施工调度方案。

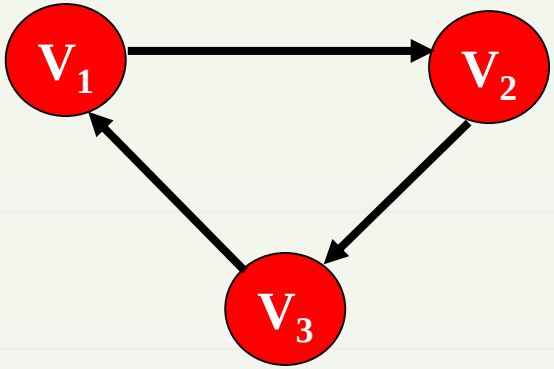
拓扑排序（分类）：得到AOV网络顶点的一个拓扑序列的过程。

对于一个工程，进行任务分解，得到各个子工程（活动），确定它们之间的优先关系，然后就可以构建出AOV网络。如果能得到AOV网络的顶点的一个拓扑序列，就得到了工程的一个活动（施工）调度方案。

8.6 有向图在工程中的应用

8.6.1 AOV网络及拓扑分类

AOV网络中不能有环！ 为什么？



如何判断AOV有没有环？

看能否找到AOV网络顶点的一个拓扑排列（拓扑序），若所有顶点能排成一个拓扑序列，则不存在环，否则，有环。

8.6 有向图在工程中的应用

8.6.1 AOV网络及拓扑分类

进行拓扑排序的方法：　重复选取入度为0的顶点输出；

假设AOV网络有n个顶点（活动），拓扑排序的步骤：

（1）在AOV网络中选一个没有直接前驱的顶点（入度为0），并输出之；如果有多个，可以任意选一个。

（2）从AOV中删去该顶点及关联的边（即凡是以该顶点为前提条件的，都没有了，邻接顶点的入度减1）

（3）重复（1）、（2）步，直到：

- 全部顶点均已输出（得到拓扑有序序列）或
- AOV中已没有入度为0的顶点（AOV中剩余的顶点构成有向环）

8.6 有向图在工程中的应用

8.6.1 AOV网络及拓扑分类

拓扑序列：

a d b e c f

算法实现：

1、图的存储结构——邻接表（同时记每个顶点的入度）；

2、入度为0顶点的收集，可以采用栈或队列（实际上无顺序）

见教材P386-387，略！

单选题 5分

设置

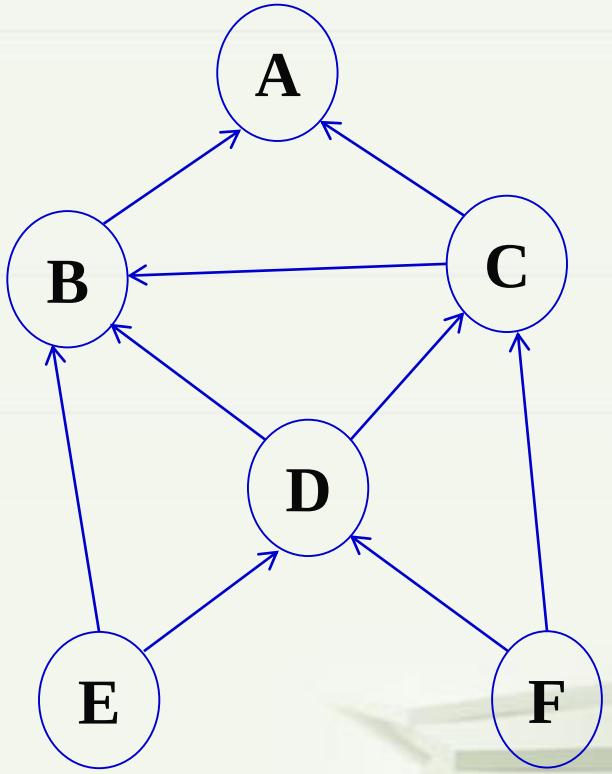
有某工程的AOV网络如下，其正确的拓扑序列（调度顺序）是：

1 F E D C B A

2 A B C D E F

3 E F C D B A

4 E F D B A C



提交

8.6 有向图在工程中的应用

8.6.2 AOE网络及关键路径

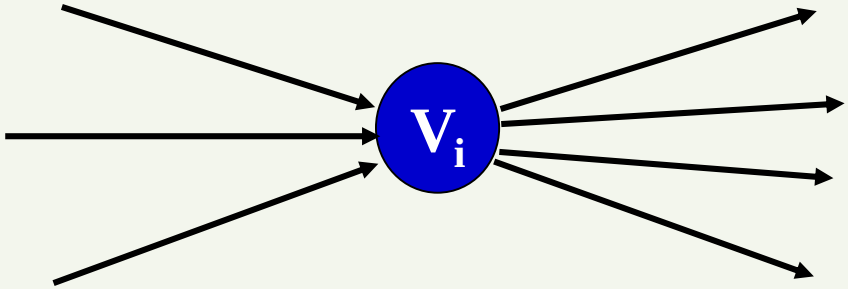
■ 有关基本概念

[事件 Event]：一些活动完成后产生的结果或状态。

AOE网络：用顶点表示事件，有向边表示活动，有向边上的权值表示活动的持续时间，这样的有向图称为AOE网络
(Activity On Edge)

[源点]：在AOE中，只有一个入度为0的顶点（起始事件）；
[汇点]：在AOE中，只有一个出度为0的顶点（结束事件）；

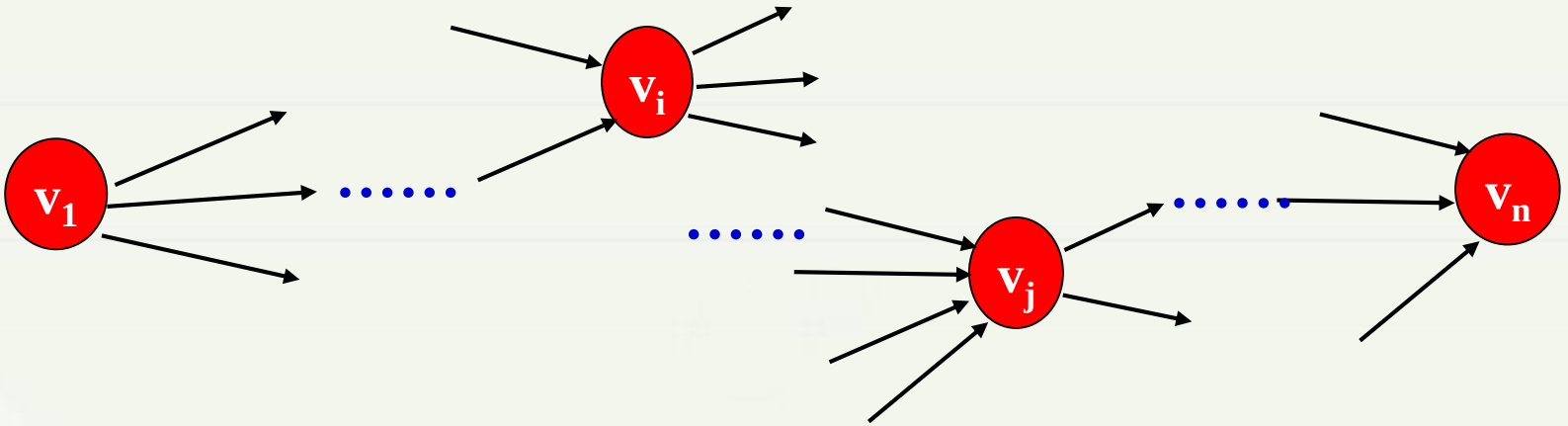
活动与事件的关系：事件发生后，从此事件出发的活动就可以开始了；进入（影响）事件的活动都完成该事件就发生。



8.6 有向图在工程中的应用

8.6.2 AOE网络及关键路径

AOE表示的工程：源点（开工）事件发生后，一些活动开始，一些活动的结束，又导致发生新事件，新事件发生又有活动开始.....，最后，一些活动完成，产生汇点（竣工）事件。

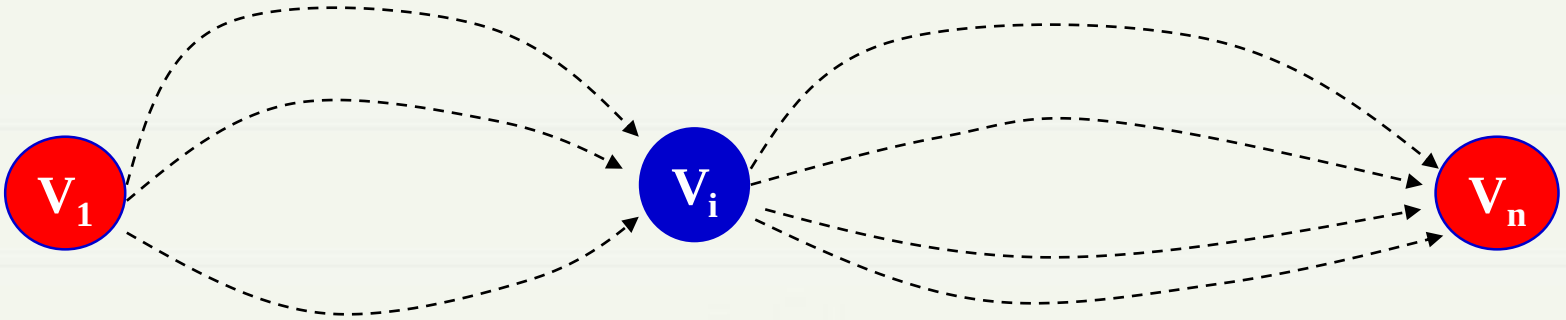


工程的工期：源点到汇点所有有向路径中，权值之和最长的路径的长度。这条路径长度最长的路径就叫做关键路径**(Critical Path)**。

8.6 有向图在工程中的应用

8.6.2 AOE网络及关键路径

事件 V_i 的最早发生时间$Ve(V_i)$=从源点V_1到事件V_i的所有有向路径中，权值之和最长的路径的长度。



事件 V_i 的最迟发生时间$Vl(V_i)$=在保证汇点V_n 在$Ve[V_n]$ 时刻完成的前提 下，事件V_i 的允许的最迟开始时间
=工程的工期-从事件V_i到汇点所有有向路径中权值之和最长的路径的长度。

8.6 有向图在工程中的应用

8.6.2 AOE网络及关键路径

活动的最早开始时间$Ae(e_k)$= 若 $e_k = <V_i, V_j>$，该活动的最早开始时间为该活动起始事件V_i的最早发生时间 $Ve(V_i)$;

活动的最晚开始时间$Al(e_k)$= 若 $e_k = <V_i, V_j>$，该活动的最迟允许开始时间为该活动的终止事件V_j的最晚发生时间 $Vl(V_j)$- e_k持续时间;

时间余量（松弛时间 slack time）= $Al(e_k)$-$Ae(e_k)$

关键活动：时间余量为0，即$Ae(e_k)$=$Al(e_k)$

8.6 有向图在工程中的应用

8.6.2 AOE网络及关键路径

■ 关键路径的求法

1. 求出各个事件的最早发生时间 $Ve(V_j)$

源点的最早发生时间设为0 ；

对于任一事件V_j，根据定义有：

$Ve(V_j)$ = 从源点到V_j所有路径最长的路径的长度

= Max{从 V_1 到 V_j 的路径长度 }

对所有V_1 到 V_j 的路径

= Max{从V_1 到V_i 的最长的路径长度+<V_i,V_j>的权 }

对所有 V_j 的前驱 V_i

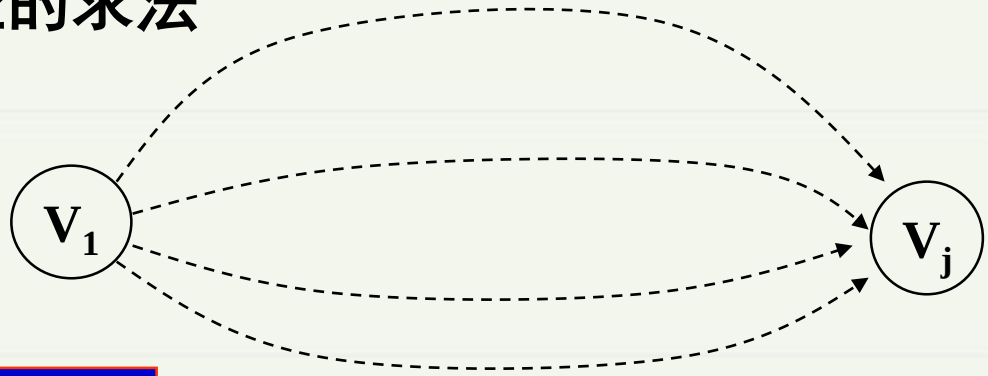
$$Ve(V_j)= Max\{Ve(V_i)+<V_i,V_j> \}$$

对所有 V_j 的前驱 V_i

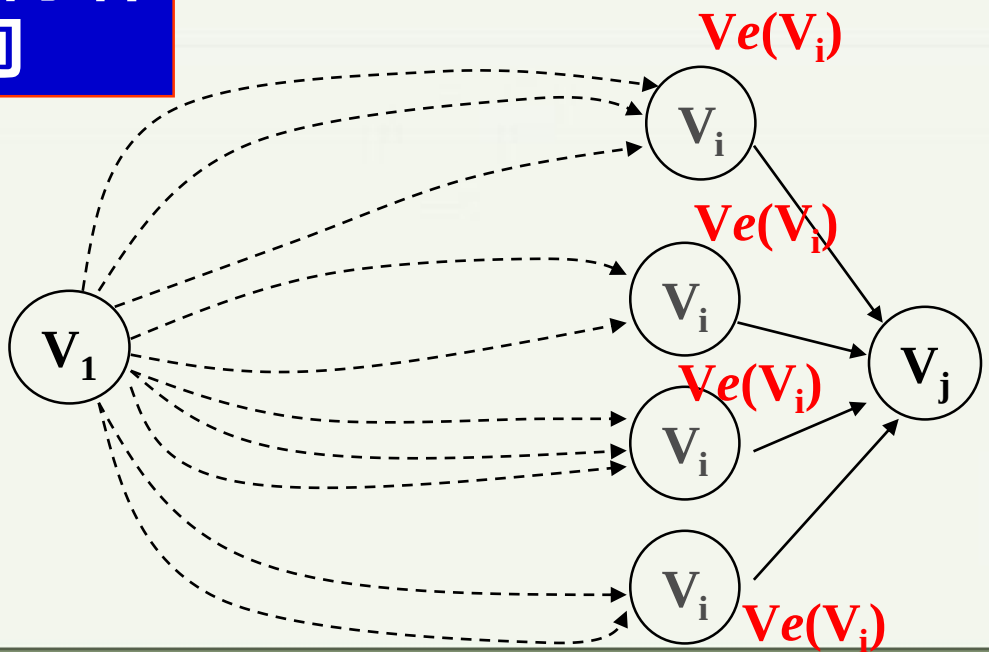
8.6 有向图在工程中的应用

8.6.2 AOE网络及关键路径

- 关键路径的求法



按拓扑顺序求各事件的最早发生时间



8.6 有向图在工程中的应用

8.6.2 AOE网络及关键路径

■ 关键路径的求法

2. 求出各个事件的最晚发生时间 $Vl(V_i)$

汇点的最晚发生时间等于它的最早发生时间 $Vl(V_n)=Ve(V_n)$

对于任一事件V_i，根据定义有：

$Vl(V_i)$=T-从V_i到V_n所有路径中最长的路径的长度

= T-Max{从 V_i 到 V_n 的路径长度 }

对所有V_i 到 V_n 的路径

=T-Max{<V_i,V_j> 的权+从V_j到V_n的最长的路径长度 }

对所有 V_i 的后继 V_j

又： $Vl(V_j)$=T-从V_j到V_n最长的路径的长度 （定义）

所以：从V_j到V_n最长的路径的长度= T- $Vl(V_j)$

8.6 有向图在工程中的应用

8.6.2 AOE网络及关键路径

关键路径的求法

2. 求出各个事件的最晚发生时间 $Vl(V_i)$

$$Vl(V_i)=T-\ Max\{<V_i,V_j>\ 的权+\ T-\ Vl(V_j)\ \}$$

对所有 V_i 的后继 V_j

$$=\ T-\ Max\{T-[Vl(V$$

对所有 V_i 的

$$=\ T-\ T\ +\ Min\{\ Vl(V_j)-\ <V_i,V_j>\ 的权\ \}$$

对所有 V_i 的后继 V_j

100-max{ 100-{10,20,15,30}}
=100-100+min{10,20,15,30}
=10

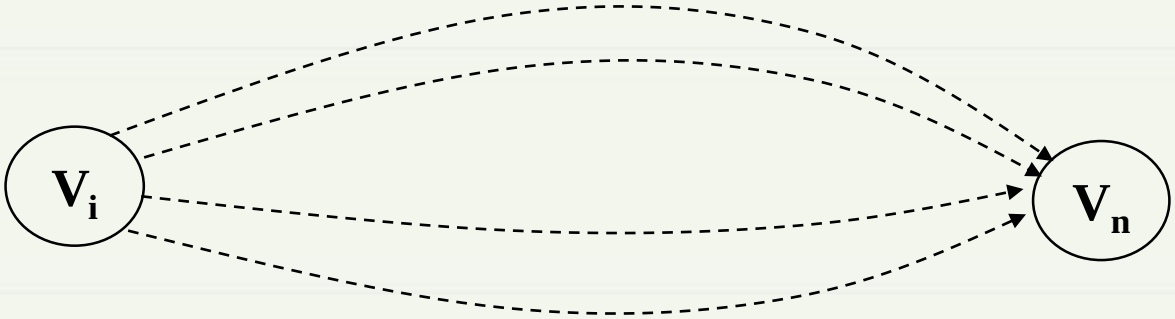
所以：$Vl(V_i)=\ Min\{\ Vl(V_j)-\ <V_i,V_j>\ 的权\ \}$

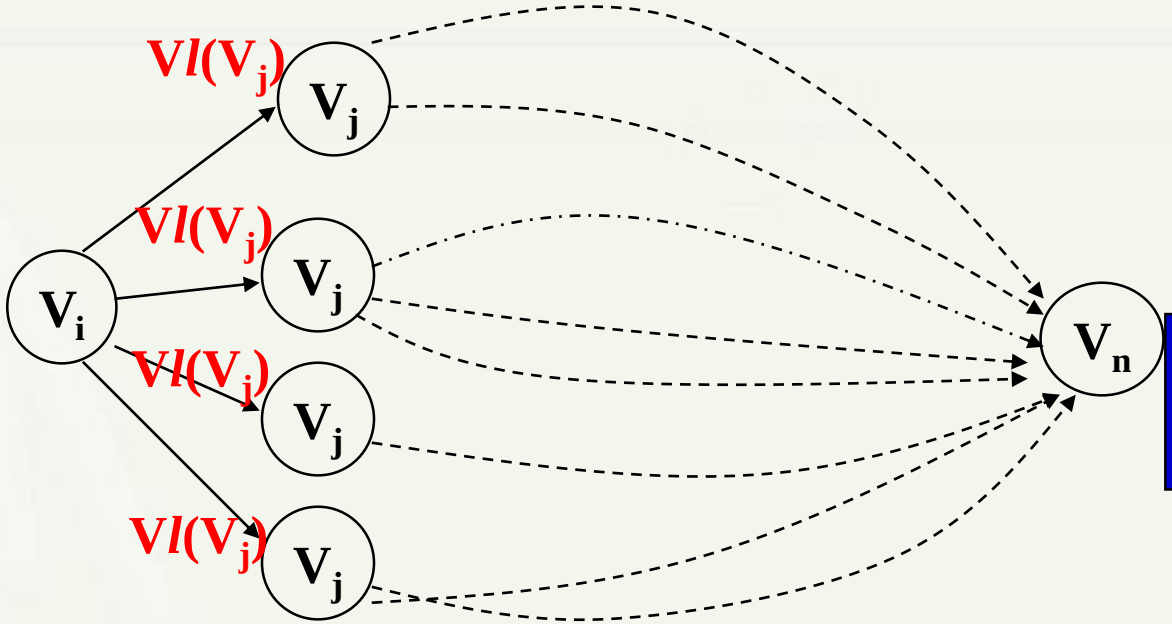
对所有 V_i 的后继 V_j

8.6 有向图在工程中的应用

8.6.2 AOE网络及关键路径

■ 关键路径的求法





按逆拓扑顺序求各事件的最晚发生时间

8.6 有向图在工程中的应用

8.6.2 AOE网络及关键路径

关键路径的求法

3. 求出各个活动的最早开始时间 $Ae(e_k)$

若 $e_k = < V_i, V_j >$，则 $Ae(e_k)=Ve(V_i)$;

4. 求出各个活动的最晚开始时间 $Al(ek)$

若 $e_k = < V_i, V_j >$，则 $Al(e_k)=Vl(V_j)-<V_i,V_j>$的权;

5. 找出关键活动

6. 求出关键路径、工期

7. 结论（缩短工期的方法）:工程管理策略

8.6 有向图在工程中的应用

8.6.2 AOE网络及关键路径

■ 关键路径的求法

算法实现：

1. 存储结构：邻接表
2. 对每个顶点按拓扑排序顺序求Ve，按逆拓扑顺序求Vl
3. 对每个活动，求得到Ae，Al，判断是否相等

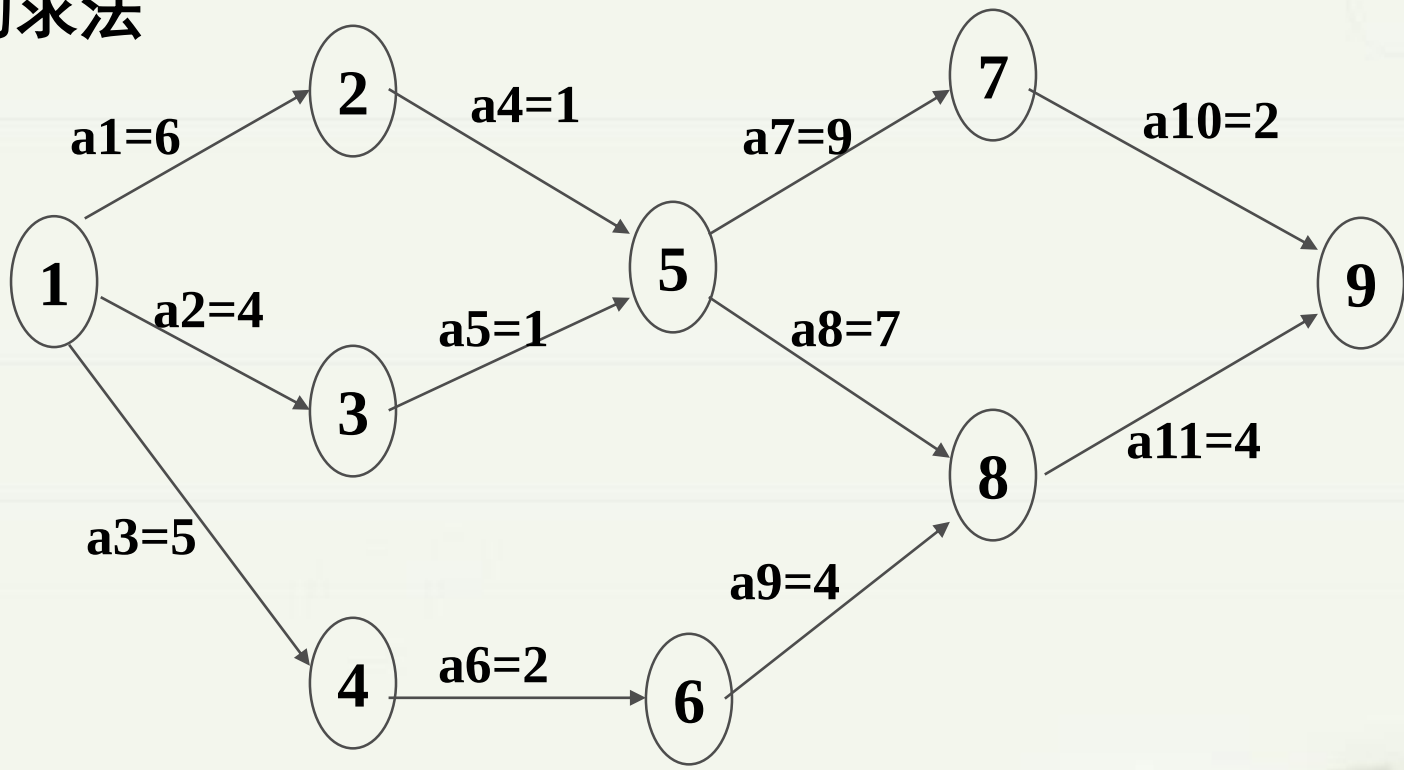
具体见教材P390-391，略！

8.6 有向图在工程中的应用

8.6.2 AOE网络及关键路径

- 关键路径的求法

举例：



V_i	V_e	V_l
1	0	0
2	6	6
3	4	6
4	5	8
5	7	7
6	7	10
7	16	16
8	14	14
9	18	18

8.6 有向图在工程中的应用

8.6.2 AOE网络及关键路径

■ 关键路径的求法

e_k	Ae	Al
a1=<1,2>=6	0	0= 6-6
a2=<1,3>=4	0	2= 6-4
a3=<1,4>=5	0	3= 8-5
a4=<2,5>=1	6	6= 7-1
a5=<3,5>=1	4	6= 7-1
a6=<4,6>=2	5	8= 10-2
a7=<5,7>=9	7	7= 16-9
a8=<5,8>=7	7	7= 14-7
a9=<6,8>=4	7	10= 14-4
a10=<7,9>=2	16	16= 18-2
a11=<8,9>=4	14	14= 18-4

关键路径：

1→2 →5 →7 →9

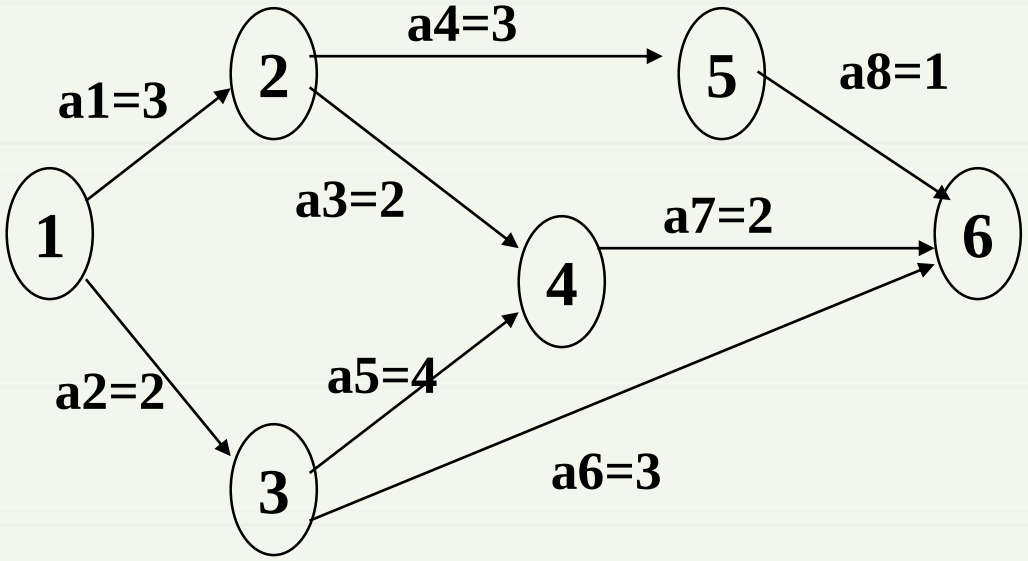
1→2 →5 →8 →9

工期= 18

8.6 有向图在工程中的应用

8.6.2 AOE网络及关键路径

练习：



V_i	V_e	V_l
1	0	0
2	3	4
3	2	2
4	6	6
5	6	7
6	8	8

e_k	A_e	A_l
a1	0	1
a2	0	0
a3	3	4
a4	3	4
a5	2	2
a6	2	5
a7	6	6
a8	6	7

8.6 有向图在工程中的应用

结论：

1、对于一个工程，可以利用AOV网络分析工程在分解时是否合理（各个子工程间有否冲突）；得到工程施工的调度顺序。

2、对于一个工程，在AOV的基础上，可以利用AOE网络分析工程的关键子工程（抓主要活动—关键活动），计算（预测）工程的工期。

3、在不改变关键路径的前提下，提高关键活动的效率，可以缩短工期！

本章小结

重点和难点：

1. 图逻辑结构的特点：多个前驱、多个后继；关系的描述不容易

2. 重要的概念和术语：有向、无向，带权、不带权，完全图，度（入度、出度），邻接，关联，连通（强连通），连通分量（强连通分量），子图，生成树，路径

3. 图ADT的定义：数据结构+基本操作，操作元素（顶点）和关系（边），注意元素和它在图中的存储位置的区别。

4. 图ADT的实现

存储结构：邻接矩阵（一维数组存储元素，二维数组存储关系）

邻接表（一维数组存储元素，单链表存储关系）

操作：操作数据元素、操作关系

本章小结

重点和难点：

5. 图的重要操作—遍历

深度优先遍历：选一个邻接元素，递归（栈）

广度优先遍历：依次访问各个邻接元素，队列

生成树（生成森林）：

注意：图的存储不唯一，遍历序列也就不唯一！

6. 图的应用之：无向图的最小代价连通问题——最小生成树

最小生成树：包含n个顶点+n-1边+连通+权值之和最小

构造方法：贪心策略（具有最小权值的边一定在最小生成树上）

Kruskal

Prim

本章小结

重点和难点：

7. 图的应用之：最短路径问题

求解单源多点最短路径问题方法：贪心策略

Dijkstra算法：按路径长度不减的次序依次求出各条最短路径

8. 图的应用之：工程管理 AOV, AOE

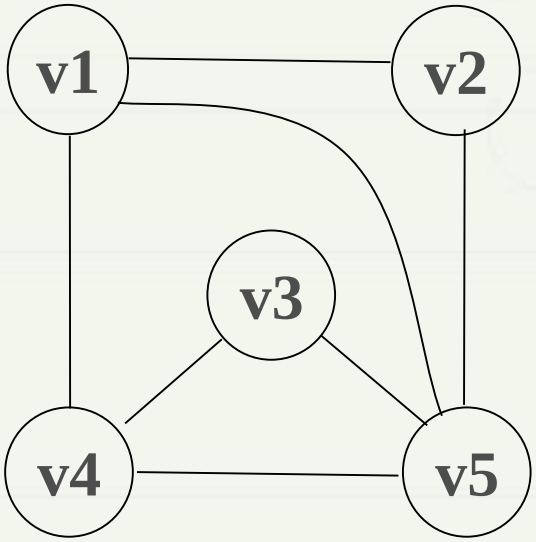
工程的施工调度：AOV，拓扑排序的方法

工程的进度（工期）：AOE，关键路径的求法（事件的最早、最晚事件，活动的最早最晚事件）

深刻理解图表示的工程含义

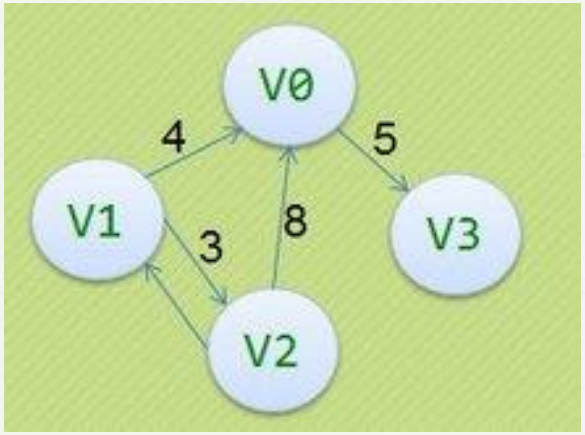

END

练 习



1. 有无向图如右，给出其邻接矩阵存储结构。

2. 有有向加权图如下，给出其邻接表、逆邻接表存储结构。



3. 假设无向图采用邻接矩阵存储，设计算法删除一条边(u,v)。
   如果存储结构是邻接表呢？

4. 假设有向图采用邻接表存储，设计算法插入一条边(u,v)。
   如果存储结构是邻接矩阵呢？

练 习

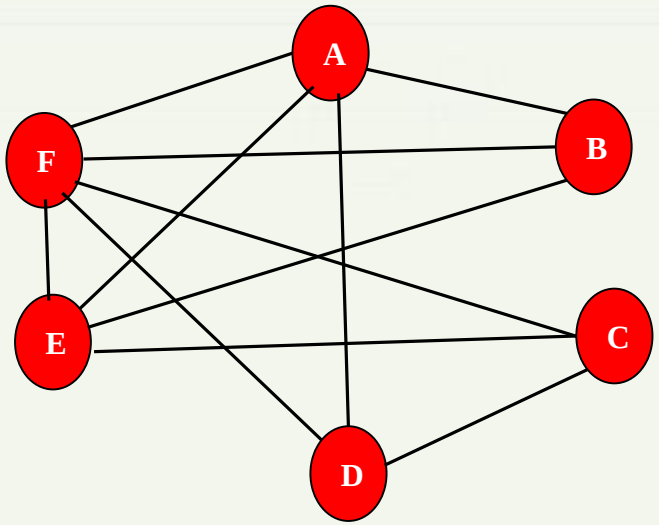
5. 有无向图如下，请完成：

（1）求出各个顶点的度；

（2）给出其邻接矩阵和邻接表存储结构；

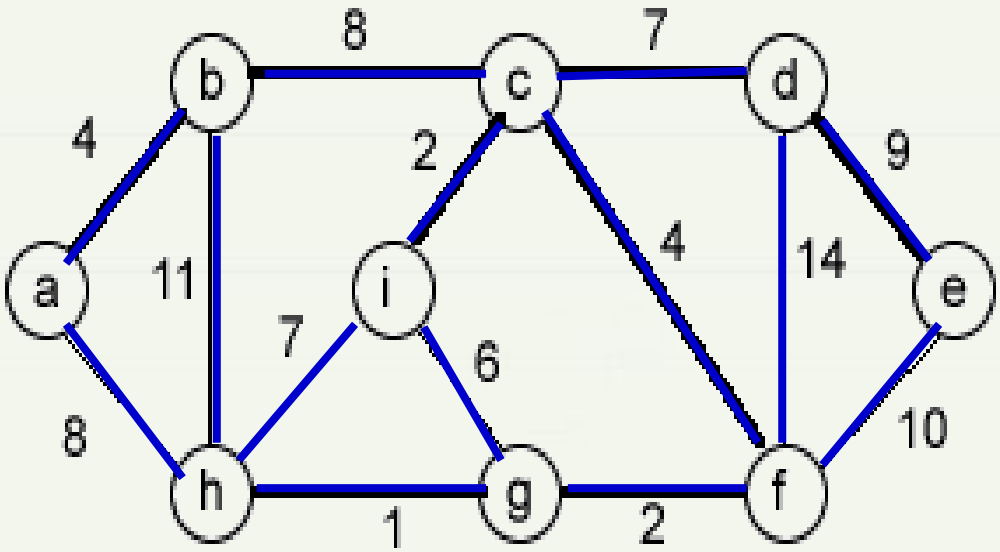
（3）依据你给出的邻接表存储结构，写出从A开始的深度遍历序列和从F开始的广度遍历序列；

（4）画出(3)得到的深度和广度生成树；



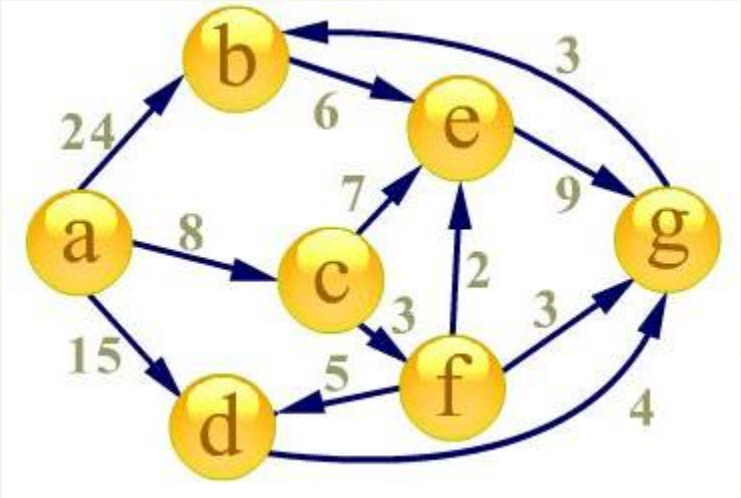
练 习

6. 有无向加权图如下，请分别用PRIM和KRUSKAL方法构造出其最小生成树。

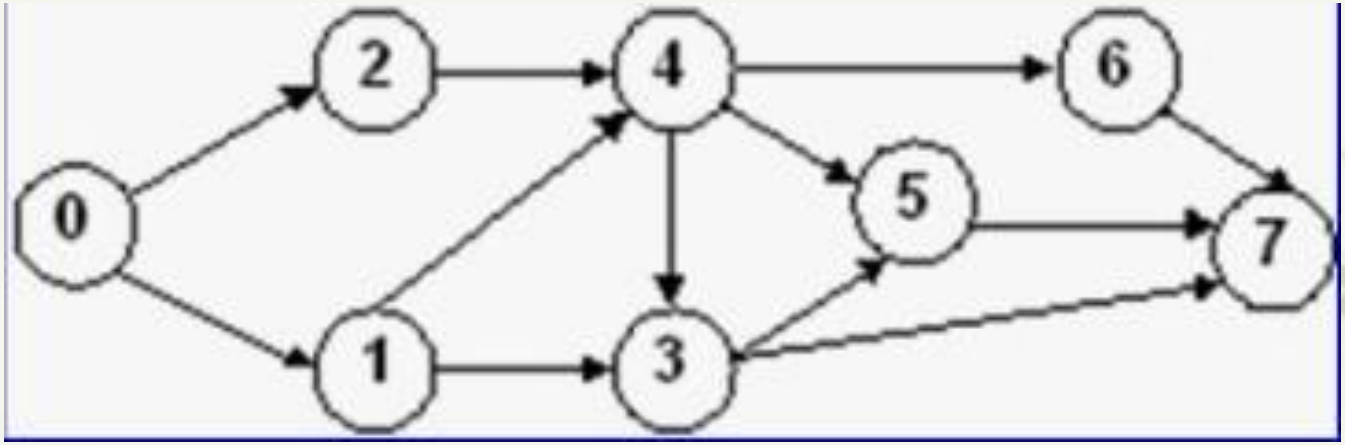


练 习

7. 求下图顶点a到其余各个顶点的最短路径

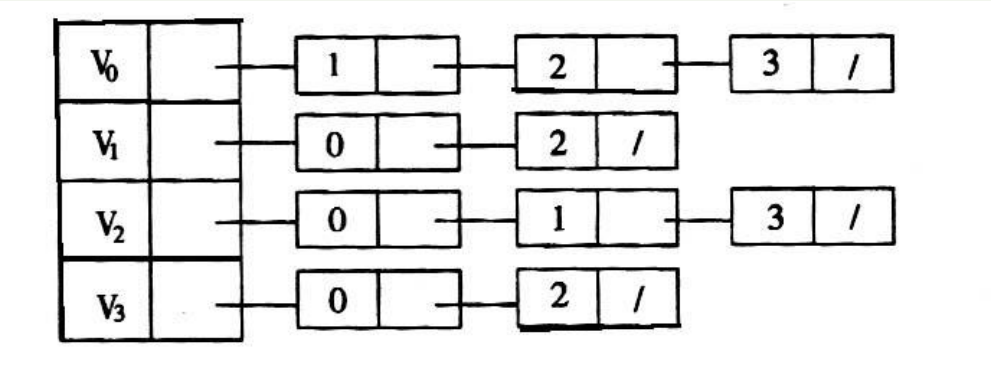


8. 写出下面AOV的拓扑序列，判断有没有环。



练 习

9. 已知图的邻接表如下所示，根据算法，写出从顶点v_0出发按广度优先遍历的顶点序列。



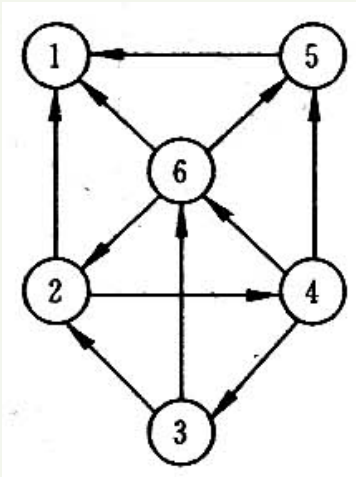
10. 已知如右所示的有向图，请给出该图的：

①每个顶点的入/出度；

②邻接矩阵；

③邻接表；

④逆邻接表。



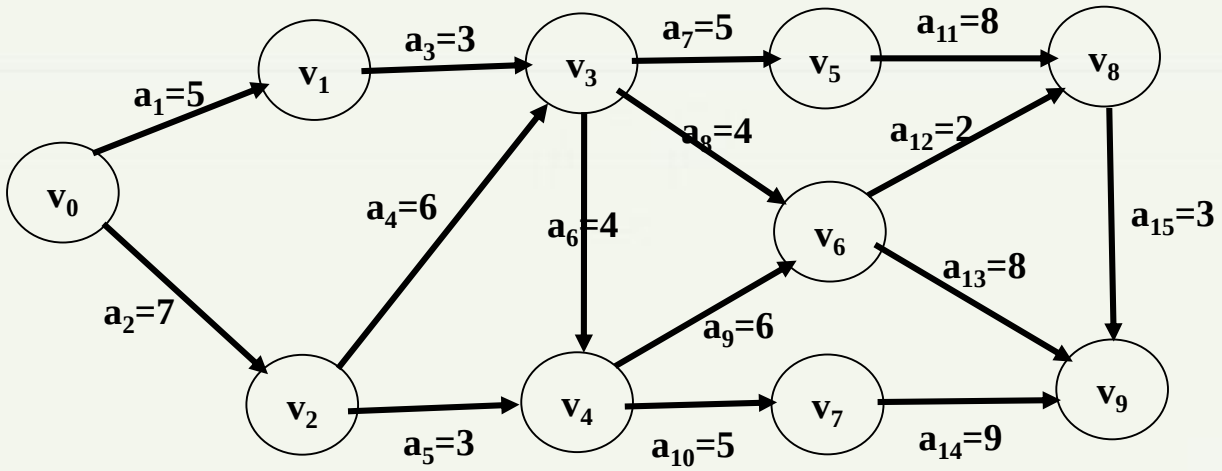
练 习

11. 对如下AOE网络，请完成：

（1）计算出每个事件的最早、最晚发生时间；

（2）计算出每个活动的最早、最迟开始时间；

（3）求出关键活动，找出关键路径；



练 习

12. 对于无向图（加权、不加权），请完成：

（1）定义出其邻接矩阵存储结构；

（2）写算法插入一条边；

（3）写算法删除一条边；

13. 对于有向图，请完成：

（1）定义出其邻接表存储结构；

（2）写算法求出每个顶点的出度；

（3）写算法求出每个顶点的入度；

14. 请写算法建立一个图（创建图）；