第八章　图

- 图的基本概念
- 图的存储结构
- 图的遍历与求图的连通分量
- 生成树与最小（代价）生成树
- 最短路径
- 拓扑排序
- 关键路径
- 本章小结

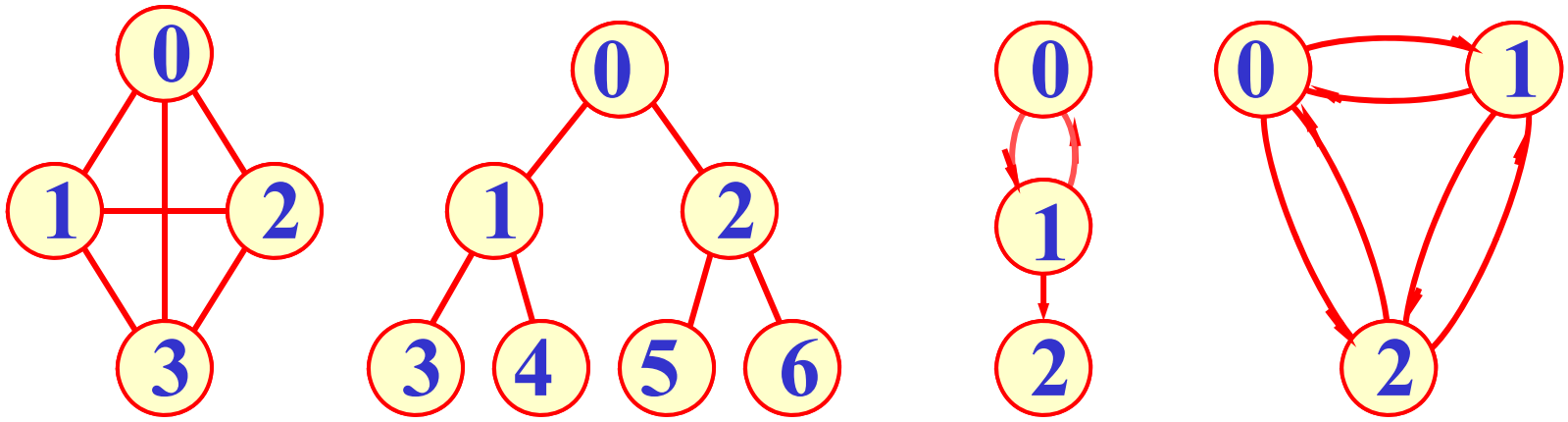
8.1 图的基本概念

- 图定义　图是由顶点集合(vertex)及顶点间的关系集合组成的一种数据结构：

  Graph＝(V, E)

  其中，**V={x|x∈某个数据对象}**
  是顶点的有穷非空集合；
  E={(x, y)|x, y∈V}
  或　**E={<x, y>|x, y∈V && Path(x, y)}**
  是顶点之间关系的有穷集合，也叫做边(edge)集合。**Path(x, y)**表示从**x**到 **y**的一条单向通路，它是有方向的。

- 有向图与无向图　在有向图中，顶点对<x, y>是有序的。在无向图中，顶点对<x, y>是无序的。
- 完全图　若有n个顶点的无向图有n(n-1)/2条边，则此图为完全无向图。有n个顶点的有向图有n(n-1)条边，则此图为完全有向图。



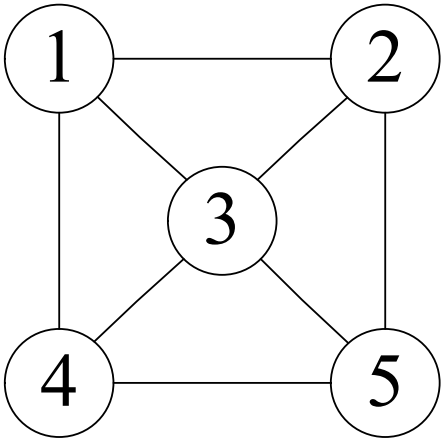


图8-1 无向图G_1

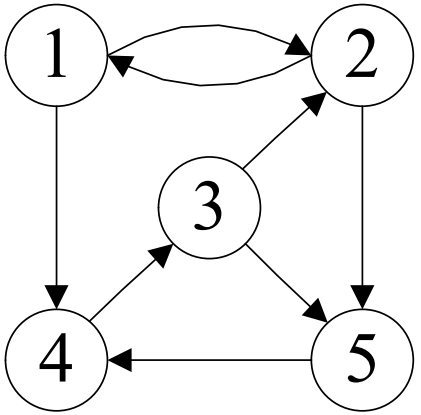
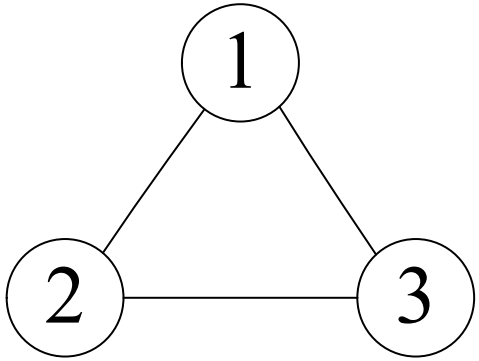


图8-2 有向图G_2



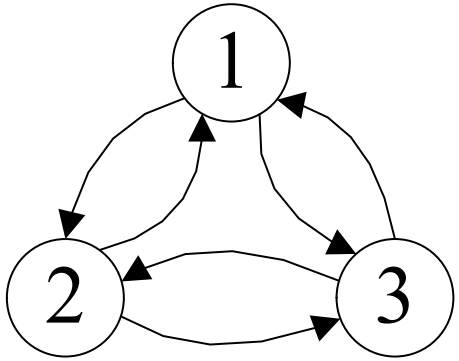
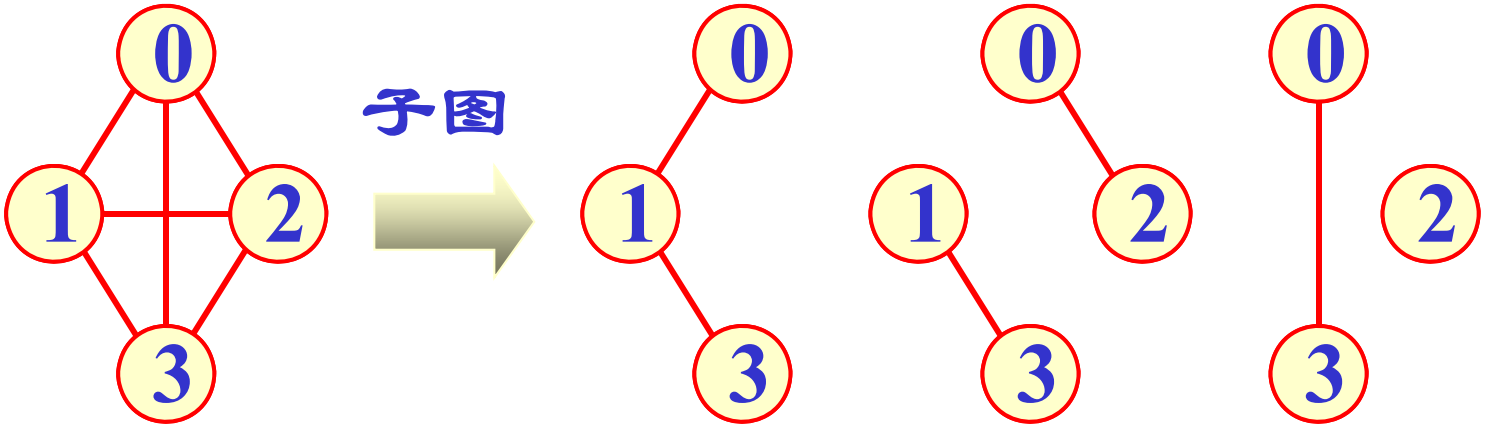


图8-5 完全图

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



- 权　某些图的边具有与它相关的数，称之为权。这种带权图叫做网络。

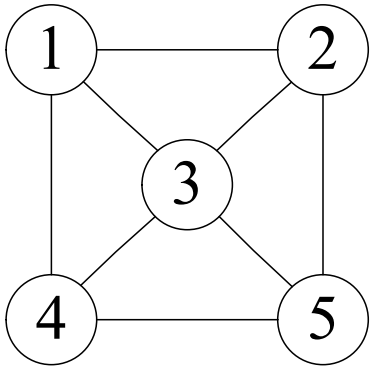
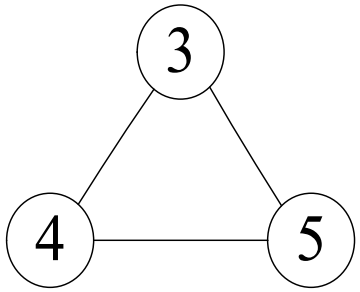
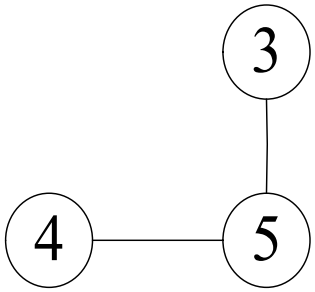


图8-1 无向图G_1





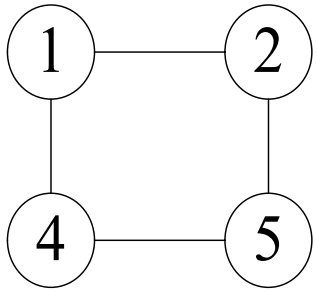


图8-3 图G_1的某些子图

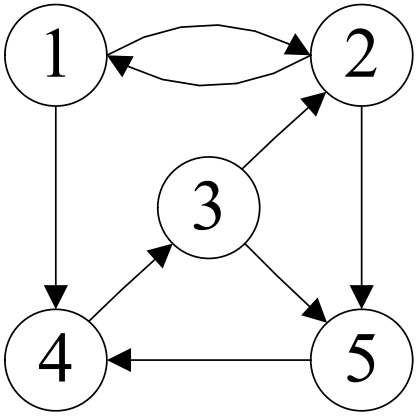
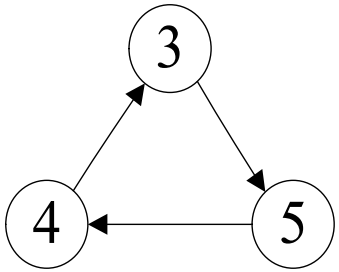
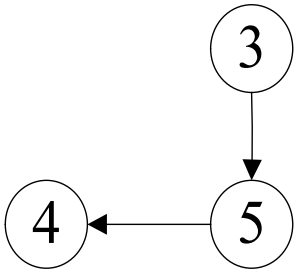


图8-2 有向图G_2





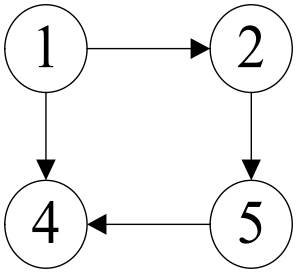
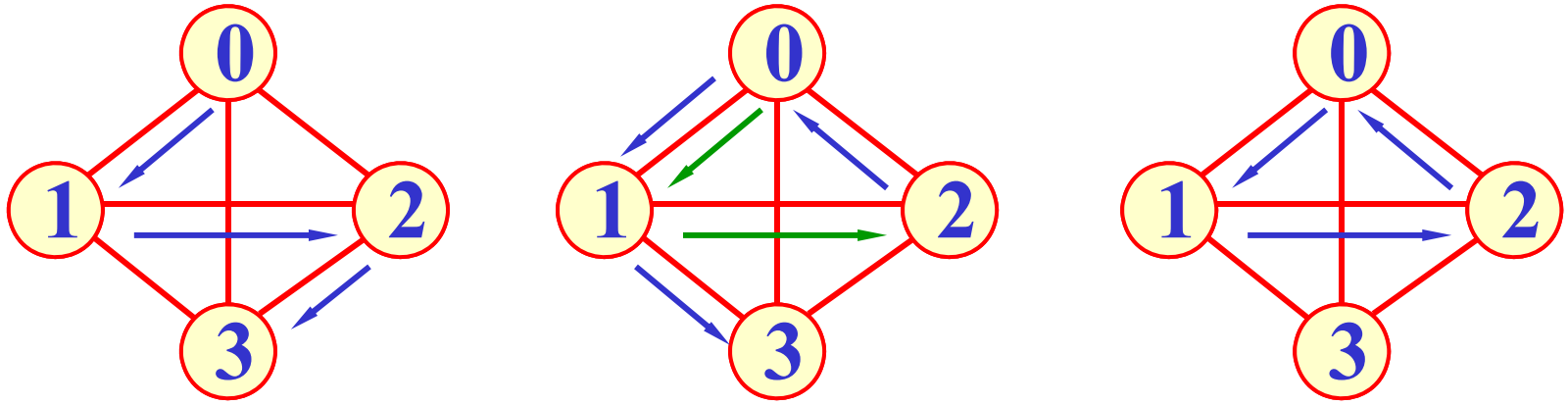


图8-4 图G_2的某些子图

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

- 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，…，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。



- 连通图与连通分量　在无向图中，若从顶点v_1到顶点v_2有路径，则称顶点v_1与v_2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。
- 生成树　一个连通图的生成树是其极小连通子图，在n个顶点的情形下，有n-1条边。

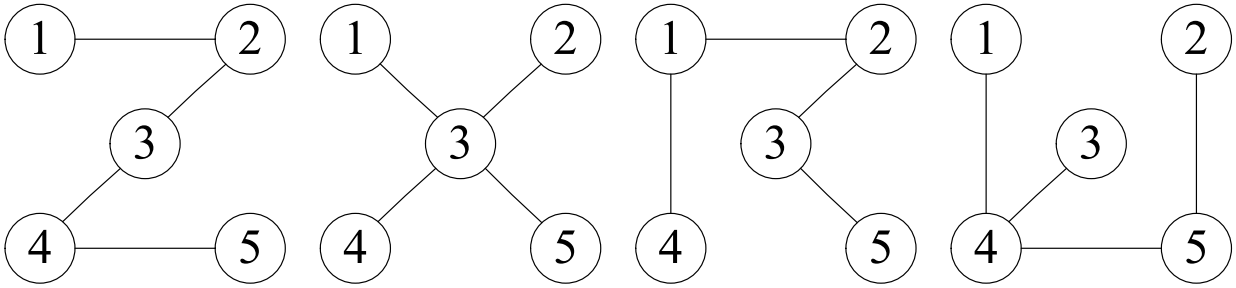


图8-6 连通的无向图

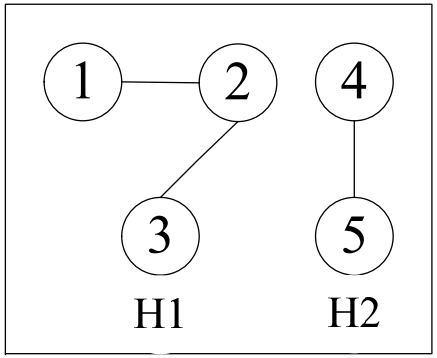


图8-7 具有两个分量的无向图

- 强连通图与强连通分量　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。非强连通图的极大强连通子图叫做强连通分量。

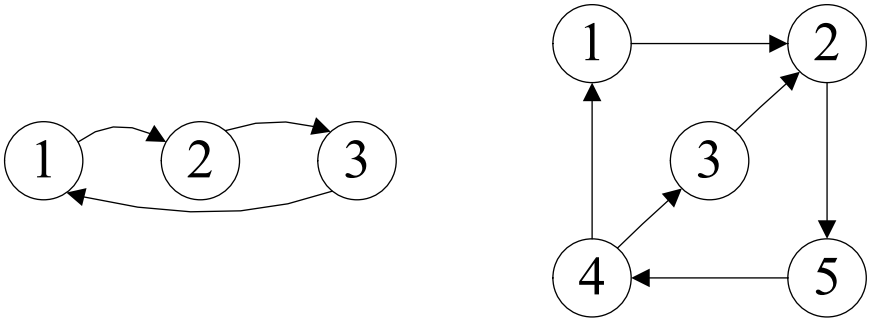


图8-8 强连通的有向图

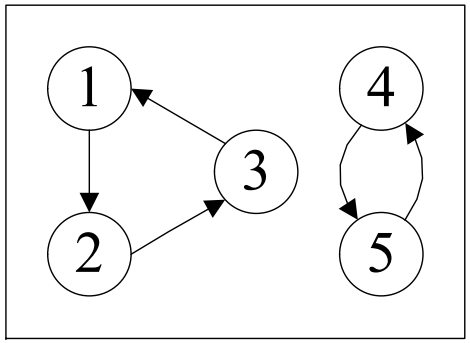


图8-9 具有两个强连通分量的有向图

- 弱连通图与弱连通分量 如果有向图G的每对顶点v和w，有一个由不同顶点组成的顶点序列<v0, v1, …, vk>，其中v0=v, vk=w且<vi, vi+1>∈E(G)或者<vi+1, vi>∈E(G) (0≤i<k)，则称有向图G是弱连通的。非弱连通图的极大弱连通子图叫做弱连通分量。

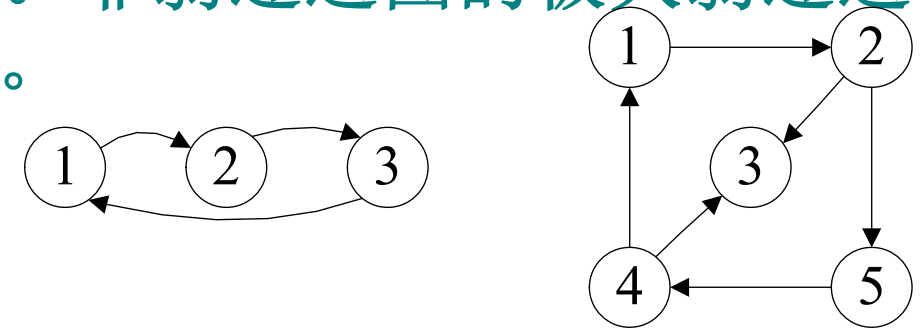


图8-10 弱连通的有向图

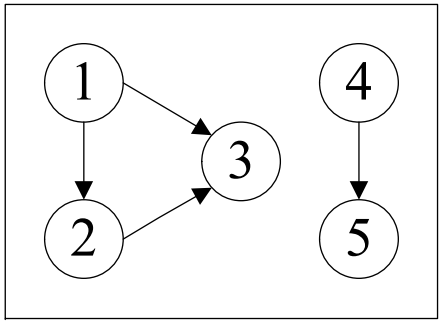


图8-11 具有两个弱连通分量的有向图

图的抽象数据类型

```
template <class Type> class Graph {
   public:
       Graph( );
       void InsertVertex(Type &vertex);
       void InsertEdge(int v1, int v2, int weight);
       void RemoveVertex(int v);
       void RemoveEdge(int v1, int v2);
       bool IsEmpty( );
       Type GetWeight(int v1, int v2);
      int GetFirstNeighbor(int v);
      int GetNextNeighbor(int v1, int v2);
};
```

8.2 图的存储结构

邻接矩阵 (Adjacency Matrix)

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

$$\mathbf{A}.edge[i][j]=\begin{cases}1, & \text{如果}<i,j>\in E \text{ 或者 } (i,j)\in E\\ 0, & \text{否则}\end{cases}$$

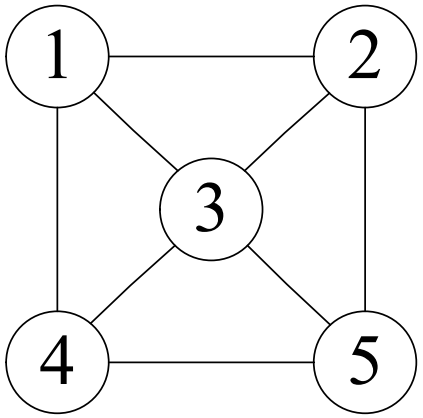


图8-1 无向图G_1

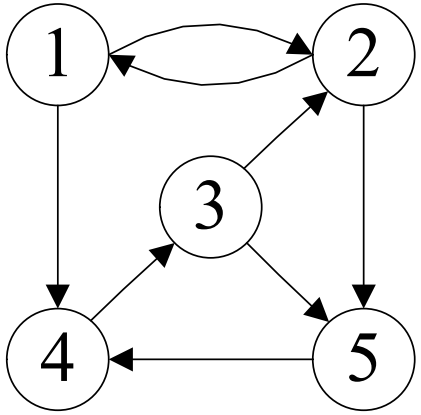
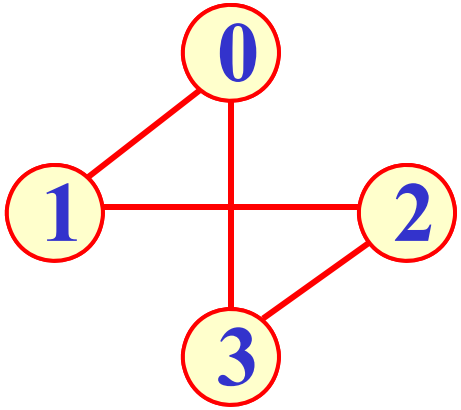


图8-2 有向图G_2

$$A_1=\begin{bmatrix}0&1&1&1&0\\1&0&1&0&1\\1&1&0&1&1\\1&0&1&0&1\\0&1&1&1&0\end{bmatrix}\qquad A_2=\begin{bmatrix}0&1&0&1&0\\1&0&0&0&1\\0&1&0&0&1\\0&0&1&0&1\\0&0&0&1&0\end{bmatrix}$$

图8-12 邻接矩阵



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

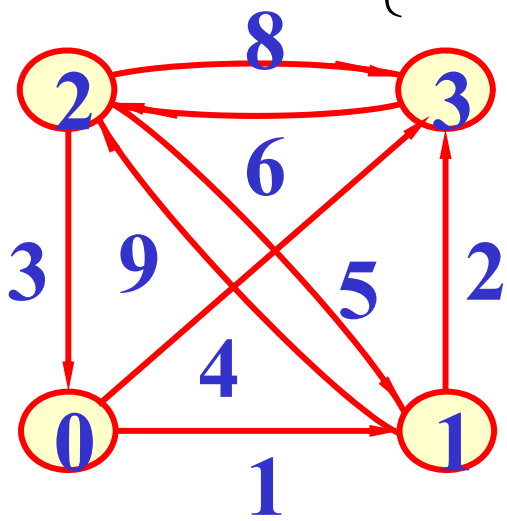
$$A.edge = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}$$

- 无向图的邻接矩阵是对称的；
- 有向图的邻接矩阵可能是不对称的。

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

网络的邻接矩阵

$$\mathbf{A.edge}[i][j]=\begin{cases}\mathbf{W}(i,j), & \text{若}i\neq j\text{且}<i,j>\in \mathbf{E}\text{或}(i,j)\in \mathbf{E}\\ \infty, & \text{若}i\neq j\text{且}<i,j>\notin \mathbf{E}\text{或}(i,j)\notin \mathbf{E}\\ 0, & \text{若}i==j\end{cases}$$



$$\mathbf{A.edge}=\begin{bmatrix}0 & 1 & \infty & 4\\ \infty & 0 & 9 & 2\\ 3 & 5 & 0 & 8\\ \infty & \infty & 6 & 0\end{bmatrix}$$

程序8-1 用邻接矩阵表示的图的类的定义

```
const int EdgesMaxNum=100;  //最大边数
const int VerticesMaxNum=20;  //最大顶点数
template <class NameType, class DistType> class Graph
{  //图的类定义
   private:
      SeqList <NameType> VerticesList(VerticesMaxNum);
      //顶点表
      DistType A[VerticesMaxNum][VerticesMaxNum];
      //邻接矩阵
      int EdgesCurrentNum;  //当前边数
   public:
      Graph(int size=VerticesMaxNum);  //构造函数
      int GraphEmpty( ) const { return VerticesList.IsEmpty( ); }
      //判断图为空否
```

```
int GraphFull( ) const  //判断图为满否
{ return VerticeList.IsFull( ) ||
                    EdgesCurrentNum==EdgesMaxNum; }
int VerticesNumber( ) { return VerticesList.*p_n+1; }
//返回当前顶点数
int EdgesNumber( ) { return EdgesCurrentNumber; }
//返回当前边数
DistType GetWeight(int v1, int v2);
//给出以顶点v1和v2为两端点的边上的权值
int GetFirstNeighborVer(int v);
//给出顶点v的第一个邻接顶点的位置
int GetNextNeighborVer(int v1, int v2);
//给出顶点v1的某邻接顶点v2的下一个邻接顶点
```

```
    void InsertVertex(const NameType &vertex);
    //插入新顶点vertex
    void InsertEdge(int v1, int v2, DistType weight);
    //插入权值为weight一条新边(v1, v2)
    void RemoveVertex(int vertex);
    //删除顶点vertex和所有与它相关联的边
    void RemoveEdge(int v1, int v2);
    //在图中删去边(v1, v2)
};
```

```
template <class NameType, class DistType> Graph
<NameType, DistType> :: Graph(int size)
{  //构造函数
    for (int i=0; i<size; i++)  //邻接矩阵初始化
        for (int j=0; j<size; j++)
            A[i][j]=0;
    EdgesCurrentNum=0;  //图中当前边数初始化
}
```

```
template <class NameType, class DistType>
DistType Graph <NameType, DistType> ::
GetWeight(int v1, int v2)
{  //给出以顶点v1和v2为两端点的边上的权值
      if (v1!=-1 && v2!=-1)
         return A[v1][v2];
      else return 0;  //带权图中权值为0，表示无权值
}
```

```
template <class NameType, class DistType>
int Graph <NameType, DistType> ::
GetFirstNeighborVer(int 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
      if (v!=-1)
      {
          for (int col=0; col<=VerticesList.*p_n; col++)
              if (A[v][col]>0 && A[v][col]<max)
                  return col;
      }
      return -1;
}
```

```
template <class NameType, class DistType>
int Graph <NameType, DistType> ::
GetNextNeighborVer(int v1, int v2)
{  //给出顶点v1的某邻接顶点v2的下一个邻接顶点
   if (v1!=-1 && v2!=-1)
   {
        for(int col=v2+1; col<=VerticesList.*p_n; col++)
            if (A[v1][col]>0 && A[v1][col]<max)
                return col;
   }
   return -1;
}
```

用邻接矩阵表示的图的类的定义

```
const int MaxValue=……;
const int MaxEdges=50;
const int MaxVertices=10;
template <class Type> class Graph {
  private:
    Type VerticesList[MaxVertices];
    float Edge[MaxVertices][MaxVertices];
    int numberEdges;
    int numberVertices;
    int GetVertexPos(const Type vertex) {
        for (int i=0; i<numberVertices; i++)
            if (VerticesList[i]==Vertex)  return i;
        return -1;
    }
```

```
public:
  Graph(int sz=MaxEdges);
  bool GraphEmpty( ) const
      { return numberVertices==0 || numberEdges==0; }
  bool GraphFull( ) const
      { return numberVertices==MaxVertices ||
                  numberEdges==MaxEdges; }
  int NumberOfVertices( )
      { return numberVertices; }
  int NumberOfEdges( )
      { return numberEdges; }
```

```
    Type GetValue(int i)
        { return i>=0 && i<=numberVertices
                                  ? VerticesList[i] : NULL; }
    float GetWeight(int u,  int v) {
      if (u!=-1 && v!=-1)  return Edge[u][v];
      else return 0;
    }
    int GetFirstNeighbor(int v);
    int GetNextNeighbor(int v, int w);
    void InsertVertex(const Type vertex);
    void InsertEdge(int u,  int v, float weight);
    void RemoveVertex(int v);
    void RemoveEdge(int u, int v);
};
```

邻接矩阵实现的部分图操作

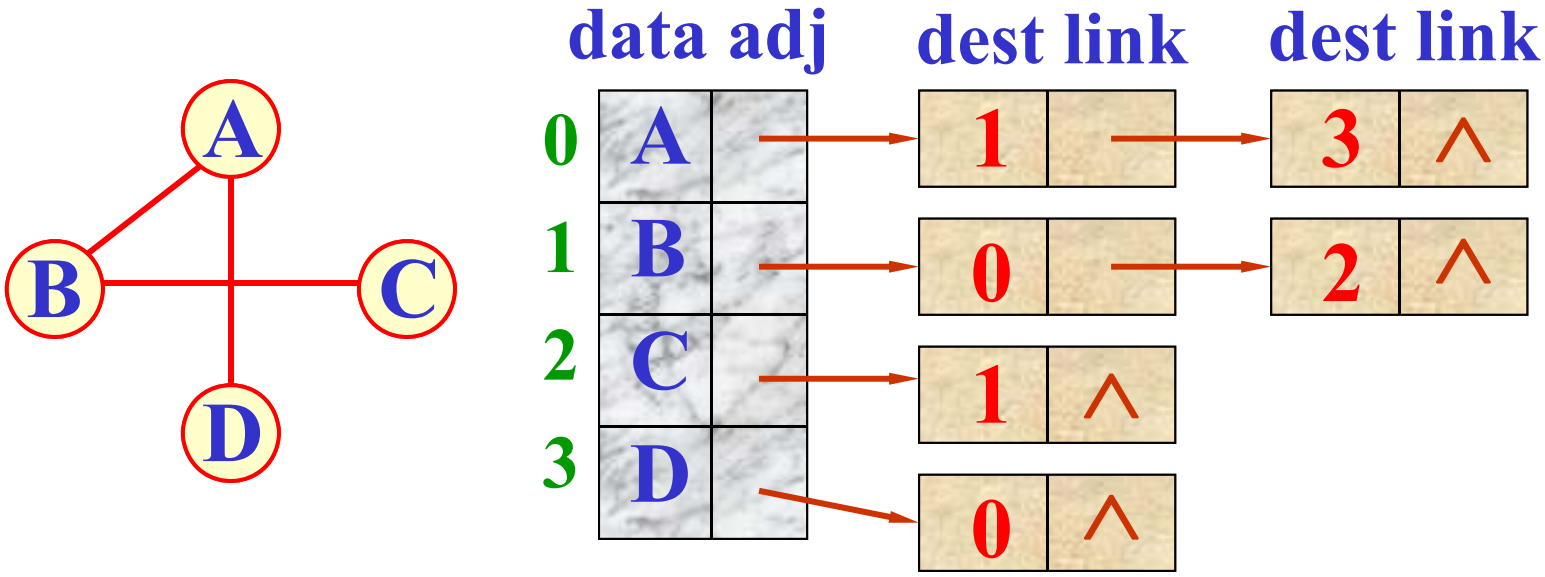
```
template <class Type> Graph <Type> ::
Graph(int sz) {  //构造函数
    for (int i=0; i<sz; i++)
        for(int j=0; j<sz; j++)
            Edge[i][j]=0;
    NumberEdges=0;
}
```

```
template <class Type> int Graph <Type> ::
GetFirstNeighbor(const int v) {
//给出顶点位置为v的第一个邻接顶点的位置
  if (v!=-1) {
    for (int j=0; j<numberVertices; j++)
        if (Edge[v][j]>0 &&
            Edge[v][j]<MaxValue)
          return j;
  }
  else return -1;
}
```

```
template <class Type> int Graph <Type> ::
GetNextNeighbor(int v,  int w) {
//给出顶点v的某邻接顶点w的下一个邻接顶点
    if (v!=-1 && w!=-1) {
      for (int j=w+1; j<numberVertices; j++)
        if (Edge[v][j]>0 &&
                Edge[v][j]<MaxValue)
            return j;
    }
    return -1;
}
```

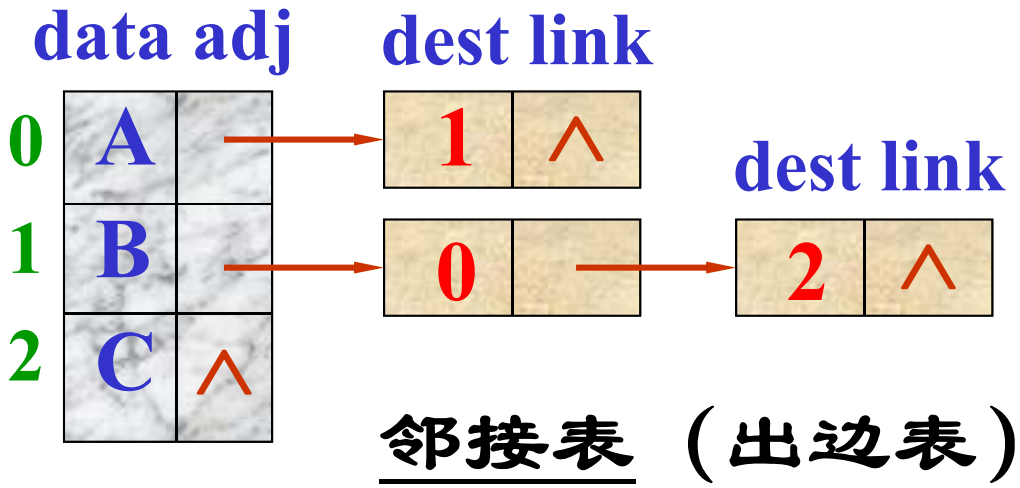
邻接表 (Adjacency List)

- 无向图的邻接表

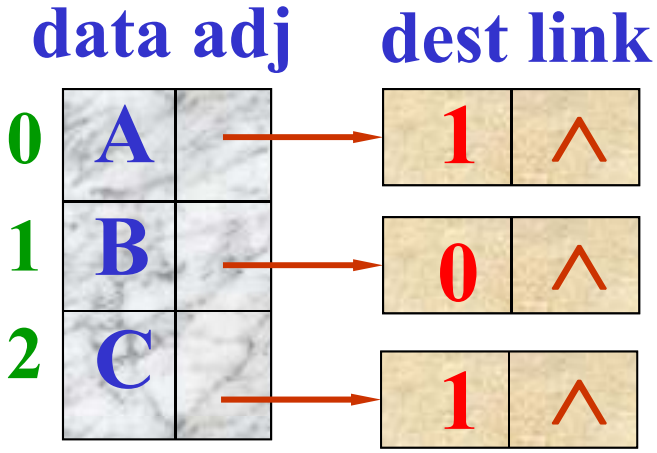


同一个顶点发出的边链接在同一个边链表中，每一个链结点代表一条边（边结点），结点中有另一顶点的下标**dest**和指针**link**。

有向图的邻接表和逆邻接表

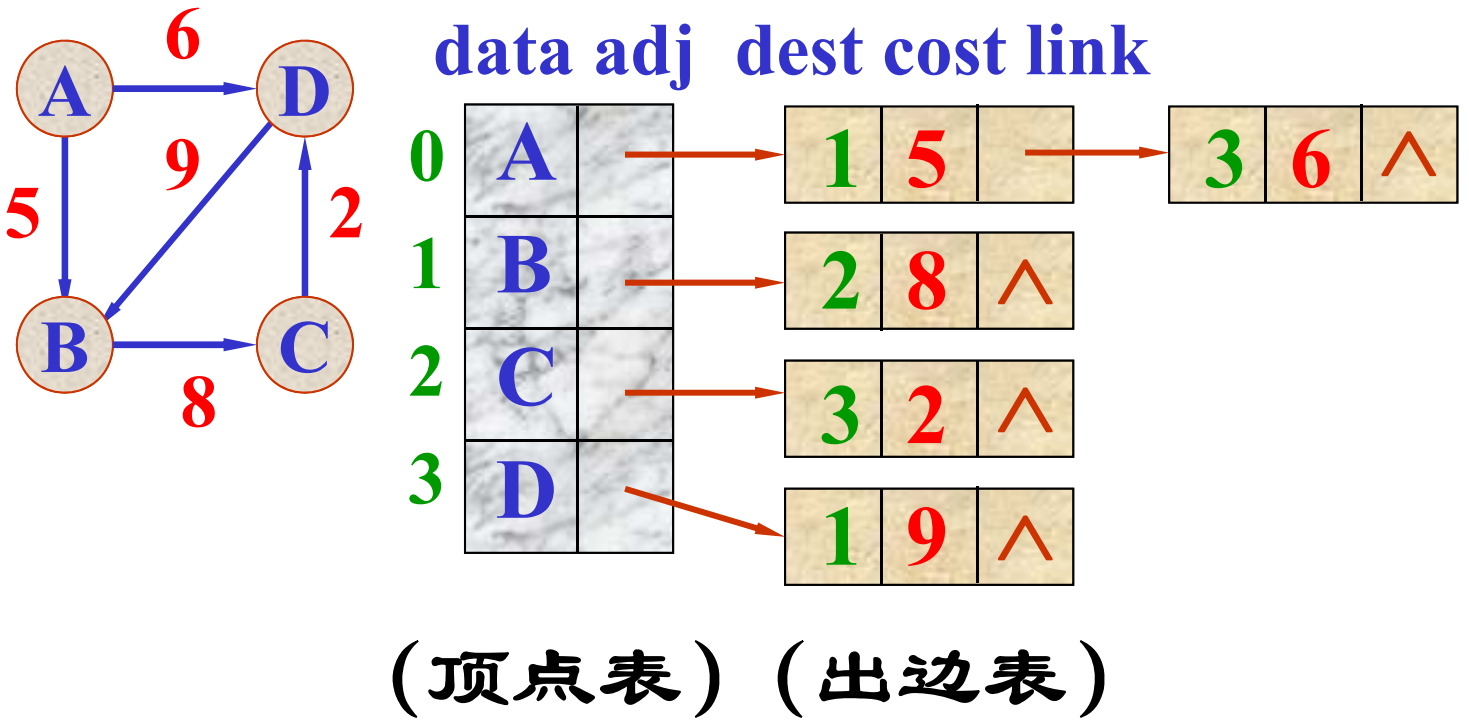


邻接表（出边表）



逆邻接表（入边表）

- 网络（带权图）的邻接表



（顶点表）（出边表）

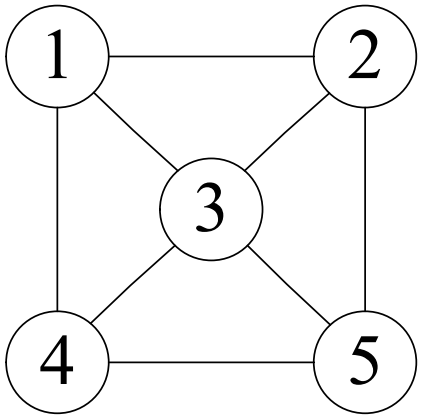


图8-1 无向图G_1

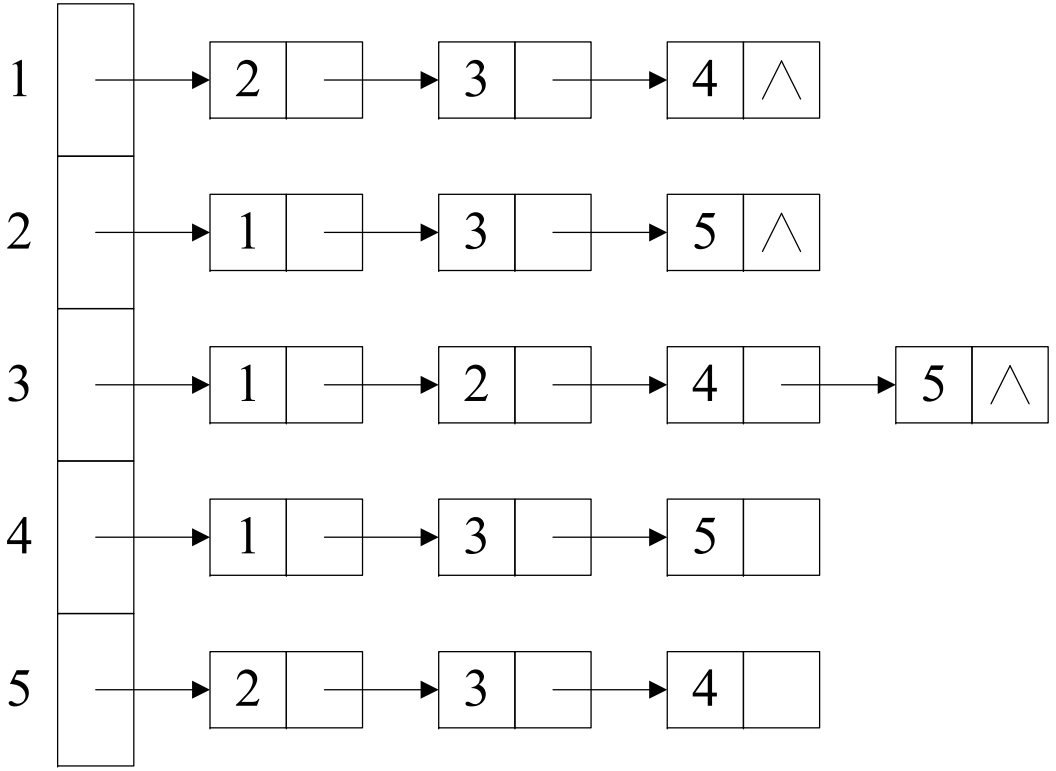


图8-13 无向图G_1的邻接表

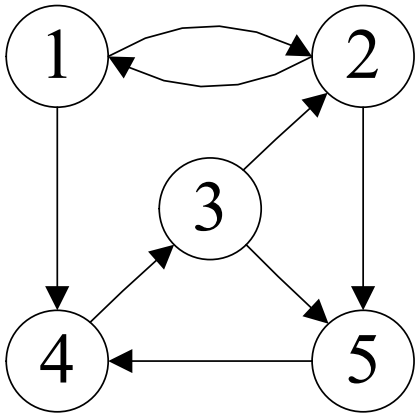


图8-2 有向图G_2

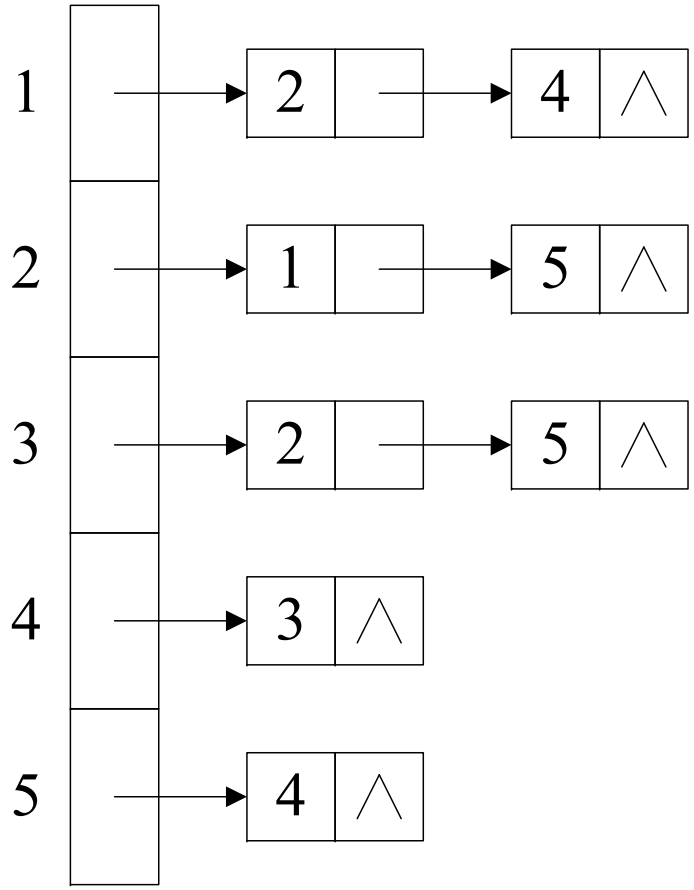


图8-14 有向图G_2的邻接表

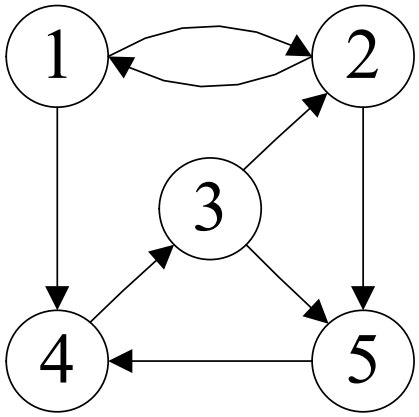


图8-2 有向图G_2

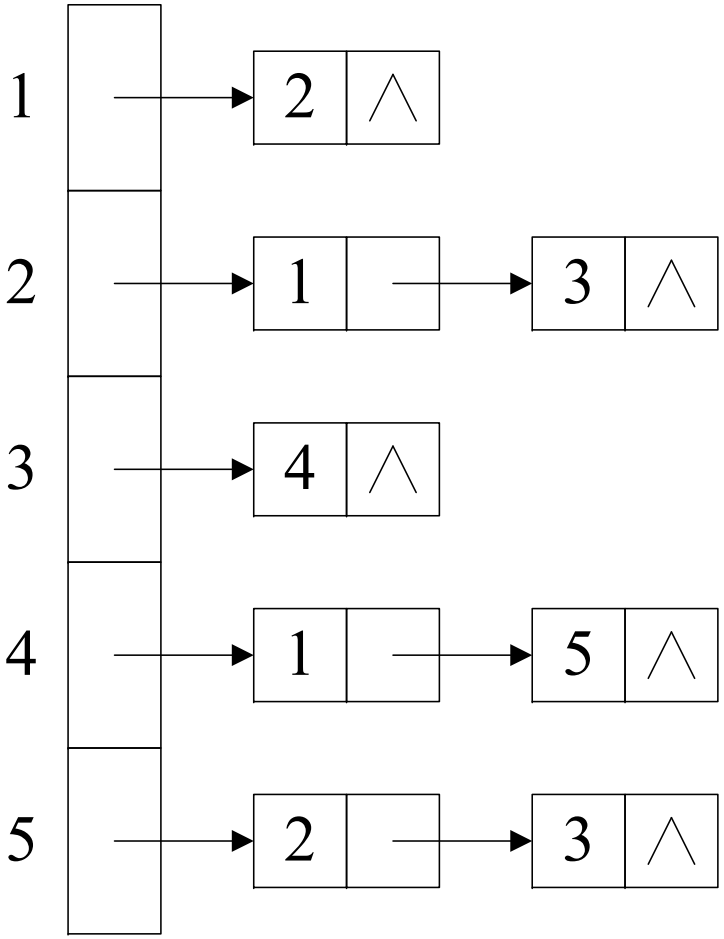


图8-15 有向图G_2的逆邻接表

- 带权图的边结点中保存该边上的权值**cost**。
- 顶点**i**的边链表的表头指针**adj**在顶点表的下标为**i**的顶点记录中，该记录还保存了该顶点的其它信息。
- 在邻接表的边链表中，各个边结点的链入顺序任意，视边结点输入次序而定。
- 设图中有**n**个顶点，**e**条边，则用邻接表表示无向图时，需要**n**个顶点结点，**2e**个边结点；用邻接表表示有向图时，若不考虑逆邻接表，只需**n**个顶点结点，**e**个边结点。

程序8-2 用邻接表表示的图的类的定义

```
const int DefaultSize=10;  //缺省顶点个数
template <class DistType> class Graph;  //图类的前向引用声明
template <class DistType> class EdgeNode {  //边结点的类定义
        friend class Graph <NameType, DistType>;
        int dest;  //边的另一顶点位置
        DistType cost;  //边上的权值
        EdgeNode <DistType> *link;  //边结点后继指针
    public:
        EdgeNode( ) { }  //构造函数
        EdgeNode(int d, DistType c) : dest(d), cost(c), link(NULL)
        { }  //构造函数
};
```

```
template <class NameType, class DistType> class VertexNode
{  //顶点结点的类定义
      friend class EdgeNode <DistType>;
      friend class Graph <NameType, DistType>;
      NameType data;  //顶点名字
      EdgeNode <DistType> *adj;  //出边表的头指针
};
template <class NameType, class DistType> class Graph {
//图的类定义
   private:
      Vertex <NameType, DistType> *NodeTable;
      //顶点表 (各边链表的头结点)
      int VerticesNum;  //当前顶点数
      int VerticesMaxNum;  //最大顶点数
      int EdgesNum;  //当前边数
      int GetVertexPos(const NameType &vertex);
      //当前顶点vertex在图中的位置
```

```
public:
    Graph(int size);  //构造函数
    ~Graph( );  //析构函数
    int GraphEmpty( ) const { return VerticesNum==0; }
    //判断图空否
    int GraphFull( ) const { return VerticesNum==
                                    VerticesMaxNum; }
    //判断图满否
    int VerteicesNumber( ) { return VerticesNum; }
    //返回图的顶点数
    int EdgesNumber( ) { return EdgeNum; }  //返回图的边数
    DistType GetWeight(int v1, int v2);  //返回边上的权值
    int GetFirstNeighbor(int v);  //取顶点v的第一个邻接顶点
```

```
    int GetNextNeighbor(int v1, int v2);
    //取顶点v1的某邻接顶点v2的下一个邻接顶点
    void InsertVertex(const NameType Vertex);
    //在图中插入一个新顶点
    void InserEdge(int v1, int v2, DistType weight);
    //在图中插入一条新边
    void RemoveVertex(int v);  //在图中删除一个顶点
    void RemoveEdge(int v1, int v2);  //在图中删除一条边
};
```

```
template <class NameType, class DistType> Graph
<NameType, DistType> ::
Graph(int size=DefaultSize) : VerticesNum(0),
VerticesMaxNum(size), EdgesNum(0)
{  //构造函数
        int n, e, k, j;
        NameType name, tail, head;
        DistType weight;
        NodeTable=new VertexNode <NameType>
                              [VerticesMaxNum];  //创建顶点表数组
        cin>>n;  //输入边数
        for (i=0; i<e; i++)
        {
            cin>>tail>>head>>weight;
            //依次输入边的两个端点与权值信息
            k=GetVertexPos(tail);
            j=GetVertexPos(head);  //获取一条边两个端点的位置
            InserEdge(k, j, weight);  //插入一条边
        }
}
```

```
template <class NameType, class DistType> Graph
<NameType, DistType> :: ~Graph( )
{ //析构函数
    for (int i=0; i<VerticesNum; i++)
    { //删除各边链表中的顶点
        EdgeNode <DistType> *p=NodeTable[i].adj;
        while (p!=NULL)
        { //循环删除
            NodeTable[i].adj=p->link;
            delete p;
            p=NodeTable[i].adj;
        }
        delete [ ] NodeTable; //释放顶点表数组空间
    }
}
```

```
template <class NameType, class DistType> int Graph
<NameType, DistType> ::
GetVertexPos(const NameType vertex)
{ //给出顶点vertex在图中的位置
      for (int i=0; i<VerticesNum; i++)
            if (NodeTable[i].data==vertex)
               return i;
      return -1;
}
```

```
template <class NameType, class DistType> DistType Graph
<NameType, DistType> :: GetWeight(int v1, int v2)
{  //获取以v1与v2为两个端点的一条边的权值
   //若该边不存在于图中则返回权值0
   if (v1!=-1 && v2!=-1)
   {
       EdgeNode <DistType> *p=NodeTable[v1].adj;
       //边链表头指针
       while (p!=NULL)
          if (p->dest==v2)
             return p->cost;  //找到对应边，返回权值
          else p=p->link;  //否则找下一条边
    }
}
```

```
template <class nameType, class DistType> int Graph
<NameType, DistType> :: GetFirstNeighbor(int v)
{  //给出顶点v的第一个邻接顶点的位置
   //如果不存在则返回-1
    if (v!=-1)
    {  //v存在
        EdgeNode <DistType> *p=NodeTable[v].adj;
        //边链表头指针
        if (p!=NULL)
            return p->dest;
        //若第一个邻接顶点存在，则返回该边的另一个顶点
    }
    return -1; //若不存在，则返回-1
}
```

```
template <class NameType, class DistType> int Graph
<NameType, DistType> :: GetNextNeighbor(int v1, int v2)
{  //给出顶点v1的某邻接顶点v2的下一个邻接顶点的位置
   //若没有下一个邻接顶点则返回-1
   if (v1!=-1)
   {  //v1存在
      EdgeNode <DistType> *p=NodeType[v1].adj;
      //边链表头指针
      while (p!=NULL)
      {  //寻找某邻接顶点v2
         if (p->dest==v2 && p->link!=NULL)
            return p->link->dest;
         else p=p->link;
      }
    }
    return -1;
}
```

邻接表表示的图的类定义

```
#define DefaultSize  10
template <class Type> class Graph;
template <class Type> struct Edge {  //边结点
friend class Graph <Type>;
    int dest;  //目标顶点下标
    float cost;  //边上的权值
    Edge *link;  //下一边链接指针
    Edge( ) { }  //构造函数
    Edge(int D, float C) :
              dest(D), cost(C), link(NULL) { }
     bool operator != (Edge &E)
          const { return dest!=E.dest; }
};
```

```
template <class Type> struct Vertex {  //顶点
friend class Graph <Type>;
    Type data;  //顶点数据
    Edge *adj;  //边链表头指针
};
```

```
template <class Type> class Graph {  //图类
private:
    Vertex <Type> *NodeTable;  //顶点表
    int numberVertices;  //当前顶点个数
    int MaxVertices;  //最大顶点个数
    int numberEdges;  //当前边数
    int GetVertexPos(const Type vertex);
public:
    Graph(int sz);
    ~Graph( );
    bool GraphEmpty( ) const
        { return numberVertices==0 ||
                        numberEdges==0; }
```

```
bool GraphFull( ) const
   { return numberVertices == MaxVertices; }
Type GetValue(int i)
   { return i>=0 && i<numberVertices ?
                NodeTable[i].data : NULL; }
int NumberOfVertices( )
   { return numberVertices; }
int NumberOfEdges( )
   { return numberEdges; }
void InsertVertex(Type vertex);
void RemoveVertex(int v);
```

```
    void InsertEdge(int u, int v, float weight);
    void RemoveEdge(int u, int v);
    float GetWeight(int u,  int v);
    int GetFirstNeighbor(int v);
    int GetNextNeighbor(int v,  int w);
};
```

邻接表的构造函数和析构函数

```
template <class Type> Graph <Type> ::
Graph(int sz=DefaultSize) : MaxVertices(sz) {
    int n, e, k, i, j;   Type name, tail, head;
    float weight;
    NodeTable=new Vertex <Type> [sz];
    //创建顶点表
    cin>>numberVertices;  //输入顶点个数
    for (int i=0; i<numberVertices; i++)
        { cin>>name;  InsertVertex(name); }
    //输入各顶点信息
```

```
    cin>>e;  //输入边条数
    for (i=0; i<e; i++) {  //逐条边输入
      cin>>tail>>head>>weight;
      k=GetVertexPos(tail);
      j=GetVertexPos(head);
      InsertEdge(k, j, weight);  //插入边
    }
}
```

```
template <class Type> Graph <Type> ::
~Graph( ) {
    for (int i=0; i<numberVertices; i++) {
        Edge *p=NodeTable[i].adj;
        while (p!=NULL) {  //逐条边释放
            NodeTable[i].adj=p->link;
            delete p;
            p=NodeTable[i].adj;
        }
    }
    delete [ ] NodeTable;  //释放顶点表
}
```

邻接表部分成员函数的实现

```
template <class Type> int Graph <Type> ::
GetVertexPos(const Type vertex) {
//根据顶点名vertex查找它在邻接表中位置
    for (int i=0; i<numberVertices; i++)
      if (NodeTable[i].data==vertex)
          return i;
    return -1;
}
```

```
template <class Type> int Graph <Type> ::
GetFirstNeighbor(int v) {
//查找顶点v第一个邻接顶点在邻接表中位置
   if (v!=-1) {  //若顶点存在
      Edge *p=NodeTable[v].adj;
      if (p!=NULL)  return p->dest;
   }
   return -1;  //若顶点不存在
}
```

```
template <class Type> int Graph <Type> ::
GetNextNeighbor(int v,  int w) {
//查找顶点v在邻接顶点w后下一个邻接顶点
   if (v!=-1) {
      Edge *p=NodeTable[v].adj;
      while (p!=NULL) {
         if (p->dest==w && p->link!=NULL)
             return p->link->dest;
         //返回下一个邻接顶点在邻接表中位置
        else p=p->link;    }
   }
  return -1;  //没有查到下一个邻接顶点
}
```

```
template <class Type> float Graph <Type> ::
GetWeight(int u, int v) {
    if (u!=-1 && v!=-1) {
        Edge <Type> *p=NodeTable[u].adj;
        while (p!=NULL)
            if (p->dest==v)  return p->cost;
            else p=p->link;
    }
    return 0;
}
```

邻接多重表 (Adjacency Multilist)

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

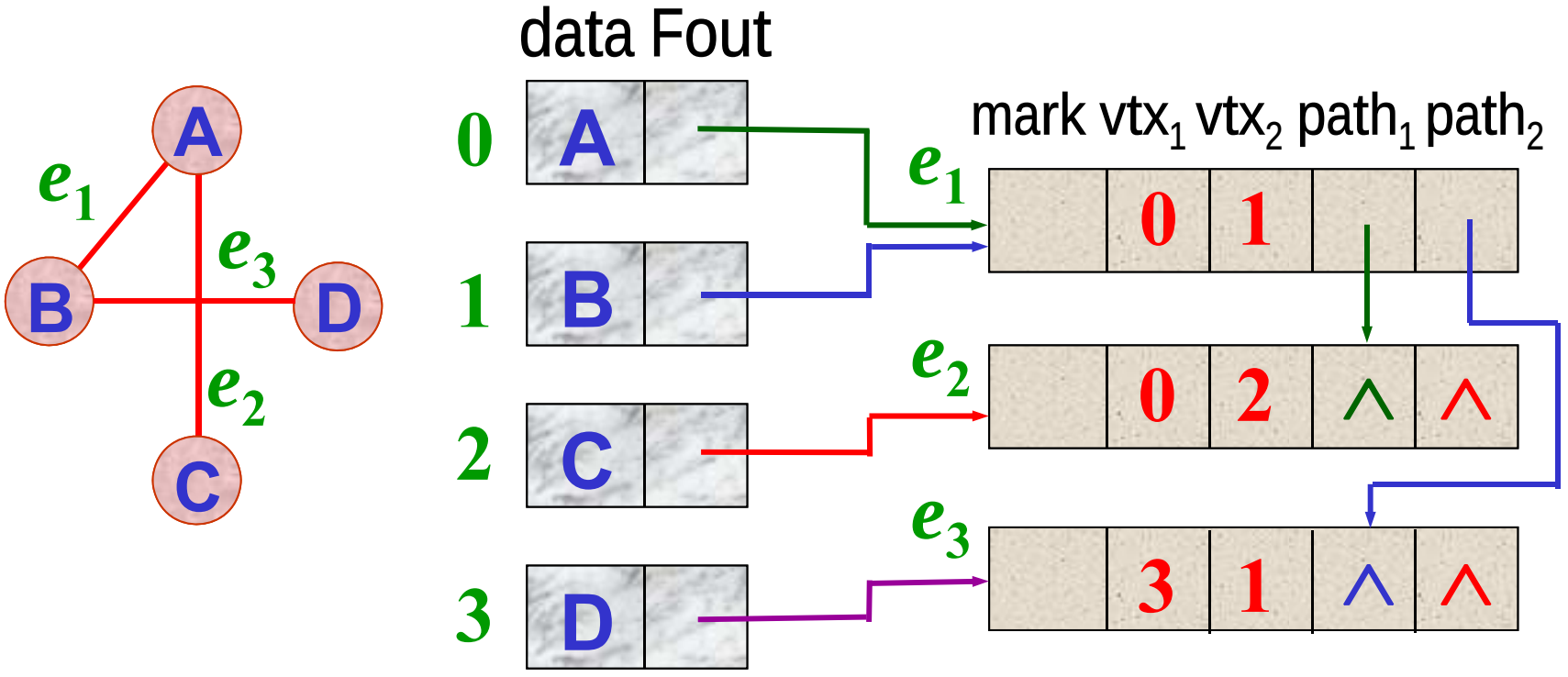
mark	vertex1	vertex2	path1	path2

其中，**mark**是记录是否处理过的标记；**vertex1**和**vertex2**是该边两顶点位置；**path1**域是链接指针，指向下一条依附顶点**vertex1**的边；**path2**是指向下一条依附顶点**vertex2**的边链接指针；需要时还可设置一个存放与该边相关的权值的域 **cost**。

- 顶点结点的结构

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

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



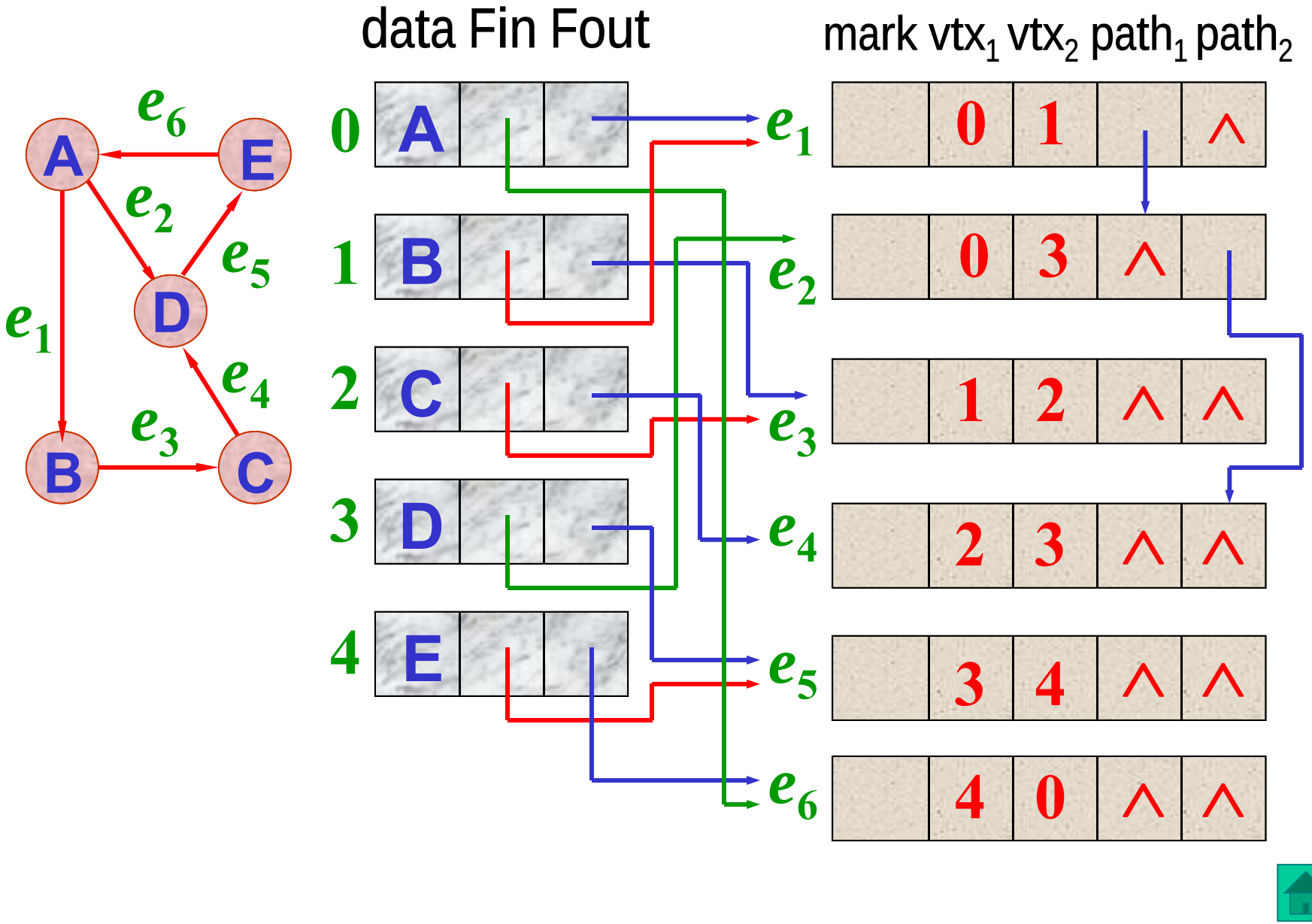
- 有向图的情形
  - 在用邻接表表示有向图时，有时需要同时使用邻接表和逆邻接表，用有向图的邻接多重表（十字链表）可把两个表结合起来表示。
    - 边结点的结构

mark	vertex1	vertex2	path1	path2

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

- 顶点结点的结构

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


data Fin Fout
mark vtx1 vtx2 path1 path2
A
B
C
D
E
e1
e2
e3
e4
e5
e6
0
1
2
3
4
^

8.3 图的遍历与求图的连通分量

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

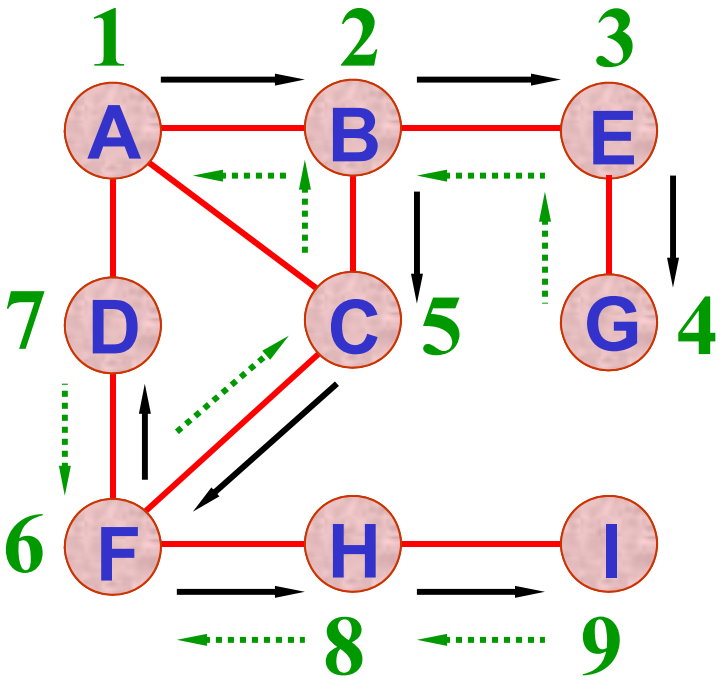
- 辅助数组Visited[ ]的初始状态为0，在图的遍历过程中，一旦某一个顶点i被访问，就立即让Visited[i]为1，防止它被多次访问。
- 图的遍历的分类
  - 深度优先搜索

    DFS (Depth First Search)
  - 广度优先搜索

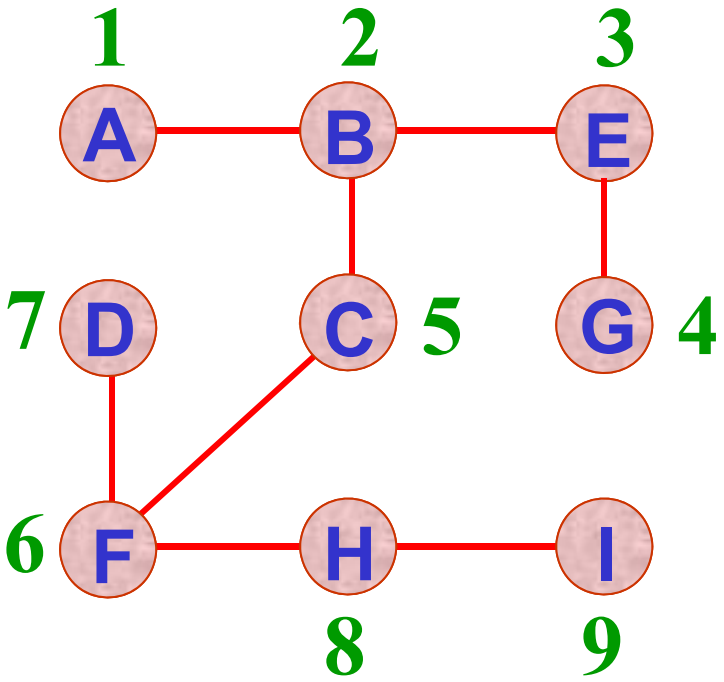
    BFS (Breadth First Search)

深度优先搜索DFS (Depth First Search)

- 深度优先搜索的示例



深度优先搜索过程



深度优先生成树

前进 ——→ 回退 ·····→

- **DFS**在访问图中某一起始顶点**v**后，由**v**出发，访问它的任一邻接顶点w_1；再从w_1出发，访问与w_1邻接但还没有访问过的顶点w_2；然后再从w_2出发，进行类似的访问，… 如此进行下去，直至到达所有的邻接顶点都被访问过的顶点**u**为止。

- 接着，退回一步，退到前一次刚访问过的顶点，看是否还有其它没有被访问的邻接顶点。如果有，则访问此顶点，之后再从此顶点出发，进行与前述类似的访问；如果没有，就再退回一步进行搜索。重复上述过程，直到连通图中所有顶点都被访问过为止。

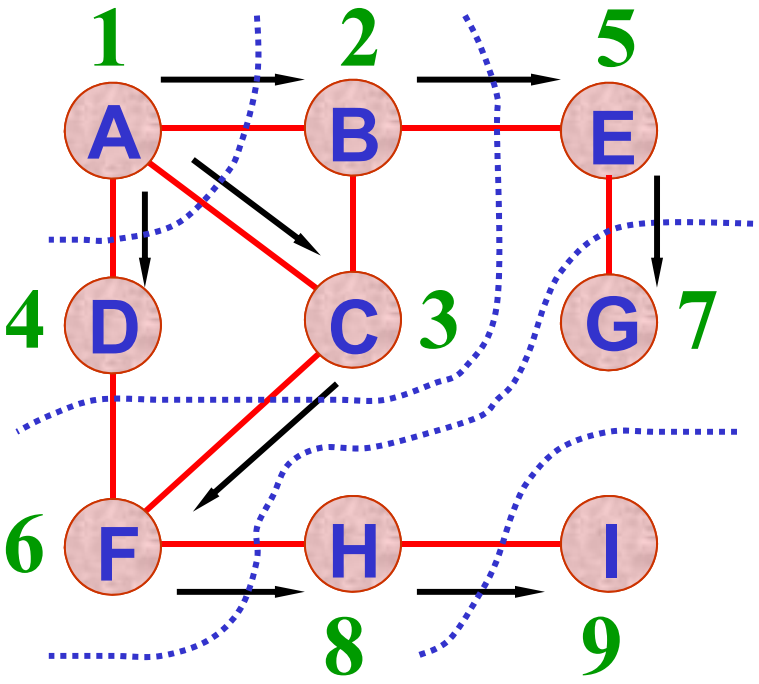
程序8-3 图的深度优先搜索算法

```
template <class NameType, class DistType> void
Graph <NameType, DistType> :: DFS( ) {
    int *Visited=new int [VerticesNumber( )];
    //创建辅助数组
    for (int i=0; i<VerticesNumber( ); i++)
        Visited[i]=0;  //辅助数组初始化
    DFS(0, Visited);
    //从顶点0开始进行深度优先搜索
    delete [ ] Visited;
}
```

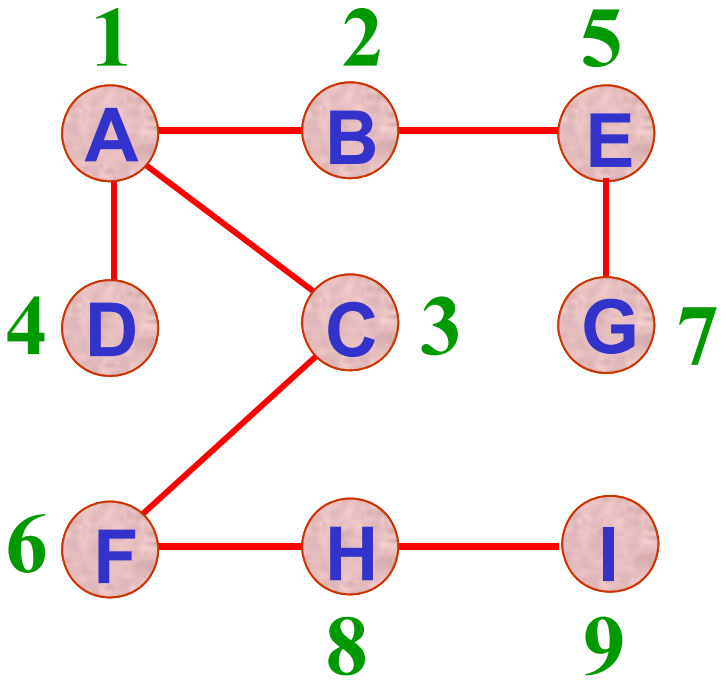
```
template <class NameType, class DistType> void Graph
<NameType, DistType> :: DFS(int v, int Visited[ ])
{  //从顶点v出发
   //按深度优先搜索次序访问所有尚未访问过的顶点
   //辅助数组Visited用于对已访问过的顶点作以访问标记
     cout<<v;  //输出访问顶点信息
     Visited[v]=1;  //该顶点访问标志修改为已访问过
     int w=GetFirstNeighbor(v);
     //寻找刚刚访问过顶点v的第一个邻接顶点w
     while (w!=-1)
     {  //若邻接顶点存在
         if (!Visited[w])  //若该邻接顶点未被访问过
            DFS(w, Visited);  //则从邻接顶点w开始进行递归访问
         w=GetNextNeighbor(v, w);
         //寻找顶点v的下一个邻接顶点
     }
}
```

广度优先搜索BFS (Breadth First Search)

- 广度优先搜索的示例



广度优先搜索过程



广度优先生成树

- **BFS**在访问了起始顶点v之后，由v出发，依次访问v的各个未被访问过的邻接顶点w_1, w_2, …, w_t，然后再顺序访问w_1, w_2, …, w_t的所有还未被访问过的邻接顶点。
- 再从这些访问过的顶点出发，再访问它们的所有还未被访问过的邻接顶点，…，如此做下去，直到图中所有顶点都被访问到为止。

- 广度优先搜索是一种分层的搜索过程，每向前走一步可能访问一批顶点，不像深度优先搜索那样有往回退的情况。因此，广度优先搜索不是一个递归的过程。

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

(1) 把队列置空。

(2) 打印出发顶点，置该顶点已被访问的标志。

(3) 让出发顶点进队。

(4) 若队列不空，则

(a) 取出队首中的顶点v。

(b) 在邻接表中，依次取得与顶点v邻接的各个顶点。

(i) 若当前取得的邻接顶点未被访问，则

① 输出该顶点，置该顶点已被访问的标志。

② 该顶点进队。

(ii) 取得下一个邻接顶点。

(c) 转(4)。

(5) 若队列空，则处理过程结束。

程序8-3 图的广度优先搜索算法

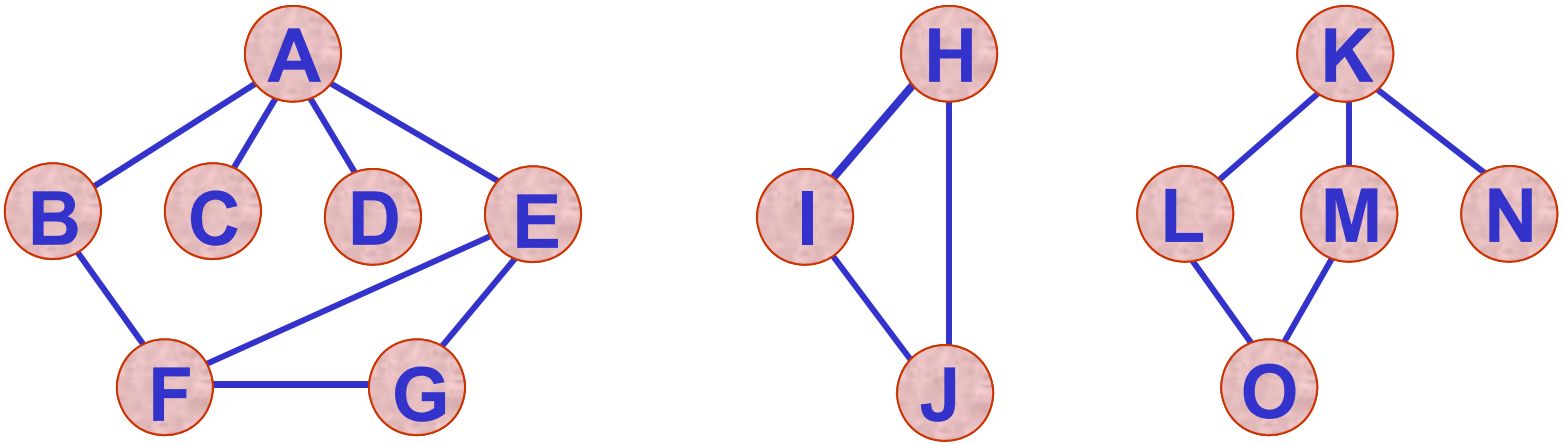
```
template <class NameType, class DistType> void Graph
<NameType, DistType> :: BFS(int v)
{ //从顶点v出发，按广度优先次序进行搜索
  //其中使用队列作为辅助存储结构
    int *Visited=new int [VerticesNumber( )]; //创建辅助数组
    for (int i=0; i<VerticesNumber( ); i++)
        Visited[i]=0; //辅助数组初始化
    cout<<v; //输出访问顶点信息
    Visited[v]=1; //该顶点访问标志修改为已访问过
    Queue <int> q; //实现分层访问的辅助存储结构——队列
    q.EnQueue(v); //顶点v进队
```

```
    while (!q.IsEmpty( ))
    {
        v=q.Dequeue( );  //从队列中退出顶点v
        int w=GetFirstNeighbor(v);
        //寻找顶点v的第一个邻接顶点w
        while (w!=-1)
        { //邻接顶点w存在
            if (!Visited[w])
            { //邻接顶点w未被访问过
                cout<<w<<‘ ’; //访问顶点w
                Visited[w]=1; //更改顶点w的访问标记
                q.Enqueue(w); //顶点w进队列
            }
            w=GetNextNeighbor(v, w);
            //寻找顶点v的下一个邻接顶点
        }
    }
    delete [ ] Visited;
}
```

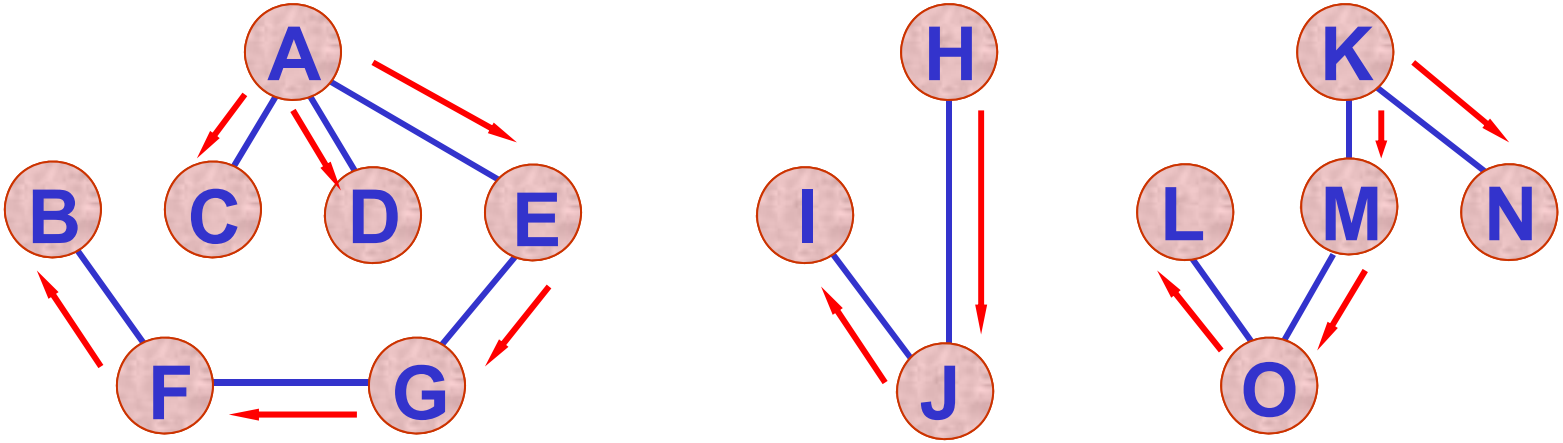
连通分量 (Connected component)

- 当无向图为非连通图时，从图中某一顶点出发，利用深度优先搜索算法或广度优先搜索算法不可能遍历到图中的所有顶点，只能访问到该顶点所在的最大连通子图（连通分量）的所有顶点。

- 若从无向图的每一个连通分量中的一个顶点出发进行遍历，可求得无向图的所有连通分量。
- 在算法中，需要对图的每一个顶点进行检测：若已被访问过，则该顶点一定是落在图中已求得的连通分量上；若还未被访问，则从该顶点出发遍历图，可求得图的另一个连通分量。
- 对于非连通的无向图，所有连通分量的生成树组成了非连通图的生成森林。



非连通无向图

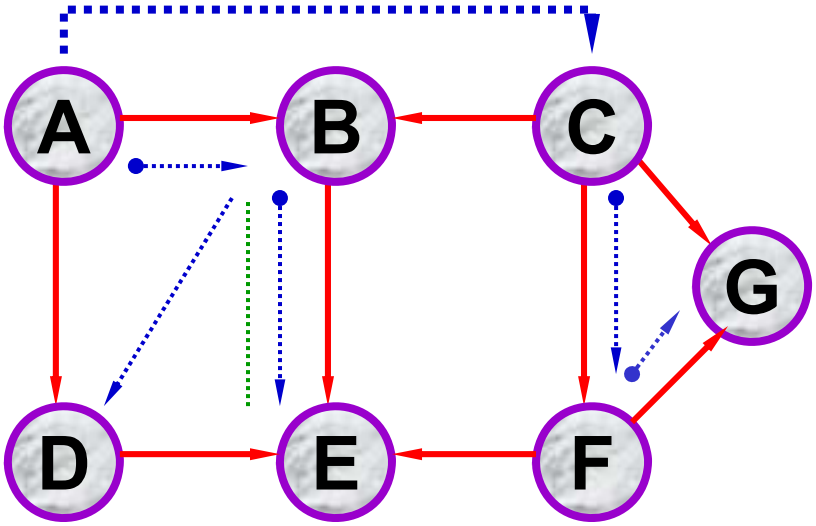


非连通图的连通分量

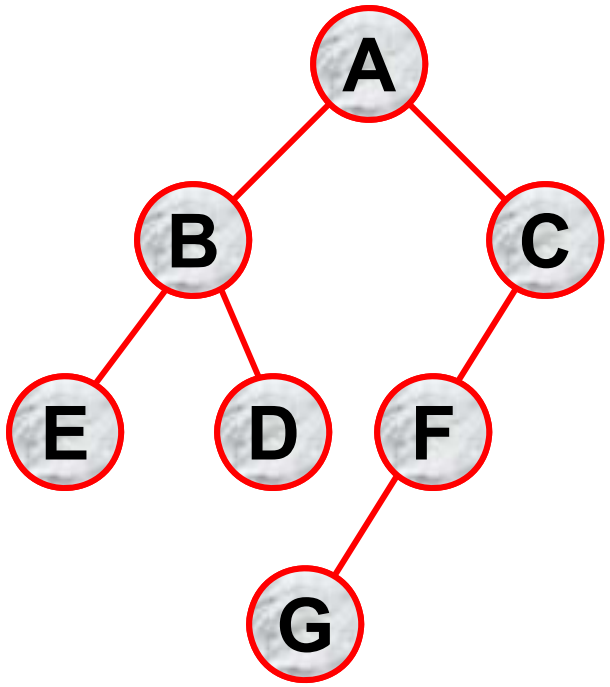
确定连通分量的算法

```
template <class Type> void Graph <Type> ::
Components( ) {
    int n=G.NumberOfVertices( );
    static int *Visited=new int[n];
    for (int i=0; i<n; i++)  Visited[i]=0;
    for (i=0; i<n; i++)
      if (!Visited[i]) {  //检测顶点是否访问过
          DFS(i, Visited);  //未访问，  出发访问
          OutputNewComponent( );  //连通分量
      }
    delete [ ] Visited;
}
```

【例】以深度优先搜索方法从顶点 A 出发遍历图，建立深度优先生成森林。



有向图



深度优先生成森林

```
template <class Type> void Graph <Type> ::
DFS_Forest(Graph <Type> &G, Tree <Type> &T) {
   TreeNode <Type> *rt, *subT;
   int n=G.NumberOfVertices( );
   static int *Visited=new int[n];  //访问数组
   for (int i=0; i<n; i++)  Visited[i]=0;
   for (i=0; i<n; i++)  //遍历所有顶点
     if (!Visited[i]) {  //顶点i未访问过
       if (T.IsEmpty( ))  //原为空森林，建根
          subT=rt=T.BuildRoot(G.GetValue(i));
          //顶点i的值成为根rt的值
```

```
        else subT=T.InsertRightSibling
                              (subT, G.GetValue(i));
        //顶点i的值成为subT右兄弟的值
        DFS_Tree(G, T, subT, i, Visited);
        //从顶点i出发深度优先遍历，建子树
    }
}
```

```
template <class Type> void void Graph <Type> ::
DFS_Tree(Graph <Type> &G, Tree <Type> &T,
                TreeNode <Type> *rt, int v,  int Visited [ ])
{
    TreeNode <Type> *p;
    Visited[v]=1;  //顶点v作访问过标志
    int w=G.GetFirstNeighbor(v);
    //取顶点v的第一个邻接顶点w
    int FirstChild=1;
    //第一个未访问子女应是v的左子女
    while (w!=-1) {  //邻接顶点w存在
    if (!visited[w]) {
    //w未访问过，将成为v的子女
```

```
            if (FirstChild) {
                p=T.InsertLeftChild(rt, G.GetValue(w));
                //p插入为rt的左子女
                FirstChild=0;  //建右兄弟    }
            else p=T.InsertRightSibling (p, G.GetValue(w));
            //p插入为p的右兄弟
            DFS_Tree(G, T, p, w, visited);
            //递归建立w的以p为根的子树
        } //邻接顶点w处理完
        w=G.GetNextNeighbor(v, w);
        //取v的下一个邻接顶点w
    } //回到while判邻接顶点w存在
}
```

8.4 最小生成树 (Minimum-Cost Spanning Tree)

- 使用不同的遍历图的方法，可以得到不同的生成树；从不同的顶点出发，也可能得到不同的生成树。
- 按照生成树的定义，n个顶点的连通网络的生成树有n个顶点、n-1条边。
- 构造最小生成树的准则
  - 必须使用且仅使用该网络中的n-1条边来联结网络中的n个顶点；
  - 不能使用产生回路的边；
  - 各边上的权值总和达到最小。

普里姆(Prim)算法

- 已知G=(V，E)是一个带权连通无向图，顶点V={1，2，...，n}，设U为V中构成最小（代价）生成树的顶点集合，初始时U={v_0}，v_0是指定的某一个顶点，$v_0 \in V$。
- T为构成最小（代价）生成树的边集合，初始时T为空。如果边(u，v)具有最小代价，且$u \in U$，$v \in V-U$，则最小生成树包含边(u，v)，即把v加到U中，把(u，v)加入T中。这个过程一直进行下去，直到U=V为止，这时T即为所求的最小生成树。

- 设T是带权连通无向图G=(V, E)的一棵正在建造的生成树，如果边(u, v)具有最小代价，且u∈U, v∈V-U，则G中包含T的最小生成树一定包含边(u, v)。

证明：用反证法证明上面提出的结论。

设G中的任何一棵包含T的最小生成树都不包含边(u, v)，且设T'就是这样的生成树。因为T'是树，所以它是连通的，从u到v必有一条路径(u, ..., v)，把(u, v)加入T'中，就构成一条回路，路径(u, ..., v)中必有(u', v')，满足u'∈U, v'∈V-U。

由假设，边(u, v)的代价小于边(u', v')的代价（因为边(u, v)具有最小代价），在回路中删去(u', v')，从而破坏了这个回路，剩下的边构成另一棵生成树T''，T''包含边(u, v)，且各边的代价总和小于T'各边的代价总和。因此，T''是一棵包含边(u, v)的最小（代价）生成树。这样，T'不是G的最小（代价）生成树，这与假设相矛盾。

(1) 设T是带权连通无向图G=(V, E)的最小生成树，初始时T为空，U为最小生成树的顶点集合，初始时U={v_0}，v_0是指定的某一个开始顶点。

(2) 若U=V，则算法终止；否则，从E中选一条代价最小的边(u, v)，使得u∈U, v∈V-U。将顶点v加到U中，将边(u, v)加到T中，转(2)。

在实现Prim算法中，采用邻接矩阵Cost表示给定的带权连通无向图，矩阵元素定义为：

$$\text{Cost}(i,j)=\begin{cases} w_{ij} & i\neq j,(i,j)\in E(G),w_{ij}\text{是}(i,j)\text{边上的权} \\ 0 & i=j \\ \infty & \text{否则} \end{cases}$$

- n为图中顶点个数，u为指定的开始顶点，同时使用两个数组CloseSet和LowCost，其中CloseSet用于存放顶点序号，而LowCost用于存放代价。
- 所有的顶点CloseSet[i] (1≤i≤n)都已在U中。
- 若LowCost[k]=0，则表明顶点k在U中；若0<LowCost[j]<∞，则j∈V-U，且(CloseSet[j], j)是与顶点j邻接的且两邻接顶点分别在U和V-U的所有边中代价最小的边，其最小代价就是LowCost[j]。
- 若LowCost[j]=∞，则表示CloseSet[j]与顶点j之间没有边，用9999表示 。

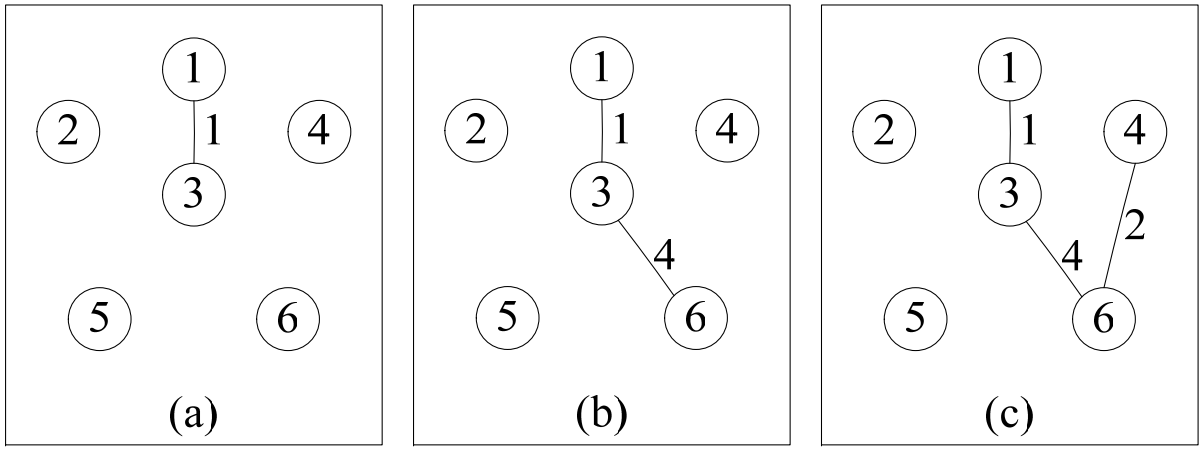
- 普里姆算法的基本思想

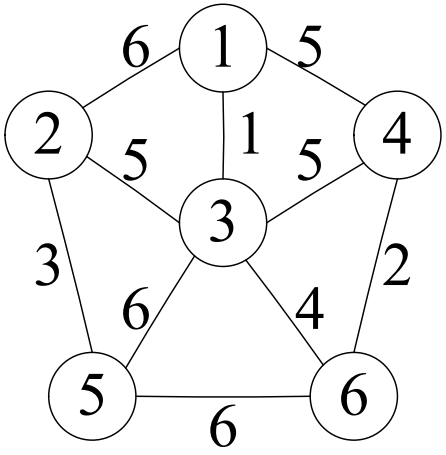
  从连通网络N={V，E} 中的某一顶点u_0出发，选择与它关联的具有最小权值的边(u_0, v)，将其顶点加入到生成树顶点集合U中。

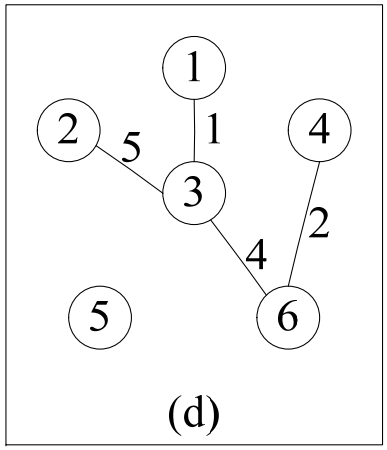
  以后每一步从一个顶点在U中，而另一个顶点不在U中的各条边中选择权值最小的(u, v)，把它的顶点加入到集合U中。如此继续下去，直到网络中的所有顶点都加入到生成树顶点集合U中为止。

- 采用邻接矩阵作为图的存储表示。

图8-19 用Prim算法构造图G_4的最小生成树的过程







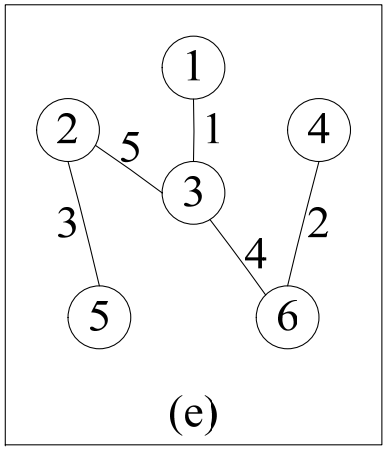
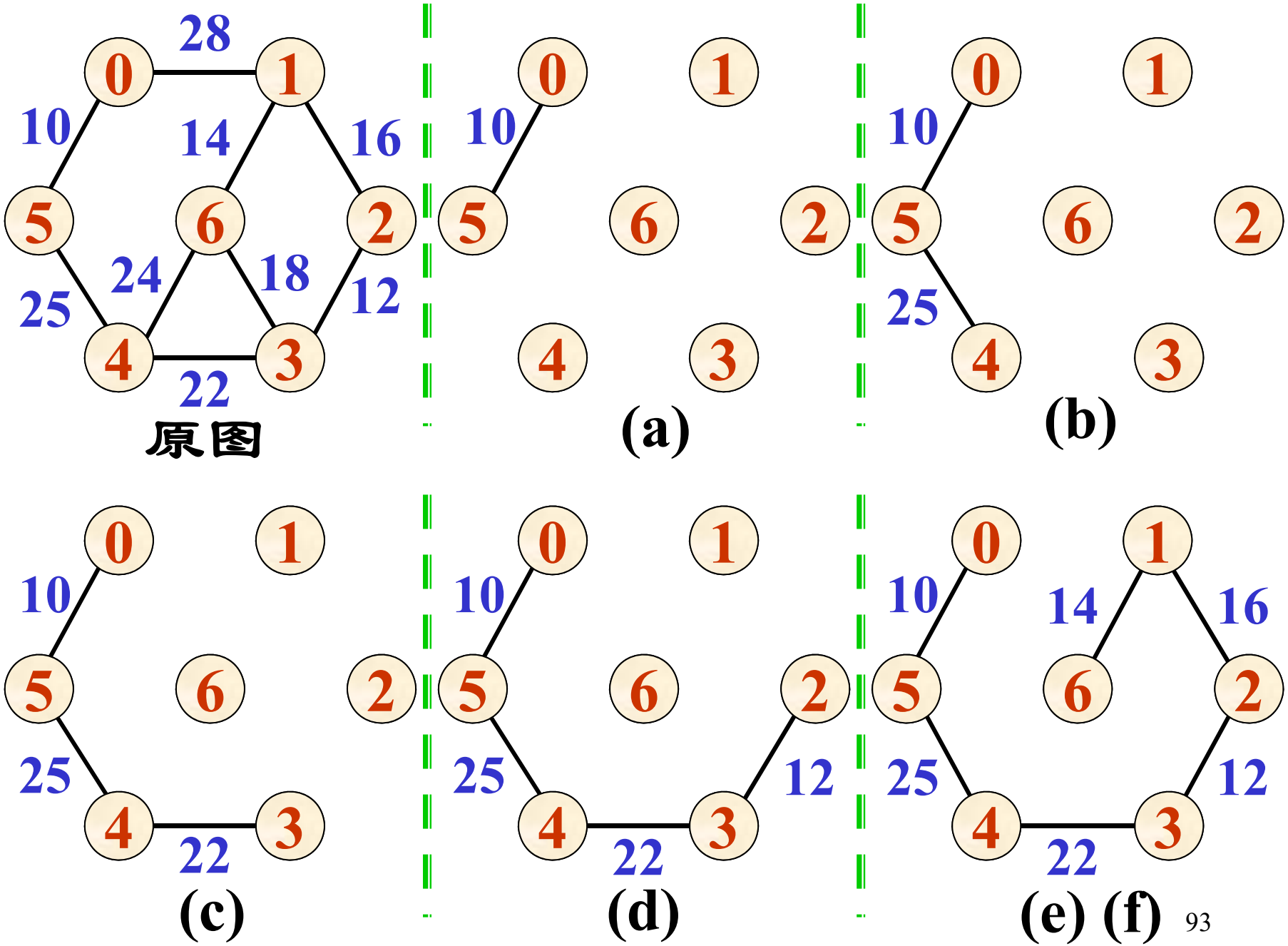
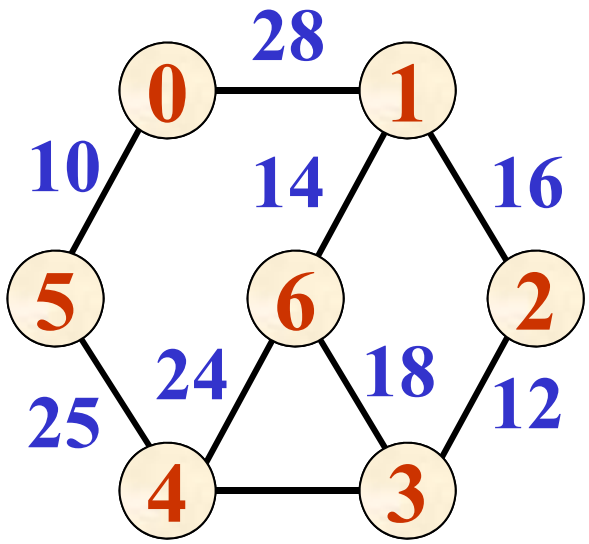


图8-18 带权连通无向图G_4


28
0
1
10
14
16
5
6
2
24
18
12
25
4
3
22
原图
(a)
(b)
(c)
(d)
(e) (f)

- 在构造过程中，还设置了两个辅助数组：
  - **lowcost[ ]**　存放生成树顶点集合内顶点到生成树外各顶点的各边上的当前最小权值；
  - **nearvex[ ]**　记录生成树顶点集合外各顶点距离集合内哪个顶点最近（即权值最小）。
- 例子



$$\begin{pmatrix} 0 & 28 & \infty & \infty & \infty & 10 & \infty \\ 28 & 0 & 16 & \infty & \infty & \infty & 14 \\ \infty & 16 & 0 & 12 & \infty & \infty & \infty \\ \infty & \infty & 12 & 0 & 22 & \infty & 18 \\ \infty & \infty & \infty & 22 & 0 & 25 & 24 \\ 10 & \infty & \infty & \infty & 25 & 0 & \infty \\ \infty & 14 & \infty & 18 & 24 & \infty & 0 \end{pmatrix}$$

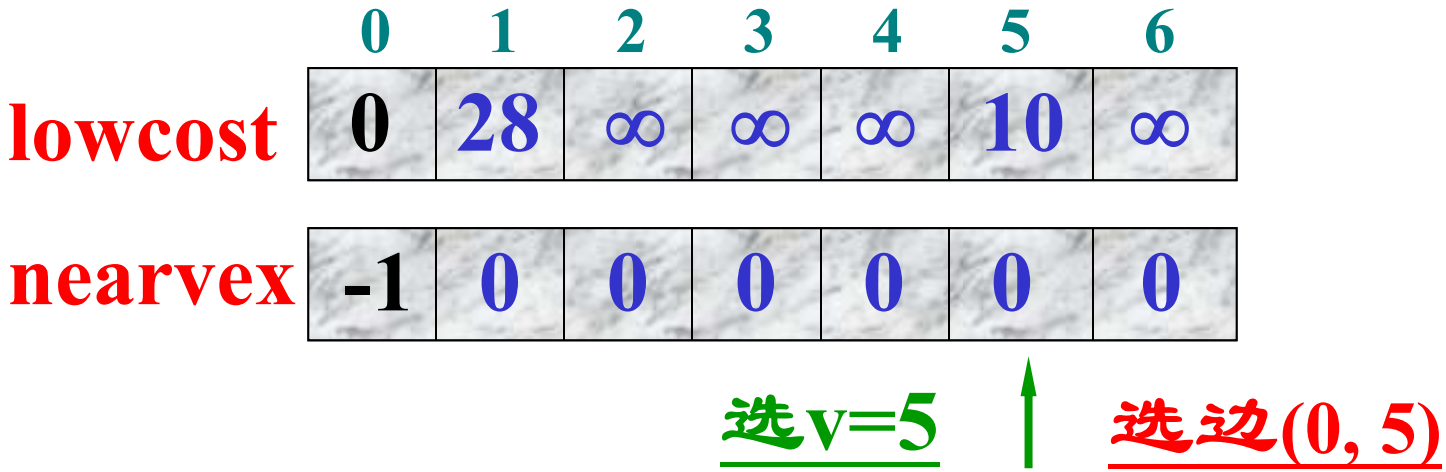
- 若选择从顶点0出发，即u_0=0，则两个辅助数组的初始状态为：

	0	1	2	3	4	5	6
lowcost	0	28	∞	∞	∞	10	∞
nearvex	-1	0	0	0	0	0	0

- 然后，反复做以下工作：
  - 在lowcost[ ]中选择nearvex[i]≠-1 && lowcost[i]最小的边，用v标记它。则选中的权值最小的边为(nearvex[v], v)，相应的权值为 lowcost[v]。

- 将**nearvex[v]**改为**-1**，表示它已加入生成树顶点集合。
- 将边**(nearvex[v]，v，lowcost[v])**加入生成树的边集合。
- 取**lowcost[i]=min{lowcost[i], Edge[v][i]}**，即用生成树顶点集合外各顶点**i**到刚加入该集合的新顶点**v**的距离**Edge[v][i]**与原来它们到生成树顶点集合中顶点的最短距离**lowcost[i]**做比较，取距离近的作为这些集合外顶点到生成树顶点集合内顶点的最短距离。

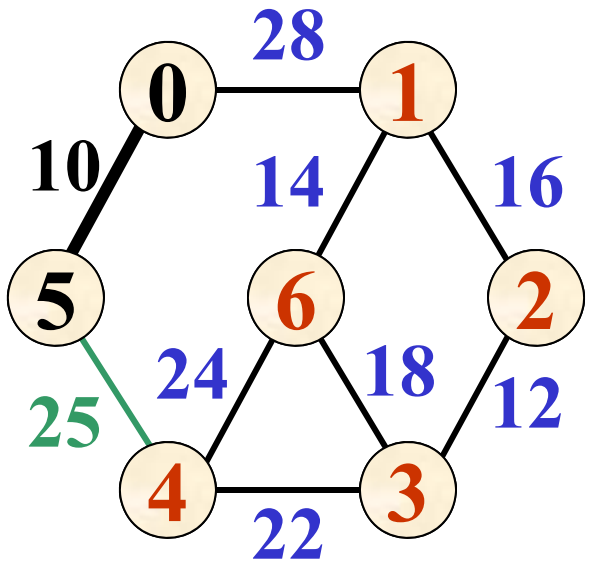
- 如果生成树顶点集合外顶点i到刚加入该集合的新顶点v的距离比原来它到生成树顶点集合中顶点的最短距离还要近，则修改nearvex[i]：nearvex[i]=v，表示生成树外顶点i到生成树内顶点v当前距离最近。



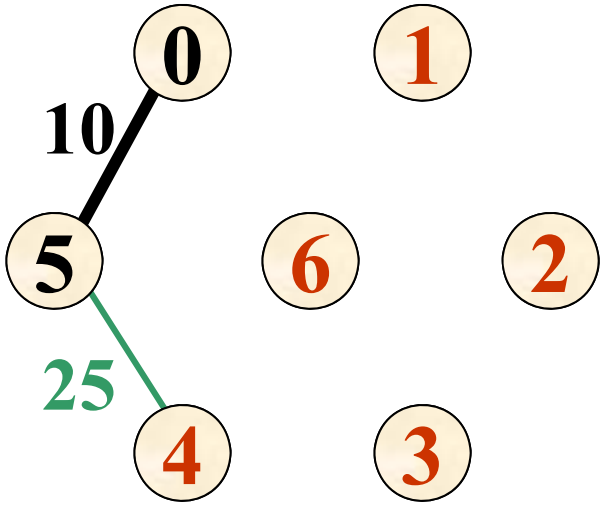
顶点v=5加入生成树顶点集合：

	0	1	2	3	4	5	6
lowcost	0	28	∞	∞	25	10	∞
nearvex	-1	0	0	0	5	-1	0

选v=4　　选边(5, 4)



原图

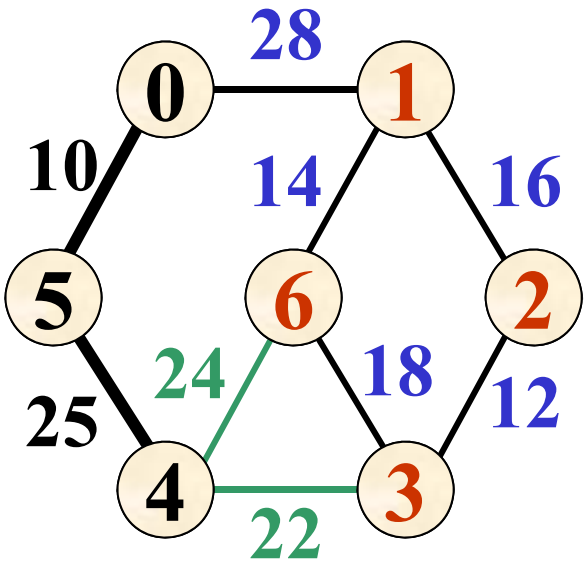


边(0, 5, 10)加入生成树

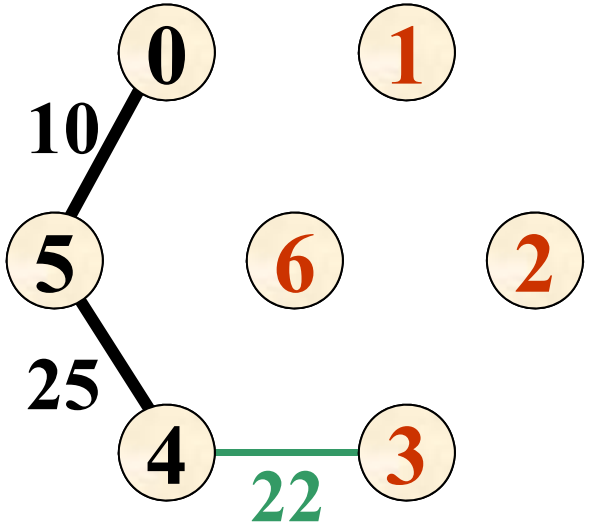
顶点v=4加入生成树顶点集合：

	0	1	2	3	4	5	6
lowcost	0	28	∞	22	25	10	24
nearvex	-1	0	0	4	-1	-1	4

选v=3　　选边(4, 3)



原图

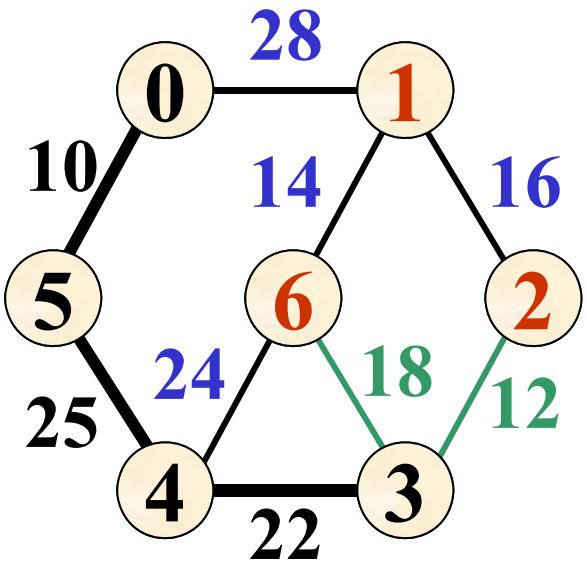


边(5, 4, 25)加入生成树

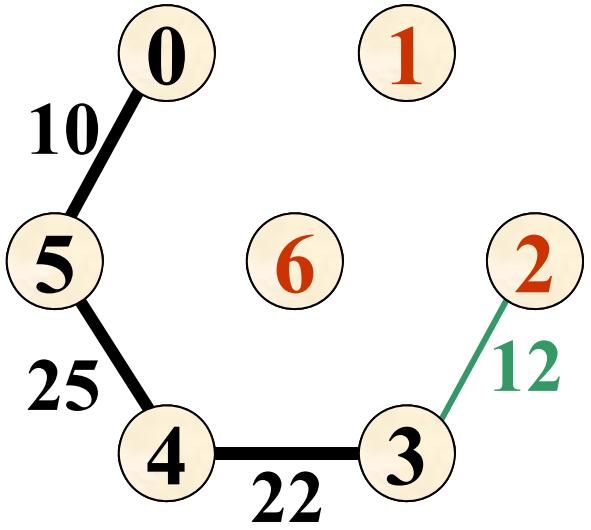
顶点v=3加入生成树顶点集合：

	0	1	2	3	4	5	6
lowcost	0	28	12	22	25	10	18
nearvex	-1	0	3	-1	-1	-1	3

选v=2　　选边(3, 2)



原图

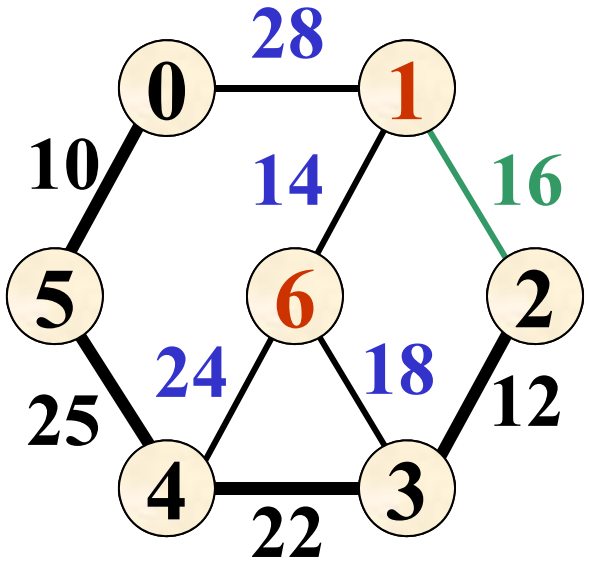


边(4, 3, 22)加入生成树

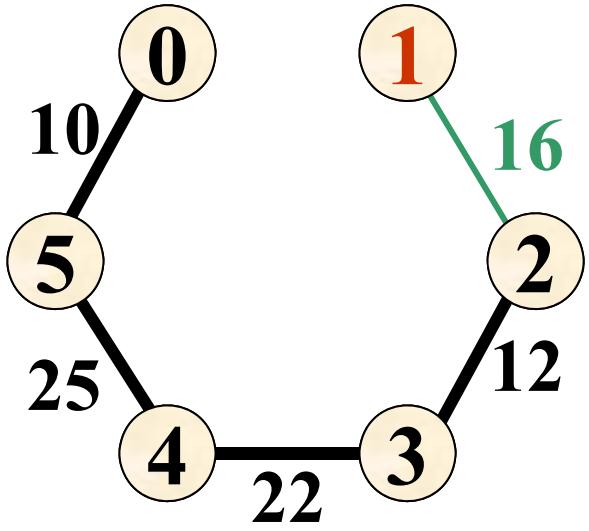
顶点v=2加入生成树顶点集合：

	0	1	2	3	4	5	6
lowcost	0	16	12	22	25	10	18
nearvex	-1	2	-1	-1	-1	-1	3

选v=1　　选边(2, 1)



原图

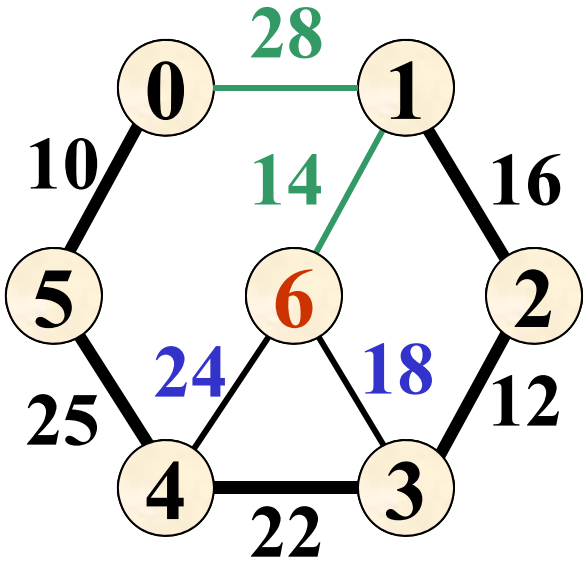


边(3, 2, 12)加入生成树

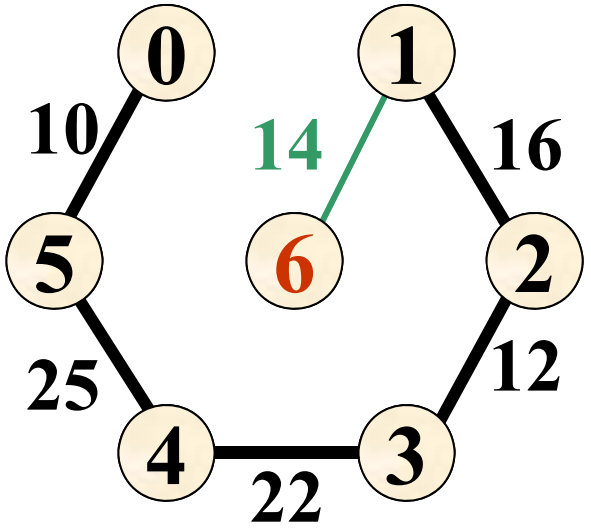
顶点v=1加入生成树顶点集合：

	0	1	2	3	4	5	6
lowcost	0	16	12	22	25	10	14
nearvex	-1	-1	-1	-1	-1	-1	1

选v=6　　选边(1, 6)



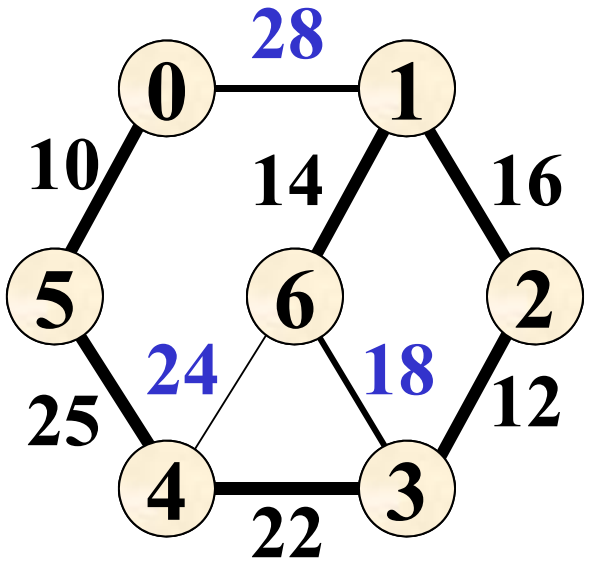
原图



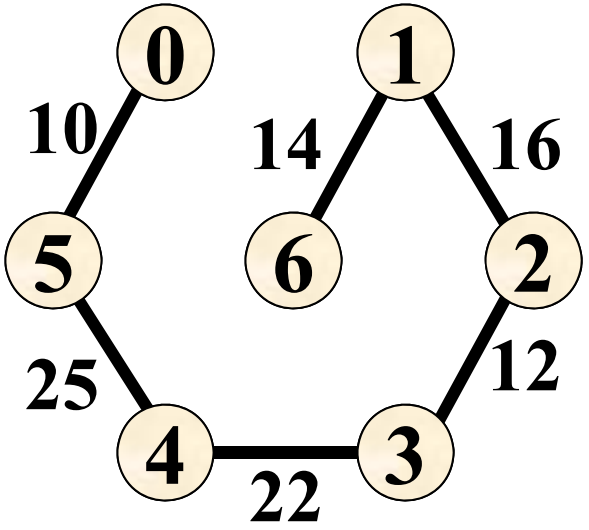
边(2, 1, 16)加入生成树

顶点v=6加入生成树顶点集合：

	0	1	2	3	4	5	6
lowcost	0	16	12	22	25	10	14
nearvex	-1	-1	-1	-1	-1	-1	-1



原图



边(1, 6, 14)加入生成树

最后生成树中边集合里存入得各条边为：

(0, 5, 10), (5, 4, 25), (4, 3, 22),
(3, 2, 12), (2, 1, 16), (1, 6, 14)

利用普里姆算法建立最小生成树

```
void Prim(Graph <string> &G, MinSpanTree &T) {
    int i, j, n=G.NumberOfVertices( );  //顶点数
    float *lowcost=new float[n];
    int *nearvex=new int[n];
    for (i=1; i<n; i++) {
        lowcost[i]=G.GetWeight(0, i);
        nearvex[i]=0;
    } //顶点0到各边代价及最短带权路径
```

```
nearvex[0]=-1;  //加到生成树顶点集合
MSTEdgeNode e;  //最小生成树结点单元
for (i=1; i<n; i++) {
//循环n-1次，加入n-1条边
   float min=MaxValue;  int v=0;
   for (j=0; j<n; j++)
      if (nearvex[j]!=-1 && lowcost[j]<min)
         { v=j;  min=lowcost[j]; }
   //求生成树外顶点到生成树内顶点
   //具有最小权值的边
```

```
        if (v) {  //v=0表示再也找不到要求顶点
            e.tail=nearvex[v];   e.head=v;
            e.cost=lowcost[v];
            T.Insert(e);  //选出的边加入生成树
            nearvex[v]=-1;  //该边加入生成树标记
            for (j=1; j<n; j++)
                if (nearvex[j]!=-1 &&
                        G.GetWeight(v, j)<lowcost[j]) {
                    lowcost[j]=G.GetWeight(v, j);
                    nearvex[j]=v;    }
        }
    }  //循环n-1次，  加入n-1条边
}
```

- 分析以上算法，设连通网络有n个顶点，则该算法的时间复杂度为$O(n^2)$，它适用于边稠密的网络。

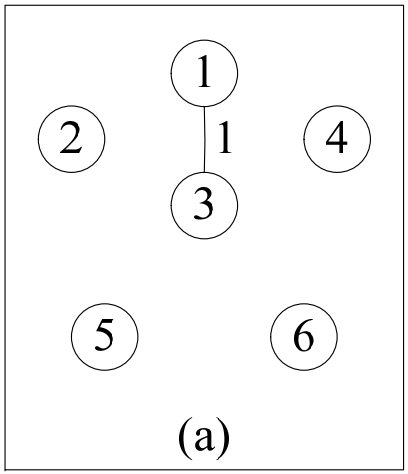
克鲁斯卡尔 (Kruskal) 算法

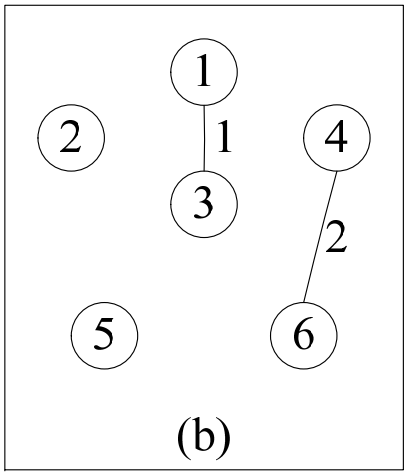
- 按权值的递增次序选择合适边来构造最小生成树。
- 图G是一个具有n个顶点的带权连通无向图，设T是G的最小（代价）生成树，初始时T=(V, Φ)，即T由n个连通分量组成，每个连通分量只有一个顶点，没有边。
  - 首先，把E中的边按代价（即权）的递增次序进行排序，然后按排好序的顺序选取边，即反复执行下面的选择步骤。这样的过程一直进行到T包含有(n-1)条边为止，算法结束，这时的T便是所求的最小（代价）生成树。
  - 选择步骤：若当前被选择的边的两个顶点在不同的连通分量中，则把这条边加到T中，选取这样的边可以保证不会构成回路。然后，再对下一条边进行选择，若当前被选择的边的两个顶点在同一连通分量中，则不能选取这条边，如果选取它，则必会构成回路。接着，对下一条边进行选择。

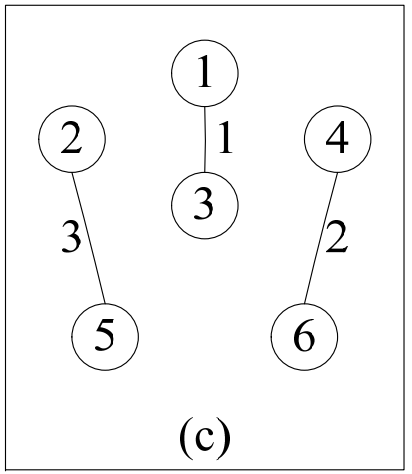
- 克鲁斯卡尔算法的基本思想

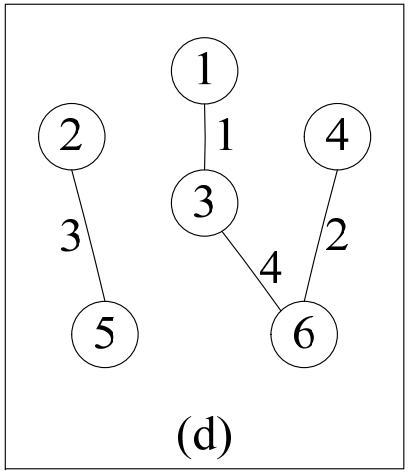
设一个有**n**个顶点的连通网络**N={V, E}**，最初先构造一个只有**n**个顶点，没有边的非连通图**T={V, ∅}**，图中每个顶点自成一个连通分量。当在**E**中选到一条具有最小权值的边时，若该边的两个顶点落在不同的连通分量上，则将此边加入到**T**中；否则将此边舍去，重新选择一条权值最小的边。如此重复下去，直到所有顶点在同一个连通分量上为止。

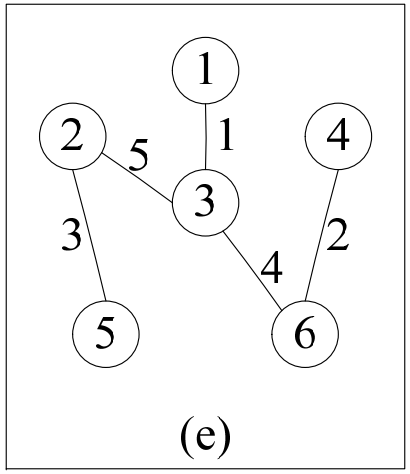
图8-20 用Kruskal算法构造图G_4的最小生成树的过程











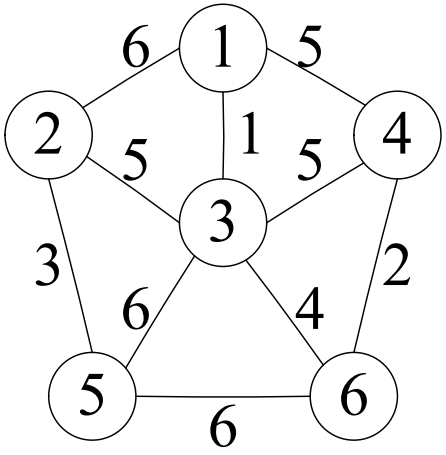
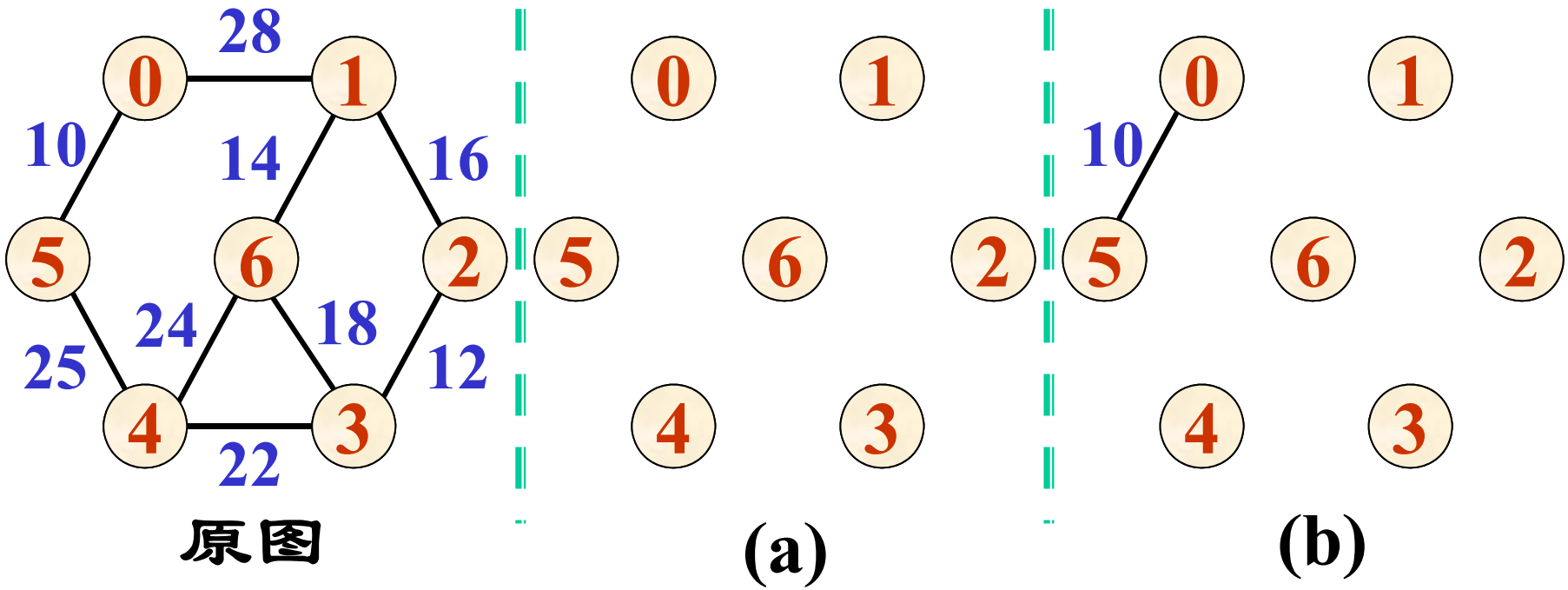
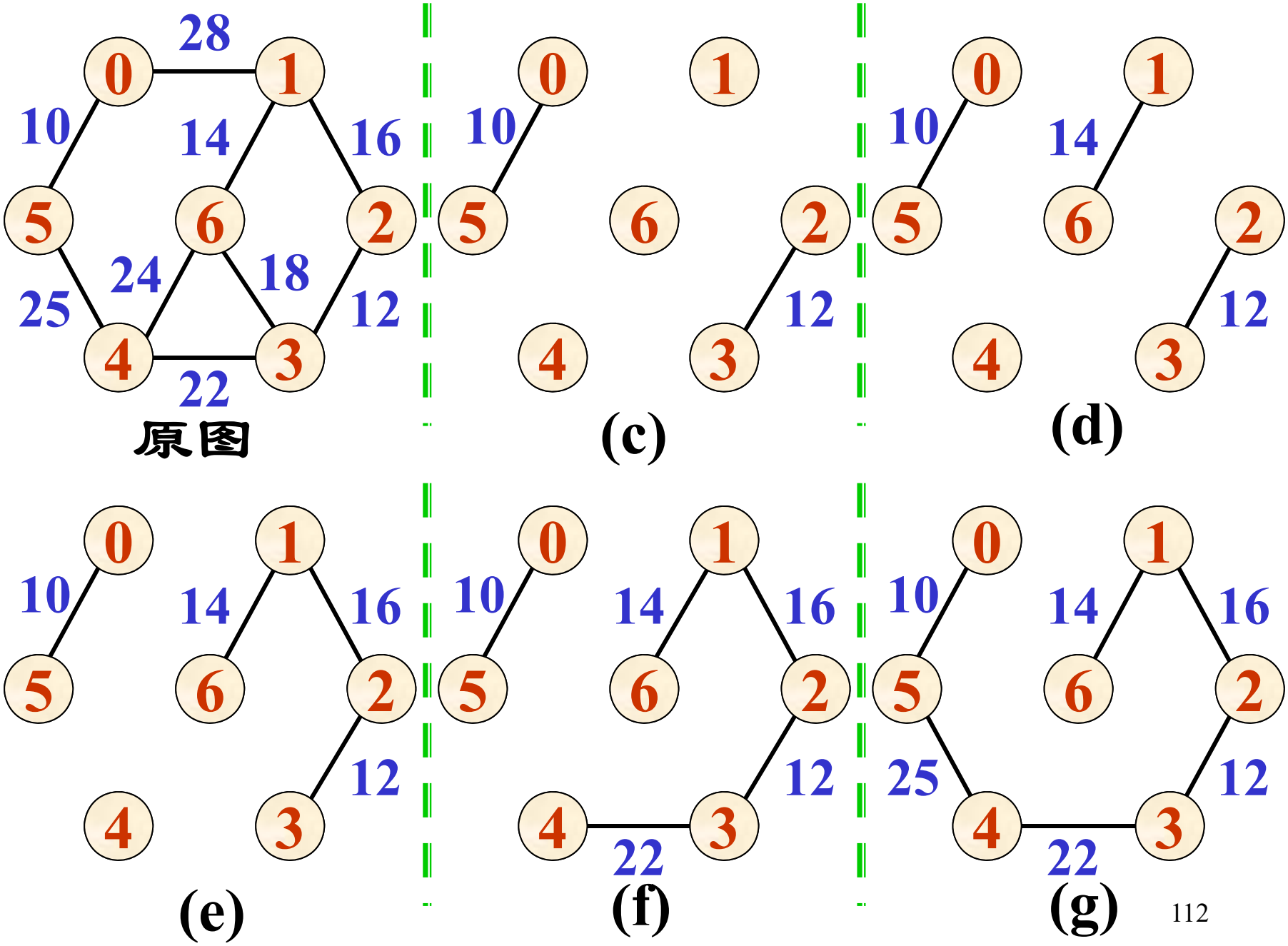


图8-18 带权连通无向图G_4

应用克鲁斯卡尔算法构造最小生成树的过程




0
1
2
3
4
5
6
28
10
14
16
24
18
12
25
22
原图
(c)
(d)
(e)
(f)
(g)

- 算法的框架　　利 用 最 小 堆 (MinHeap) 和 并 查 集 (DisjointSets)来实现克鲁斯卡尔算法。
- 首先，利用最小堆来存放E中所有的边，堆中每个结点的格式为

- 在构造最小生成树过程中，利用并查集的运算检查依附一条边的两顶点tail、head是否在同一连通分量（即并查集的同一个子集合）上，是则舍去这条边；否则将此边加入T，同时将这两个顶点放在同一连通分量。
- 随着各边逐步加入到最小生成树的边集合中，　各连通分量也在逐步合并，直到形成一个连通分量为止。

最小生成树类定义

```
class MinSpanTree;
class MSTEdgeNode {  //生成树边结点类
friend class MinSpanTree;
private:
    int tail, head;  //生成树各边的两顶点
    float cost;  //生成树各边的权值
public:
    MSTEdgeNode( )  //构造函数
        : tail (-1), head(-1), cost(0) { }
};
```

```
class MinSpanTree {
  protected:
    MSTEdgeNode *edgevalue;
    int MaxSize, n;
  public:
    MinSpanTree(int sz=NumVertices-1)
      : MaxSize(sz), n(0) {
        edgevalue=new MSTEdgeNode[MaxSize];
    }
    int Insert(MSTEdgeNode &item);
};
```

利用克鲁斯卡尔算法建立最小生成树

```
void Kruskal(Graph <string> &G,
                          MinSpanTree &T) {
   MSTEdgeNode ed;  //边结点辅助单元
   int n=G.NumberOfVertices( );  //顶点数
   int e=G.NumberOfEdges( );  //边数
   MinHeap <MSTEdgeNode> H(e);  //最小堆
   UFSets F(n);  //并查集
   for (int u=0; u<n; u++)
      for (int v=u+1; v<n; v++)
         if (G.GetWeight(u, v)!=MaxValue) {
            ed.tail=u;  ed.head=v;  //插入堆
```

```
            ed.cost=GetWeight(u, v);  H.Insert(ed);
        }
    int count=1;  //最小生成树加入边数计数
    while (count<n) {
        H.RemoveMin(ed);  //从堆中退出一条边
        u=F.Find(ed.tail);  v=F.Find(ed.head);
        if (u!=v) {  //两端不在同一连通分量
            F.Union(u, v);  //合并
            T.Insert(ed);  //该边存入最小生成树
            count++;
        }
    }
}
```

出堆顺序

(0,5,10)选中　(2,3,12)选中

(1,6,14)选中　(1,2,16)选中　(3,6,18)舍弃

(3,4,22)选中　(4,6,24)舍弃　(4,5,25) 选中

	0	1	2	3	4	5	6
F	-1	-1	-1	-1	-1	-1	-1
	-2	-1	-1	-1	-1	0	-1
	-2	-1	-2	2	-1	0	-1
	-2	-2	-2	2	-1	0	1
	-2	-4	1	2	-1	0	1
	-2	-5	1	2	1	0	1
	1	-7	1	2	1	0	1

并查集

	0	1	2	3	4	5	6	
	0	28	∝	∝	∝	10	∝	0
	28	0	16	∝	∝	∝	14	1
	∝	16	0	12	∝	∝	∝	2
	∝	∝	12	0	22	∝	18	3
	∝	∝	∝	22	0	25	24	4
	10	∝	∝	∝	25	0	∝	5
	∝	14	∝	18	24	∝	0	6

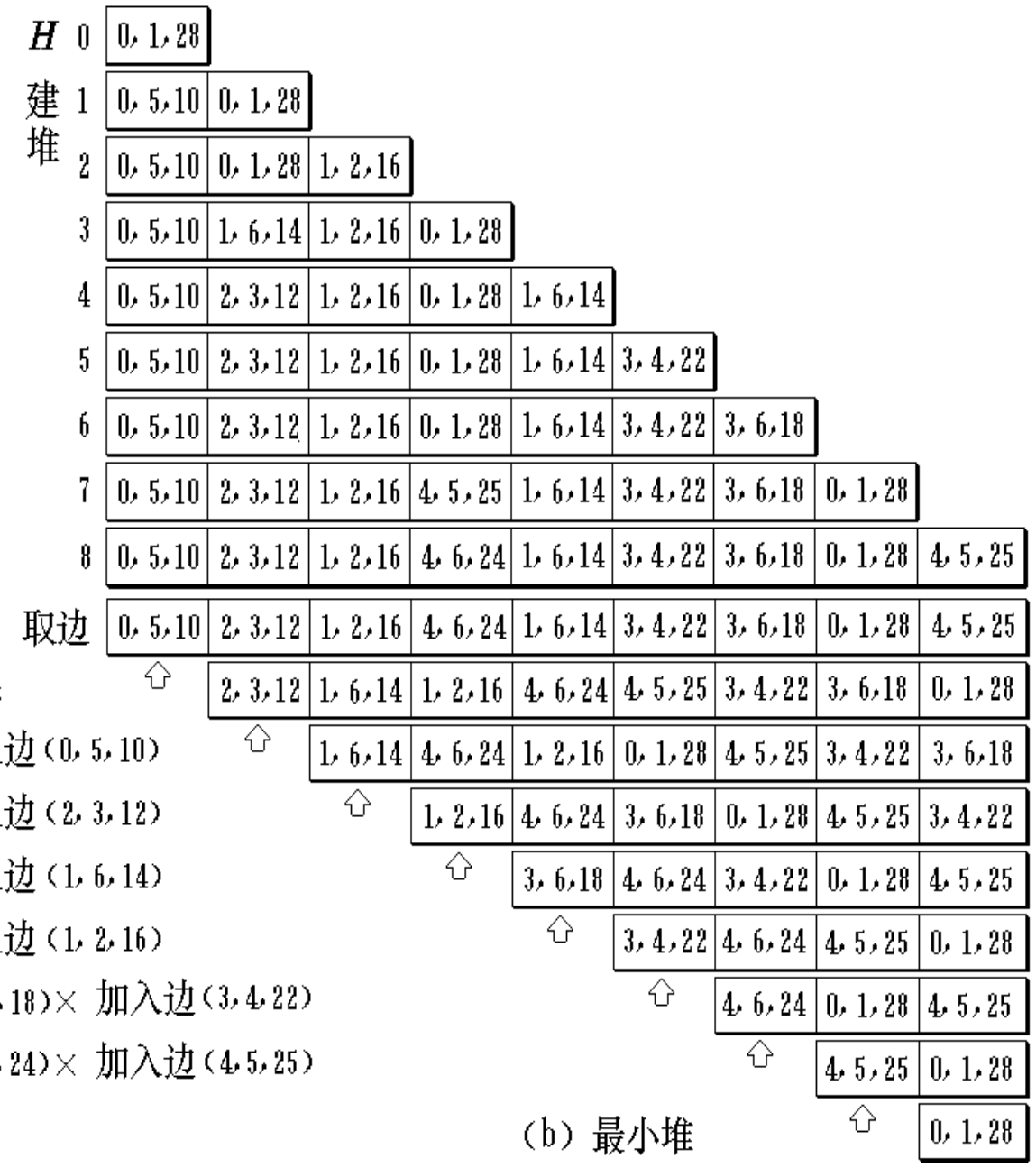
邻接矩阵表示

	0	1	2	3	4	5	6	
	0	28	∞	∞	∞	10	∞	0
	28	0	16	∞	∞	∞	14	1
	∞	16	0	12	∞	∞	∞	2
	∞	∞	12	0	22	∞	18	3
	∞	∞	∞	22	0	25	24	4
	10	∞	∞	∞	25	0	∞	5
	∞	14	∞	18	24	∞	0	6

（a）邻接矩阵表示

	0	1	2	3	4	5	6	
F	-1	-1	-1	-1	-1	-1	-1	初态
	-2	-1	-1	-1	-1	0	-1	加入边(0,5,10)
	-2	-1	-2	2	-1	0	-1	加入边(2,3,12)
	-2	-2	-2	2	-1	0	1	加入边(1,6,14)
	-2	-4	1	2	-1	0	1	加入边(1,2,16)
	-2	-5	1	2	1	0	1	(3,6,18)× 加入边(3,4,22)
	1	-7	1	2	1	0	1	(4,6,24)× 加入边(4,5,25)

（c）并查集



（b）最小堆

- 克鲁斯卡尔算法不仅适合于边稠密的情形，也适合于边稀疏的情形。
- 注意：当各边有相同权值时，由于选择的随意性，产生的生成树可能不惟一。

8.5 最短路径 (Shortest Path)

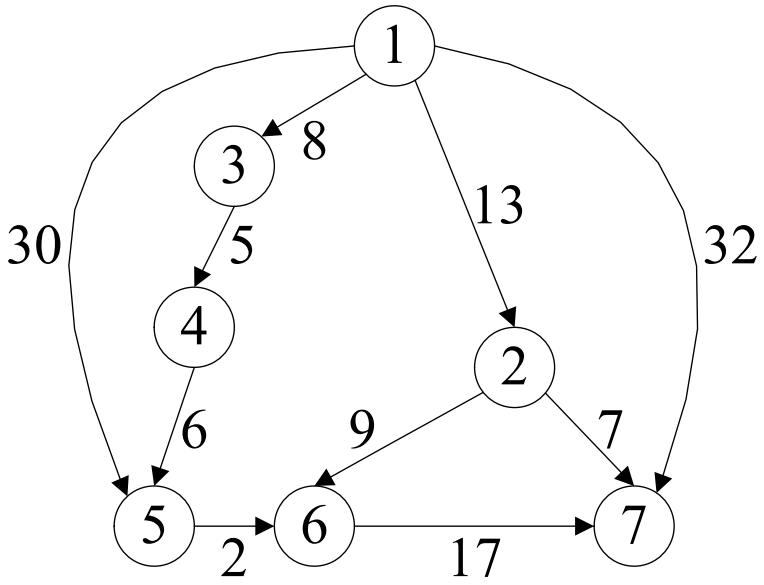
- 最短路径问题：如果从图中某一顶点（称为源点）到达另一顶点（称为终点）的路径可能不止一条，如何找到一条路径使得沿此路径上各边上的权值总和达到最小。
- 问题解法
  - 边上权值非负情形的单源最短路径问题
    — **Dijkstra算法**
  - 边上权值为任意值的单源最短路径问题
    — **Bellman和Ford算法**
  - 所有顶点之间的最短路径
    — **Floyd算法**

边上权值非负情形的单源最短路径问题

- 问题的提法：给定一个带权有向图D与源点v，求从v到D中其它顶点的最短路径，限定各边上的权值大于或等于0。
- 为求得这些最短路径，Dijkstra提出按路径长度的递增次序，逐步产生最短路径的算法。首先求出长度最短的一条最短路径，再参照它求出长度次短的一条最短路径，依此类推，直到从顶点v到其它各顶点的最短路径全部求出为止。

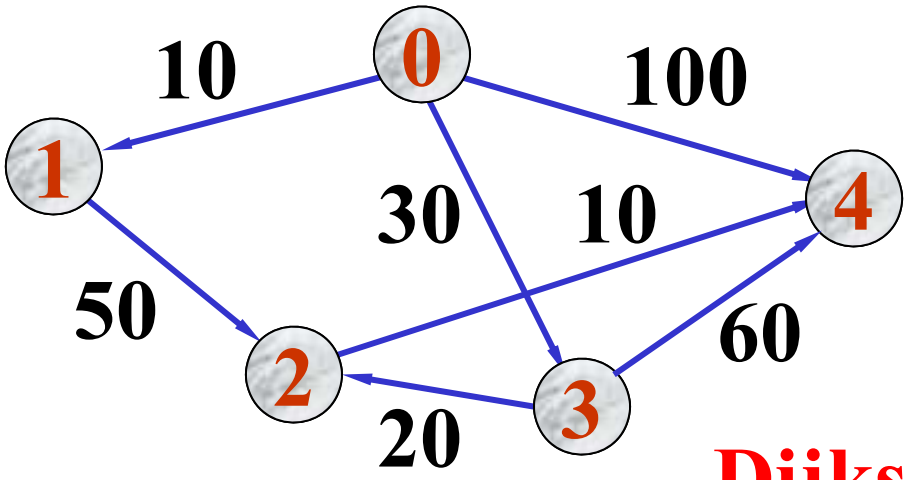
- 设**G**是具有**n**个顶点的带权的有向图，用代价邻接矩阵**Cost**表示图**G**。

$$\text{Cost}(i,j)=\begin{cases} w_{ij} & i\neq j, <i,j>\in E(G), w_{ij}\text{是}<i,j>\text{边上的权} \\ 0 & i=j \\ \infty & i\neq j, <i,j>\text{不在}E(G)\text{中} \end{cases}$$



最短路径	长度
<1, 2>	13
<1, 3>	8
<1, 3, 4>	13
<1, 3, 4, 5>	19
<1, 3, 4, 5, 6>	21
<1, 2, 7>	20

图8-21 G_5及从顶点1到其余顶点的最短路径



举例说明

Dijkstra逐步求解的过程

源点	终点	最短路径			路径长度
v_0	v_1	(v_0,v_1)			10
	v_2	——	(v_0,v_1,v_2)	(v_0,v_3,v_2)	∞,60,50
	v_3	(v_0,v_3)			30
	v_4	(v_0,v_4)	(v_0,v_3,v_4)	(v_0,v_3,v_2,v_4)	100,90,60

- 基本思想：把图中顶点集合V分成两组，令S表示已求出最短路径的顶点集合为一组，其余的尚未确定最短路径的顶点集合为第二组。
  - 按最短路径长度的递增次序逐个地把第二组的顶点加入S中，直至从v出发可以到达的所有顶点都在S中。
  - 在这过程中，总保持从v到S中各顶点的最短路径长度都不从v到第二组的任何顶点的最短路径长度。
  - 另外，每个顶点对应一个距离，S中的顶点的距离就是从v到此顶点的最短路径长度，第二组顶点的距离是从v到此顶点的只包括S中的顶点为中间顶点的当前最短路径长度。

- 具体做法
  - 初始时，S={v}，即S中只包含源点v，v的距离为零。
  - 第二组包括其它所有顶点，而这一组的顶点的距离为：若图中有边<v, w>，则w的距离就是这条边上的权；否则w的距离为一个很大的数（即算法中用9999表示）。
  - 然后，每次从第二组的顶点中选取一个距离最小的顶点k，把k加入S中，每次加入一个顶点到S中后，就要对第二组的各个顶点的距离进行一次修改。
  - 若加进顶点k做中间顶点，从v到顶点j (j∈V-S)的距离比原来不经过顶点k的距离短，则要修改顶点j的距离值。修改后，再选距离最小的顶点加入S中，并对V-S中的顶点的距离进行修改。
  - 这样的过程连续进行下去，直到G中所有顶点都包含在S中，或再也没有可加入S的顶点存在。

- 求解算法

(1) 把顶点v放入集S中。

(2) 按如下步骤逐个求得从v到其它顶点的最短路径，直至把 所有顶点的最短路径都求出为止：

(a) 选取不在S中，且具有最小距离的顶点k。

(b) 把顶点k加入集合S中。

(c) 修改不在S中的顶点的距离。

- 具体实现
  - 使用一个存放各顶点的当前距离的数组**Dist**，一旦从源点**v**到顶点**k**的最短路径已求出，则**Dist[k]**就是从源点到顶点**k**的最短路径长度。
  - 使用数组**Pre**，数组元素**Pre[j]**存放从源点**v**到顶点**j**的最短路径中**j**前面的顶点。有了**Pre**数组，就能很容易地求得从源点**v**到其它各个顶点的路径。下面给出的算法只求出从源点**v**到图中各个顶点的最短路径长度，同时求得**Pre**数组各元素的值，但没有求得从**v**到各个顶点的最短路径。
  - 用**9999**表示邻接矩阵的 。

程序8-6 最短路径的Dijkstra算法

```
const int MAXINT=9999;
const int VerticesMaxNumber=10;
//图中最大顶点个数
class Graph {  //图的类定义
   private:
      float Cost[VerticesMaxNumber][VerticesMaxNumber];
      //图的邻接矩阵
      float Dist[VerticesMaxNumber];
      //存放从顶点0到其它各顶点的最短路径长度
      int Pre[VerticesMaxNumber];
      //存放在最短路径上该顶点的前一顶点的顶点号
      int S[VerticesMaxNumber];
      //已求得的在最短路径上的顶点的顶点号
   public:
      void ShortestPath(int n, int v);
      ……
};
```

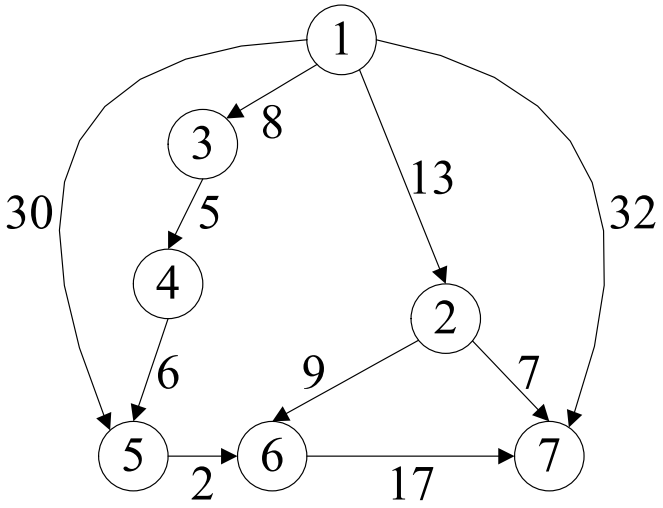
```
void Graph :: ShortestPath(int n, int v)
{  //G是一个具有n个顶点的带权有向图
   //各边上的权值由Cost[i][j]给出
   //建立一个数组Dist[j], 0≤j<n
   //是当前求到的从顶点v到顶点j的最短路径长度
   //建立数组path[j] , 0≤j<n，存放求到的最短路径
    int i, j, k;
    float Min;
    for (i=1; i<=n; i++)
    { //Dist、Pre和S数组初始化
        Dist[i]=Cost[v][i]; //邻接矩阵第v行复制到Dist中
        S[i]=0;
        if (Dist[i]<MAXINT)
            Pre[i]=v;
        else Pre[i]=0;
    }
    S[v]=1; //顶点v加入S集合中
    Pre[v]=0;
```

```
for (i=1; i<=n; i++)
{  //从顶点v确定n-1条路径
      Min=MAXINT;
      k=0;
      for (j=1; j<=n; j++)
      //选择当前不在集合S中具有最短路径的顶点u
          if (S[j]==0)
              if (Dist[j]!=0 && Dist[j]<Min)
              {
                    Min=Dist[j];
                    k=j;
              }
       if (k==0)  continue;
       S[k]=1;
       //将顶点u加入S集合，表示它已在最短路径上
```

```
            for (j=1; j<=n; j++)
                if (S[j]==0 && Cost[k][j]<MAXINT)  //修改
                    if (Dist[k]+Cost[k][j]<Dist[j])
                    {
                        Dist[j]=Dist[k]+Cost[k][j];
                        Pre[j]=k;
                    }
        }
}
```

图8-22 图G_5的代价邻接矩阵

$$\begin{bmatrix} 0 & 13 & 8 & * & 30 & * & 32 \\ * & 0 & * & * & * & 9 & 7 \\ * & * & 0 & 5 & * & * & * \\ * & * & * & 0 & 6 & * & * \\ * & * & * & * & 0 & 2 & * \\ * & * & * & * & * & 0 & 17 \\ * & * & * & * & * & * & 0 \end{bmatrix}$$



迭代次数	栈S	选取顶点	Dist[1]至Dist[7]							Pre[1]至Pre[7]						
0	1	3	0	13	8	*	30	*	32	0	1	1	0	1	0	1
1	1, 3	2	0	13	8	13	30	*	32	0	1	1	3	1	0	1
2	1, 3, 2	4	0	13	8	13	30	22	20	0	1	1	3	1	2	2
3	1, 3, 2, 4	5	0	13	8	13	19	22	20	0	1	1	3	4	2	2
4	1, 3, 2, 4, 5	7	0	13	8	13	19	21	20	0	1	1	3	4	5	2
5	1, 3, 2, 4, 5, 7	6	0	13	8	13	19	21	20	0	1	1	3	4	5	2
			0	13	8	13	19	21	20	0	1	1	3	4	5	2
6	1, 3, 2, 4, 5, 7, 6															

图8-23 G_5中以顶点1为源点时，算法中每次迭代后的数据

- 引入辅助数组**dist**，它的每一个分量**dist[i]**表示当前找到的从源点v_0到终点v_i的最短路径的长度。初始状态：
  - 若从源点v_0到顶点v_i有边，则**dist[i]**为该边上的权值；
  - 若从源点v_0到顶点v_i无边，则**dist[i]**为∞。
- 假设**S**是已求得最短路径的终点的集合，则可证明：下一条最短路径必然是从v_0出发，中间只经过**S**中的顶点便可到达的那些顶点v_x ($v_x \in V-S$)的路径中的一条。
- 每次求得一条最短路径后，其终点v_k加入集合**S**，然后对所有$v_i \in V-S$，修改其**dist[i]**值。

Dijkstra算法可描述如下：

① 初始化：S←{v_0};

dist[j]←Edge[0][j]，j=1, 2, …, n-1;

//n为图中顶点个数

② 求出最短路径的长度：

dist[k]←min{dist[i]}，i∈V- S;

S←SU{k};

③ 修改：

dist[i]←min{dist[i], dist[k]+Edge[k][i]},

对于每一个i∈V-S;

④ 判断：若S=V，则算法结束，否则转②。

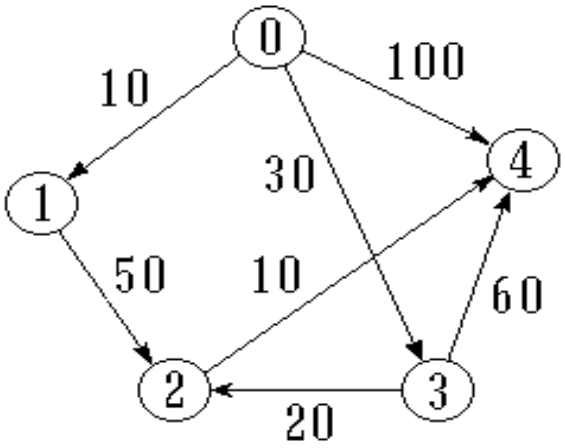
计算从单个顶点到其它各顶点最短路径

```
void ShortestPath(Graph <float> &G, int v,
                  float dist[ ], int path[ ]) {
//Graph是一个带权有向图
//各边上的权值由Edge[i][j]
//给出本算法建立一个数组dist[j], 0≤j<n
//是当前求到的从顶点v到顶点j的最短路径长度
//同时用数组path[j], 0≤j<n，存放求到的最短路径
   int n=G.NumberOfVertices( );
   int *S=new int [n];  //最短路径顶点集
   int i, j, k;   float w;
```

```
 for (i=0; i<n; i++) {
    dist[i]=G.GetWeight(v, i);  //数组初始化
    S[i]=0;
    if (i!=v && dist[i]<MaxValue)
        path[i]=v;
    else path[i]=-1;
}
S[v]=1;  dist[v]=0;  //顶点v加入顶点集合
//选当前不在集合S中具有最短路径的顶点u
for (i=0; i<n-1; i++) {
    float min=MaxValue;  int u=v;
```

```
        for (j=0; j<n; j++)
            if (!S[j] && dist[j]<min)
                { u=j;  min=dist[j]; }
        S[u]=1;  //将顶点u加入集合S
        for (k=0; k<n; k++) {  //修改
           w=G.GetWeight(u, k);
            if (!S[k] && w<MaxValue
                          && dist[u]+w<dist[k]) {
            //顶点k未加入S，且绕过u可以缩短路径
                dist[k]=dist[u]+w;
                path[k]=u;  //修改到k的最短路径
            }
    }
}
```

选 取	顶点 1			顶点 2			顶点 3			顶点 4		
终 点	*S*[1]	*d*[1]	*p*[1]	*S*[2]	*d*[2]	*p*[2]	*S*[3]	*d*[3]	*p*[3]	*S*[4]	*d*[4]	*p*[4]
0	0	**10**	0	0	∞	0	0	30	0	0	100	0
1	**1**	**10**	**0**	0	60	1	0	**30**	0	0	100	0
3	**1**	**10**	**0**	0	**50**	3	**1**	**30**	**0**	0	90	3
2	**1**	**10**	**0**	**1**	**50**	**3**	**1**	**30**	**0**	0	**60**	2
4	**1**	**10**	**0**	**1**	**50**	**3**	**1**	**30**	**0**	**1**	**60**	**2**



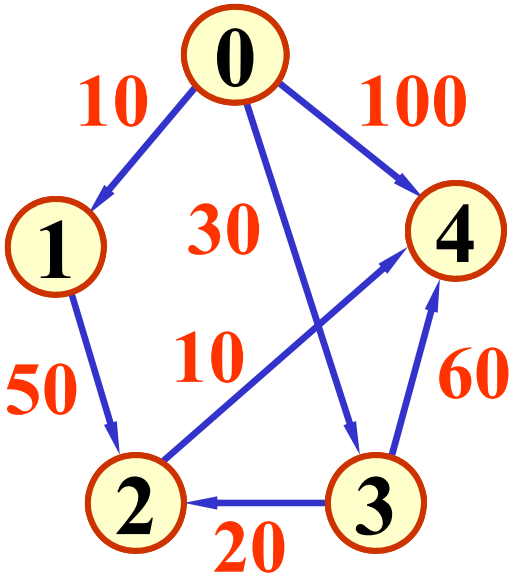
（a）带权有向图

$$\begin{array}{c} \begin{matrix} 0 & 1 & 2 & 3 & 4 \end{matrix} \\ \begin{pmatrix} 0 & 10 & \infty & 30 & 100 \\ \infty & 0 & 50 & \infty & \infty \\ \infty & \infty & 0 & \infty & 10 \\ \infty & \infty & 20 & 0 & 60 \\ \infty & \infty & \infty & \infty & 0 \end{pmatrix} \end{array} \begin{matrix} 0 \\ 1 \\ 2 \\ 3 \\ 4 \end{matrix}$$

（b）邻接矩阵

Dijkstra算法中各辅助数组的最终结果

序号	顶点 1	顶点 2	顶点 3	顶点 4
Dist	10	50	30	60
path	0	3	0	2



从表中读取源点0到终点v的最短路径的方法：举顶点4为例

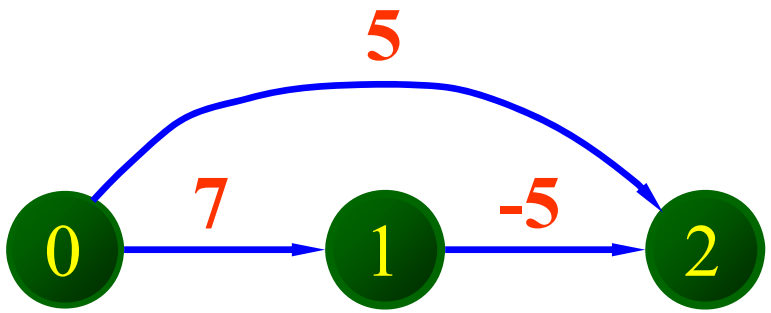
path[4]=2→path[2]=3→

path[3]=0，反过来排列，得到路径0, 3, 2, 4，这就是源点0到终点4的最短路径。

边上权值为任意值的单源最短路径问题

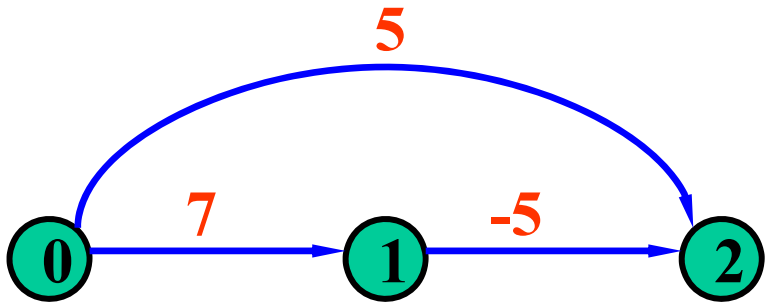
- 带权有向图***D***的某几条边或所有边的长度可能为负值，利用**Dijkstra**算法，不一定能得到正确的结果。

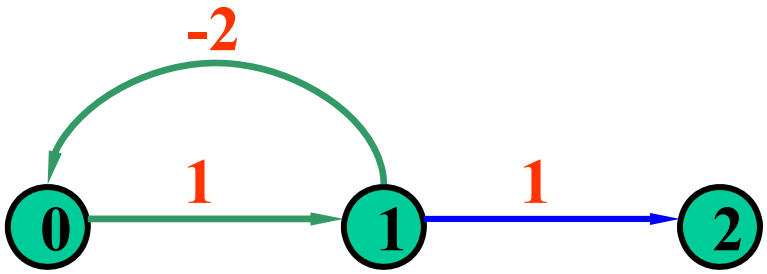
若设源点**v=0**，使用**Dijkstra**算法所得结果。



选取	顶点 0			顶点 1			顶点 2		
顶点	S[0]	d[0]	p[0]	S[1]	d[1]	p[1]	S[2]	d[2]	p[2]
0	1	0	-1	0	7	0	0	5	0
2	1	0	-1	0	7	0	1	5	0
1	1	0	-1	1	7	0	1	5	0

- 源点0到终点2的最短路径应是0，1，2，其长度为2，小于算法中计算出来的dist[2]值。
- Bellman和Ford提出了从源点逐次绕过其它顶点，以缩短到达终点的最短路径长度的方法。该方法有一个限制条件，即要求图中不能包含由带负权值的边组成的回路。





- 当图中没有由带负权值的边组成的回路时，有**n**个顶点的图中任意两个顶点之间如果存在最短路径，此路径最多有**n-1**条边。
- 我们将以此为依据考虑计算从源点**v**到其它顶点**u**的最短路径的长度**dist[u]**。
- **Bellman-Ford**方法构造一个最短路径长度数组序列$\mathbf{dist^1[u]}$, $\mathbf{dist^2[u]}$, …, $\mathbf{dist^{n-1}[u]}$。

其中，

- $dist^1[u]$是从源点v到终点u的只经过一条边的最短路径长度。

  $dist^1[u]=Edge[v][u]$

- $dist^2[u]$是从源点v最多经过两条边到达终点u的最短路径长度。
- $dist^3[u]$是从源点v出发最多经过不构成带负长度边回路的三条边到达终点u的最短路径的长度，…。
- $dist^{n-1}[u]$是从源点v出发最多经过不构成带负长度边回路的n-1条边到达终点u的最短路径的长度。

- 算法的最终目的是计算出$dist^{n-1}[u]$。

- 可以用递推方式计算$dist^k[u]$。

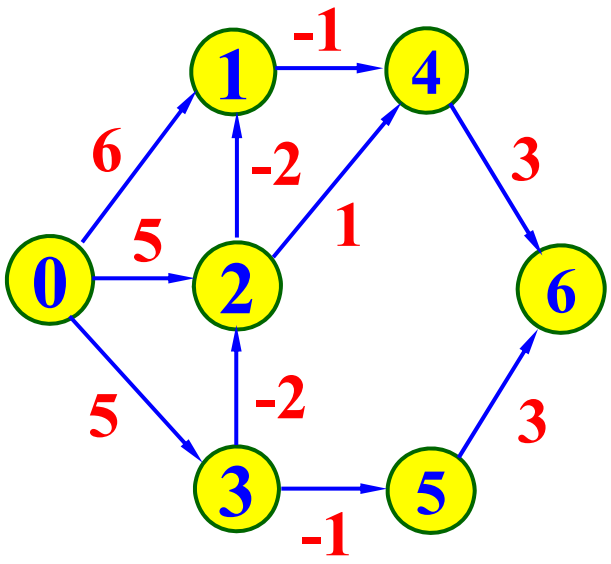
  $dist^1[u]=Edge[v][u]$;

  $dist^k[u]=min\ \{dist^{k-1}[u],$

  $\qquad\qquad min\{dist^{k-1}[j]+Edge[j][u]\}\}$

  - 设已经求出$dist^{k-1}[j]$, j=0, 1, …, n-1，此即从源点v最多经过不构成带负长度边回路的k-1条边到达终点j的最短路径长度。
  - 计算$min\{dist^{k-1}[j]+Edge[j][u]\}$，可得从源点v绕过各顶点j，最多经过不构成带负长度边回路的k条边到达终点u的最短路径长度。用它与$dist^{k-1}[u]$比较，取小者作为$dist^k[u]$的值。

图的最短路径长度



k	$d^k[0]$	$d^k[1]$	$d^k[2]$	$d^k[3]$	$d^k[4]$	$d^k[5]$	$d^k[6]$
1	0	6	5	5	∞	∞	∞
2	0	3	3	5	5	4	∞
3	0	1	3	5	2	4	7
4	0	1	3	5	0	4	5
5	0	1	3	5	0	4	3
6	0	1	3	5	0	4	3

计算最短路径的Bellman和Ford算法

```
void BellmanFord(Graph <float> &G, int v,
                 float dist[ ], int path[ ]) {
//在带权有向图中有的边具有负的权值
//从顶点v找到所有其它顶点的最短路径
    int n=G.NumberOfVertices( );
    for (int i=0; i<n; i++) {
        dist[i]=G.GetWeight(v, i);
        if (i!=v && dist[i]<MaxValue)
            path[i]=v;
        else path[i]=-1;
    }
```

```
    for (int k=2; k<n; k++)
        for (int u=0; u<n; u++)
            if (u!=v)
                for (i=0; i<n; i++) {
                    Type w=G.GetWeight(i, u);
                    if (w<MaxValue &&
                                dist[u]>dist[i]+w) {
                        dist[u]=dist[i]+w;
                        path[u]=i;
                    }
                }
}
```

所有顶点之间的最短路径

- 问题的提法：已知一个各边权值均大于0的带权有向图，对每一对顶点$v_i \neq v_j$，要求求出v_i与v_j之间的最短路径和最短路径长度。
- **Floyd算法的基本思想**

定义一个n阶方阵序列：

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

其中，$A^{(-1)}[i][j]=Edge[i][j]$；

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

$$k=0, 1, \ldots, n-1$$

- $A^{(0)}[i][j]$是从顶点v_i到v_j，中间顶点是v_0的最短路径的长度;
- $A^{(k)}[i][j]$是从顶点v_i到v_j，中间顶点的序号不大于k的最短路径的长度;
- $A^{(n-1)}[i][j]$是从顶点v_i到v_j的最短路径长度。

- **Floyd算法的基本思想**
  - 递推地产生一个矩阵序列$A^{(0)}$，$A^{(1)}$，$A^{(2)}$，…，$A^{(k)}$，…，$A^{(n)}$，其中$A^{(0)}$为给定的代价邻接矩阵，$A^{(k)}(i, j)$ $(1\le i, j\le n)$表示从顶点i到顶点j的中间顶点序号不大于k的最短路径的长度。
  - 若从i到j的路径没有中间顶点，则对于$1\le k\le n$，有$A^{(k)}(i, j)=A^{(0)}(i, j)=Cost(i, j)$。
  - 递推地产生$A^{(0)}$，$A^{(1)}$，$A^{(2)}$，…，$A^{(k)}$，…，$A^{(n)}$的过程就是逐步允许越来越多的顶点作为路径的中间顶点，直至找到所有允许作为中间顶点的顶点，算法结束而求出最短路径。

- 假设已求出$A^{(k-1)}(i, j)$ $(1\le i, j\le n)$，怎样求出$A^{(k)}(i, j)$？
  - 如果从顶点i到顶点j的最短路径不经过顶点k，则由$A^{(k)}(i, j)$的定义可知，从i到j的中间顶点序号不大于k的最短路径长度就是$A^{(k-1)}(i, j)$，即$A^{(k)}(i, j)=A^{(k-1)}(i, j)$。
  - 如果从顶点i到顶点j的最短路径经过顶点k，则这样的一条路径是由i到k和由k到j的两条路径所组成。由于$A^{(k-1)}(i, k)+A^{(k-1)}(k, j)<A^{(k-1)}(i, j)$，则$A^{(k)}(i, j)=A^{(k-1)}(i, k)+A^{(k-1)}(k, j)$。

- 计算$A^{(k)}(i, j)$的递推公式
  - 对于$1 \le i, j \le n$

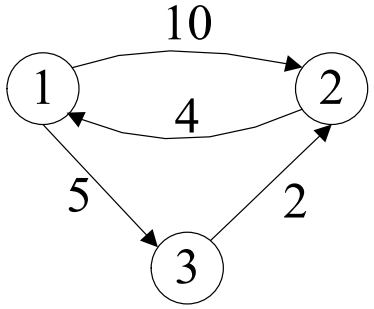
$$\begin{cases} A^{(0)}(i,j) = Cost(i,j) \\ A^{(k)}(i,j) = \min\left\{A^{(k-1)}(i,j), A^{(k-1)}(i,k) + A^{(k-1)}(k,j)\right\} \\ 1 \le k \le n \end{cases}$$

- 具体实现
  - 用MAXINT=9999表示∞，输出时用*表示∞。
  - 函数Floyd求出每对顶点之间的最短路径，主要数组A与Path在函数Floyd执行之后，A[i][j]存放从顶点i到顶点j的最短路径的长度，若A[i][j]=MAXINT=9999，则i到j没有路径；Path[i][j]存放从i到j的最短路径的一个中间顶点，若Path[i][j]=0，则没有中间顶点。
  - 函数Print_All_Path输出由n*n阶邻接矩阵表示的带权有向图中的所有最短路径。

程序8-7 最短路径的Floyd算法

```
const int MAXINT=9999;
const int VerticesMaxNumber=10;  //图中最大顶点个数
class Graph {  //图的类定义
    private:
        float Cost[VerticesMaxNumber][VerticesMaxNumber];
        //图的邻接矩阵
        float A[VerticesMaxNumber][VerticesMaxNumber];
        //存放顶点之间的最短路径长度
        int Path[VerticesMaxNumber][VerticesMaxNumber];
        //存放在最短路径上终结顶点的前一顶点的顶点号
    public:
         void Floyd(int n);
         ……
};
```

```
void Graph :: Floyd(int n)
{  //Cost[n][n]是一个具有n个顶点的图的邻接矩阵
   //A[i][j]是顶点i和j之间的最短路径长度
   //Path[i][j]是相应路径上顶点j前一顶点的顶点号
     int i, j, k;
     for (i=1; i<=n; i++)
         for (j=1; j<=n; j++)  //矩阵A与数组Path初始化
     {
           A[i][j]=Cost[i][j];
           if (i!=j && A[i][j]<MAXINT)  //i和j之间有路径
                Path[i][j]=i;
           else Path[i][j]=0;  //i和j之间无有路径
      }
      for (k=1; k<=n; k++)
          for (i=1; i<=n; i++)
              for (j=1; j<=n; j++)  //产生A(k)及相应的Path(k)
                  if (A[i][k]+A[k][j]<A[i][j])
                  {  //缩短路径长度，绕过k到j
                       A[i][j]=A[i][k]+A[k][j];
                       Path[i][j]=k;
                   }
}
```



$$Cost = \begin{bmatrix} 0 & 10 & 5 \\ 4 & 0 & \infty \\ \infty & 2 & 0 \end{bmatrix}$$

图8-24 一个带权有向图及其代价邻接矩阵

$A^{(0)}$	1	2	3
1	0	10	5
2	4	0	∞
3	∞	2	0

$Path^{(0)}$	1	2	3
1	0	0	0
2	0	0	0
3	0	0	0

$A^{(1)}$	1	2	3
1	0	10	5
2	4	0	9
3	∞	2	0

$Path^{(1)}$	1	2	3
1	0	0	0
2	0	0	1
3	0	0	0

$A^{(2)}$	1	2	3
1	0	10	5
2	4	0	9
3	6	2	0

$Path^{(2)}$	1	2	3
1	0	0	0
2	0	0	1
3	2	0	0

$A^{(3)}$	1	2	3
1	0	7	5
2	4	0	9
3	6	2	0

$Path^{(3)}$	1	2	3
1	0	3	0
2	0	0	1
3	2	0	0

图8-25 A和Path的变化情况

从顶点1到顶点2的最短路径为(1, 3, 2)，其长度为7。

从顶点1到顶点3的最短路径为(1, 3)，其长度为5。

从顶点2到顶点1的最短路径为(2, 1)，其长度为4。

从顶点2到顶点3的最短路径为(2, 1, 3)，其长度为9。

从顶点3到顶点1的最短路径为(3, 2, 1)，其长度为6。

从顶点3到顶点2的最短路径为(3, 2)，其长度为2。

显然，上面函数的执行时间为$O(n^3)$。

所有各对顶点之间的最短路径

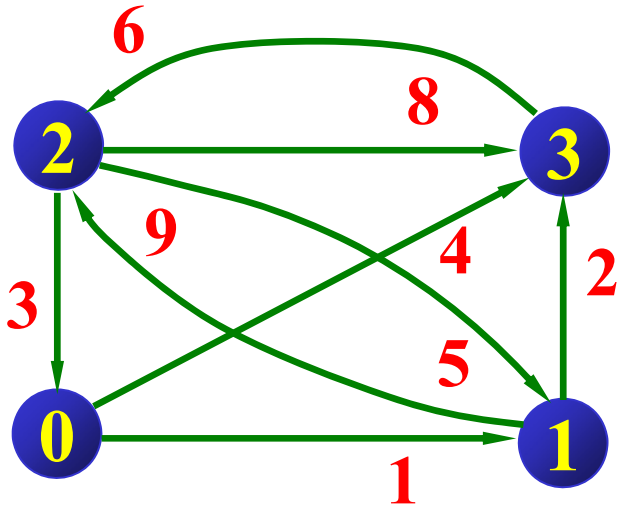
```
void AllLengths(Graph <float> &G,
                                float a[ ][ ], int path[ ][ ]) {
//a[i][j]是顶点i和j之间的最短路径长度
//path[i][j]是相应路径上顶点j的前一顶点的顶点号
    int i, j, k, n=G.NumberOfVertices( );
    for (i=0; i<n; i++)  //矩阵a与path初始化
       for (j=0; j<n; j++) {
          a[i][j]=G.GetWeight(i, j);
          if (i!=j && a[i][j]<MaxValue)
             path[i][j]=i;  //i到j有路径
          else path[i][j]=0;  //i到j无路径    }
```

```
    for (k=0; k<n; k++)
    //针对每一个k，产生a(k)及path(k)
        for (i=0; i<n; i++)
            for (j=0; j<n; j++)
                if (a[i][k]+a[k][j]<a[i][j]) {
                    a[i][j]=a[i][k]+a[k][j];
                    path[i][j]=path[k][j];
                     //缩短路径长度，绕过k到j
                }
}
```

Floyd算法允许图中有带负权值的边，但不许有包含带负权值的边组成的回路。

	$A^{(-1)}$				$A^{(0)}$				$A^{(1)}$				$A^{(2)}$				$A^{(3)}$			
	0	*1*	*2*	*3*	*0*	*1*	*2*	*3*	*0*	*1*	*2*	*3*	*0*	*1*	*2*	*3*	*0*	*1*	*2*	*3*
0	0	1	∞	4	0	1	∞	4	0	1	10	3	0	1	10	3	0	1	9	3
1	∞	0	9	2	∞	0	9	2	∞	0	9	2	12	0	9	2	11	0	8	2
2	3	5	0	8	3	4	0	7	3	4	0	6	3	4	0	6	3	4	0	6
3	∞	∞	6	0	∞	∞	6	0	∞	∞	6	0	9	10	6	0	9	10	6	0

	$Path^{(-1)}$				$Path^{(0)}$				$Path^{(1)}$				$Path^{(2)}$				$Path^{(3)}$			
	0	*1*	*2*	*3*	*0*	*1*	*2*	*3*	*0*	*1*	*2*	*3*	*0*	*1*	*2*	*3*	*0*	*1*	*2*	*3*
0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	1	1	0	0	3	1
1	0	0	1	1	0	0	1	1	0	0	1	1	2	0	1	1	2	0	3	1
2	2	2	0	2	2	0	0	0	2	0	0	1	2	0	0	1	2	0	0	1
3	0	0	3	0	0	0	3	0	0	0	3	0	2	0	3	0	2	0	3	0

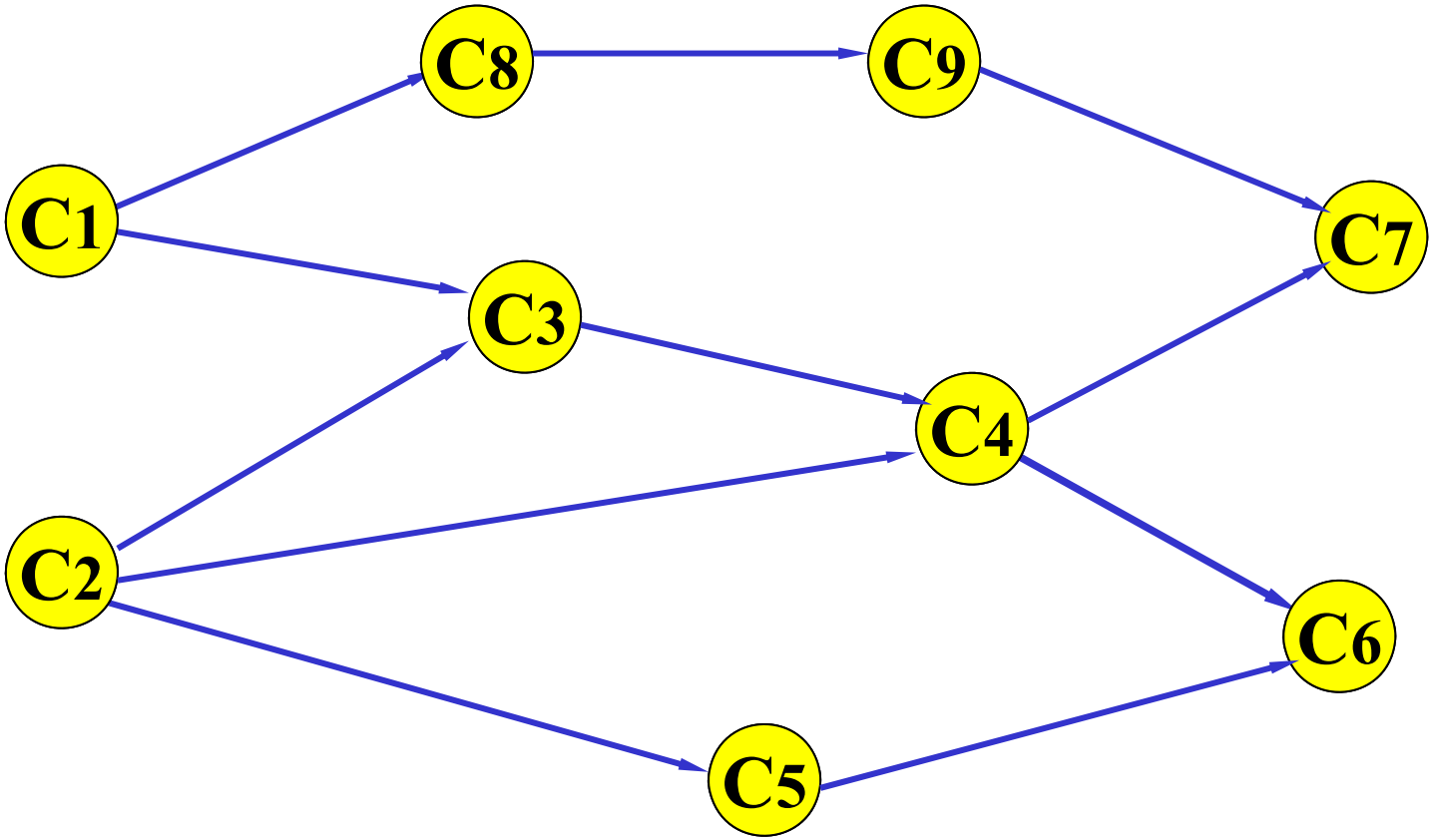


- 以$Path^{(3)}$为例，对最短路径读法加以说明。从 $A^{(3)}$ 知，点 1 到 0 的最短路径长度为 $a[1][0]=11$，其最短路径看 path[1][0]=2, path[1][2]=3, path[1][3]=1，表示顶点0←顶点2←顶点3←顶点1；从顶点1到顶点0最短路径为<1, 3>, <3, 2>, <2, 0> 。
- 求解最短路径的算法不仅适用于带权有向图，对带权无向图也可适用。因为带权无向图可以看作是有往返二重边的有向图。

8.6 拓扑排序

- 计划、施工过程、生产流程、程序流程等都是“工程”。除很小的工程外，一般都把工程分为若干个叫做“活动”的子工程。完成这些活动，这个工程就可以完成。
- 例如，计算机专业学生的学习就是一个工程，每一门课程的学习就是整个工程的一些活动。其中，有些课程要求先修课程，有些则不要求。这样，在有的课程之间有领先关系，有的课程可以并行地学习。

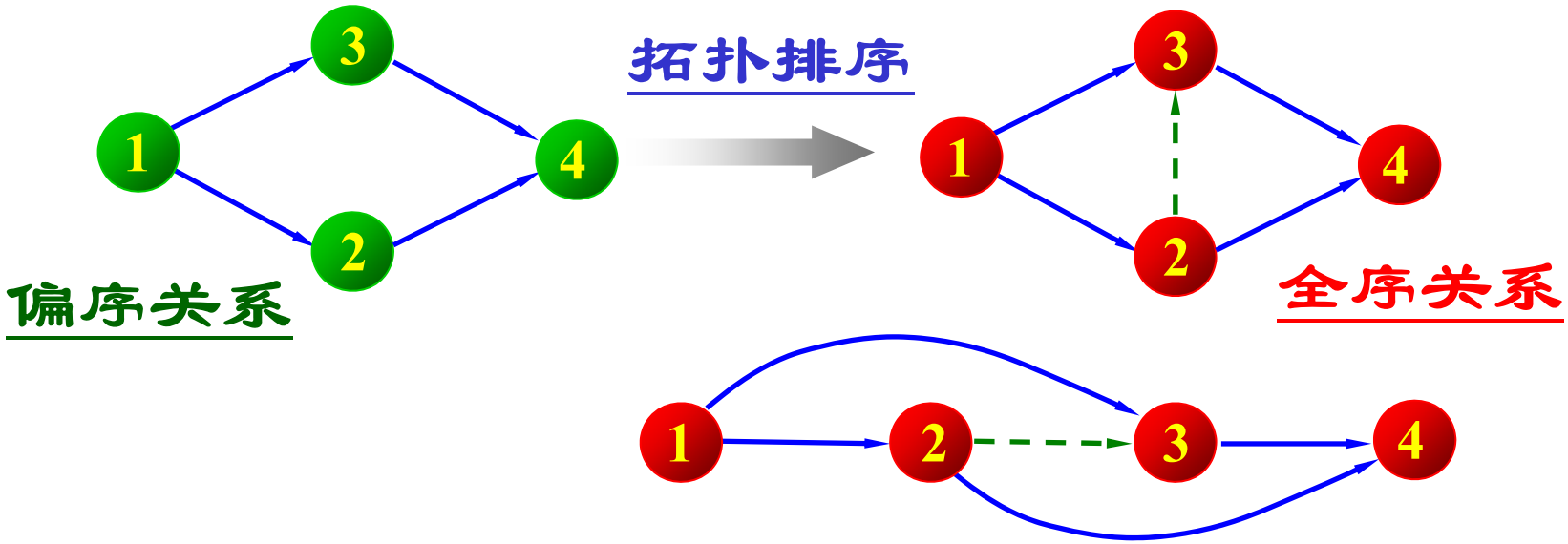
课程代号	课程名称	先修课程
C_1	高等数学	
C_2	程序设计基础	
C_3	离散数学	C_1, C_2
C_4	数据结构	C_3, C_2
C_5	高级语言程序设计	C_2
C_6	编译方法	C_5, C_4
C_7	操作系统	C_4, C_9
C_8	普通物理	C_1
C_9	计算机原理	C_8



学生课程学习工程图

- 可以用有向图表示一个工程，在这种有向图中，用顶点表示活动，用有向边<V_i, V_j>表示活动V_i必须先于活动V_j进行。这种有向图叫做顶点表示活动的**AOV**网络**(Activity On Vertices)**。
- 在**AOV**网络中不能出现有向回路，即有向环。如果出现了有向环，则意味着某项活动应以自己作为先决条件。
- 因此，对给定的**AOV**网络，必须先判断它是否存在有向环。

- 检测有向环的一种方法是对AOV网络构造它的拓扑有序序列。即将各个顶点（代表各个活动）排列成一个线性有序的序列，使得AOV网络中所有应存在的前驱和后继关系都能得到满足。

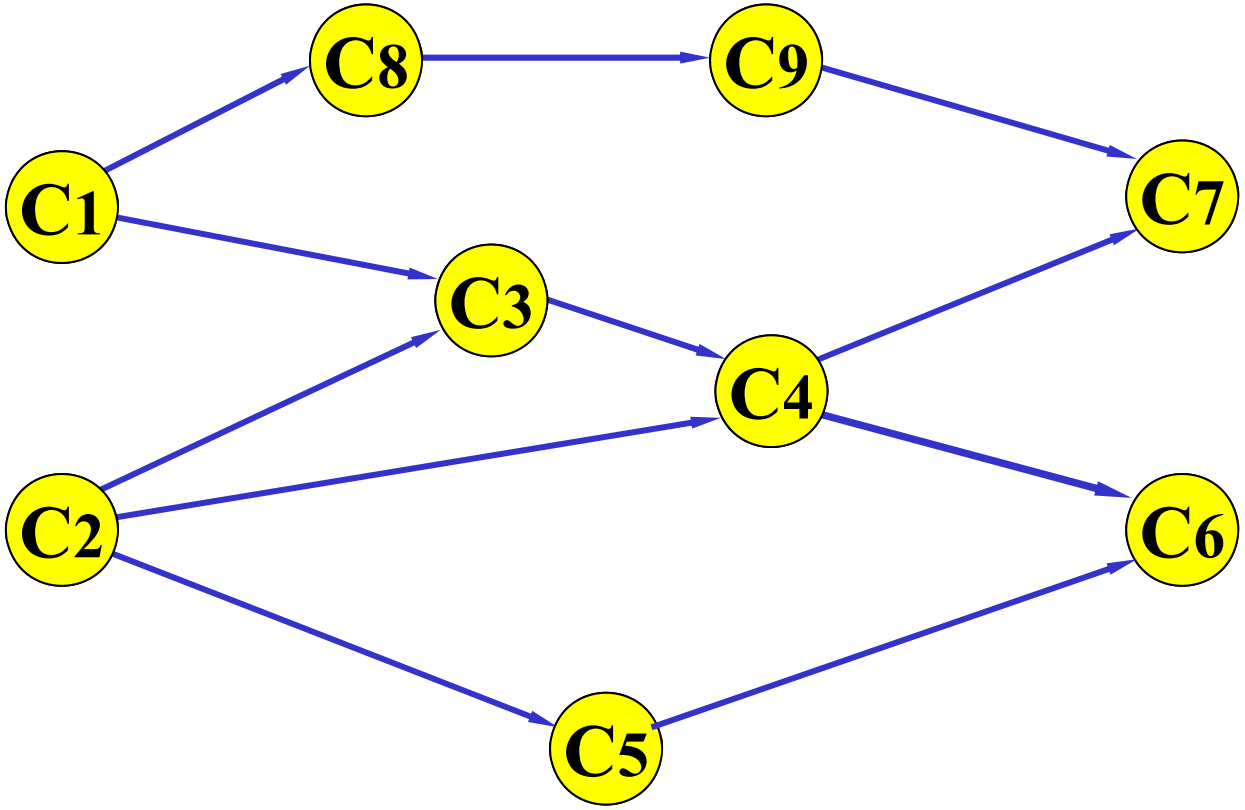


- 这种构造AOV网络全部顶点的拓扑有序序列的运算就叫做拓扑排序。
- 如果通过拓扑排序能将AOV网络的所有顶点都排入一个拓扑有序的序列中，则该网络中必定不会出现有向环。
- 如果AOV网络中存在有向环，此AOV网络所代表的工程是不可行的。

- 例如，对学生选课工程图进行拓扑排序，得到的拓扑有序序列为：

$C_1, C_2, C_3, C_4, C_5, C_6, C_8, C_9, C_7$

或 $C_1, C_8, C_9, C_2, C_5, C_3, C_4, C_7, C_6$



进行拓扑排序的方法

①输入**AOV**网络，令***n***为顶点个数。

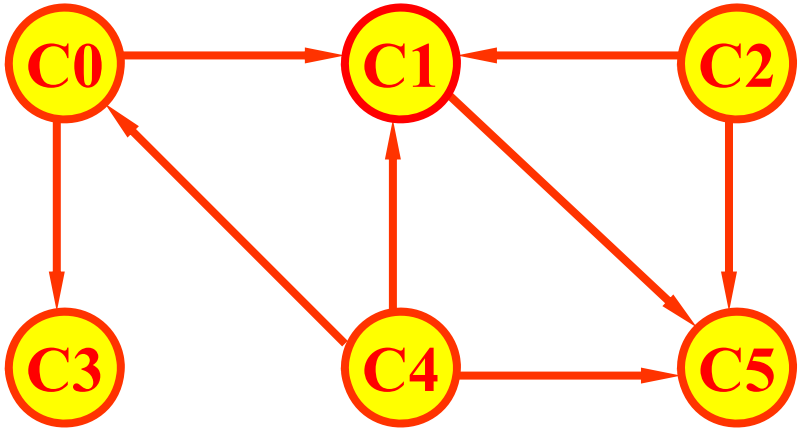
②在**AOV**网络中选一个没有直接前驱的顶点，并输出之；

③从图中删去该顶点，同时删去所有它发出的有向边；

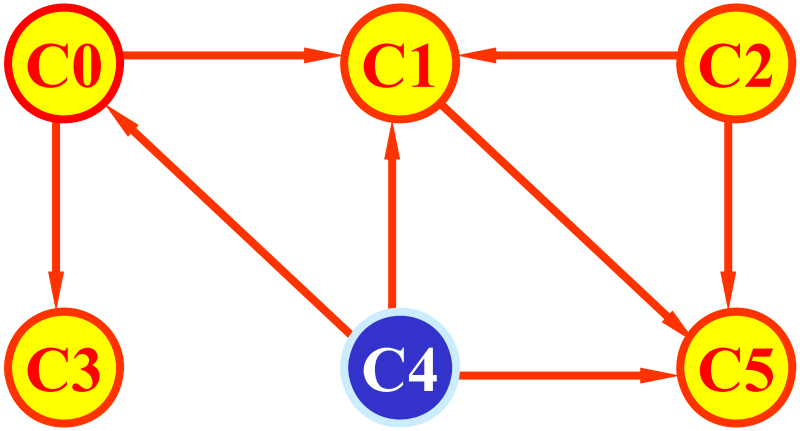
④重复以上 ②、③步，直到

- ☞全部顶点均已输出，拓扑有序序列形成，拓扑排序完成；
- ☞或图中还有未输出的顶点，但已跳出处理循环。说明图中还剩下一些顶点，它们都有直接前驱，这时网络中必存在有向环。

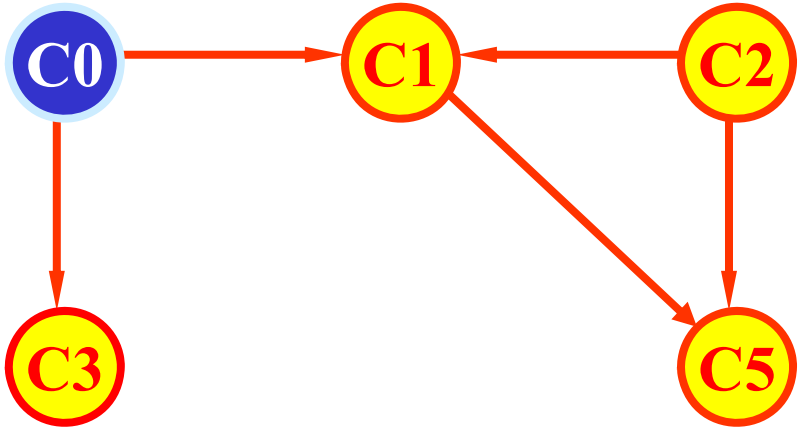
拓扑排序的过程



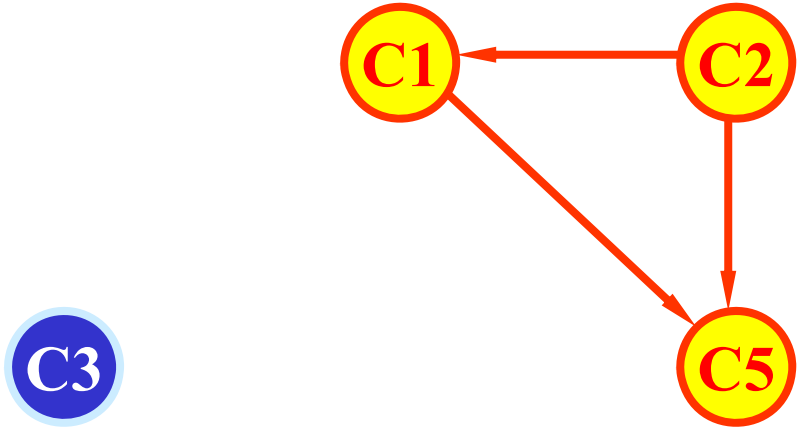
(a) 有向无环图



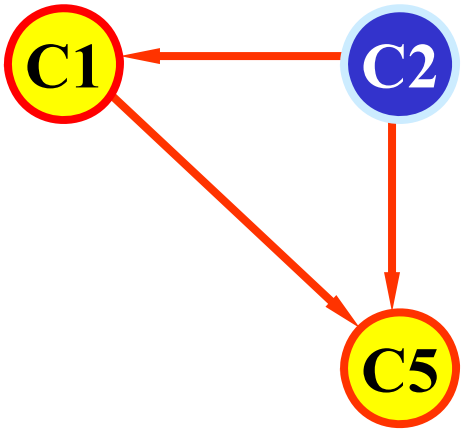
(b) 输出顶点C4



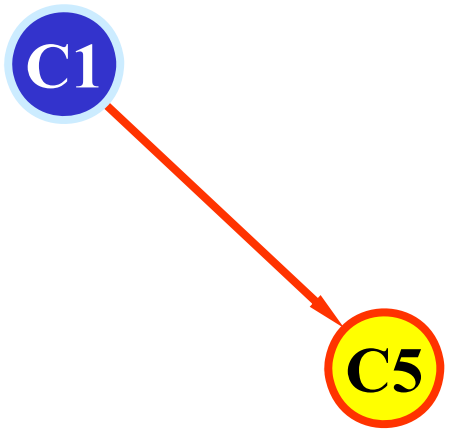
(c) 输出顶点C0



(d) 输出顶点C3



(e) 输出顶点C2



(f) 输出顶点C1

(g) 输出顶点C5

(h) 拓扑排序完成

最后得到的拓扑有序序列为C_4, C_0, C_3, C_2, C_1, C_5，它满足图中给出的所有前驱和后继关系。对于本来没有这种关系的顶点，如C_4和C_2，也排出先后次序关系。

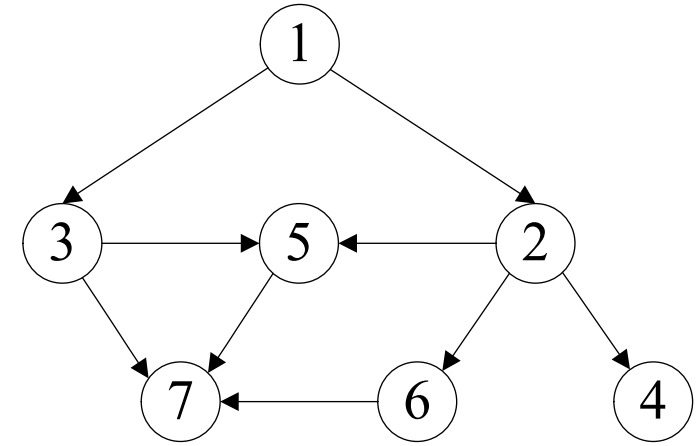
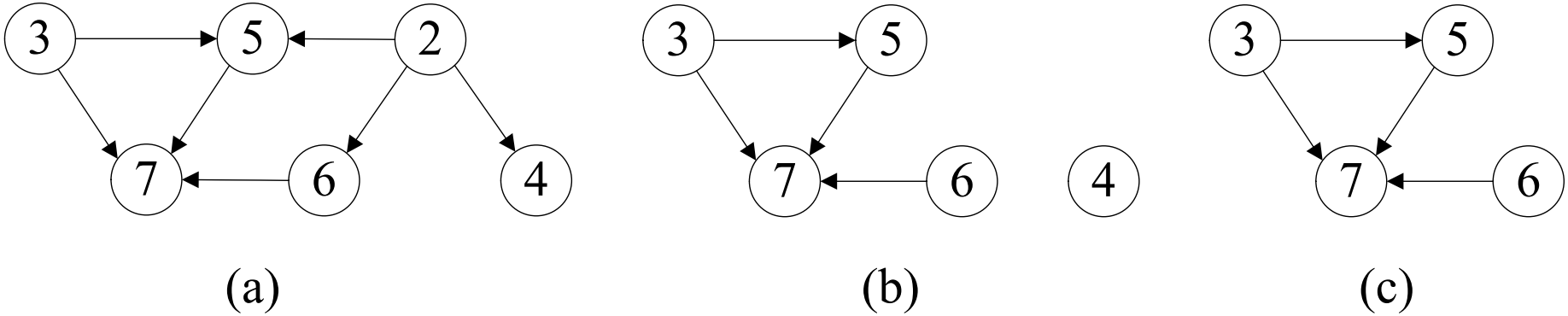


图8-27 有向图G_6



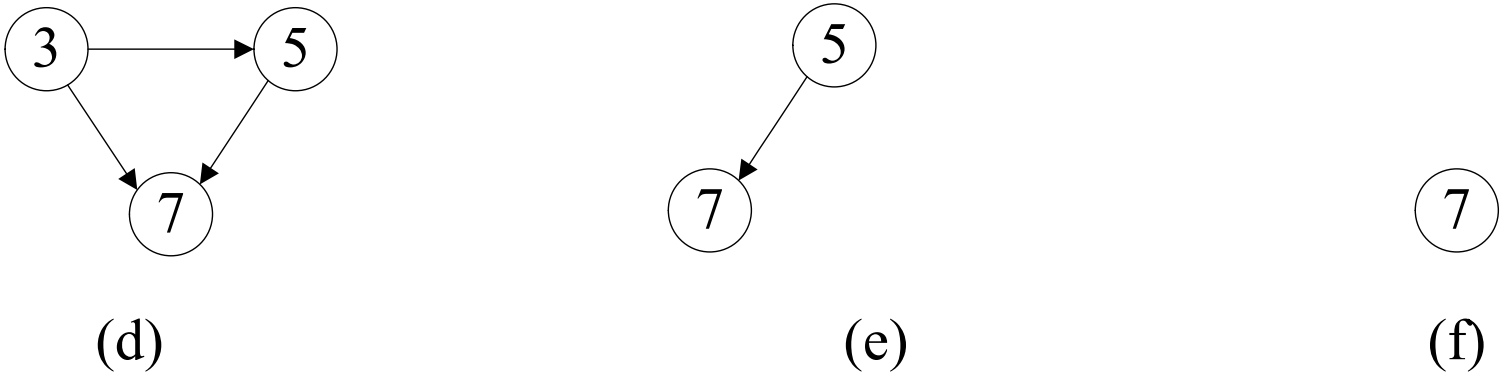
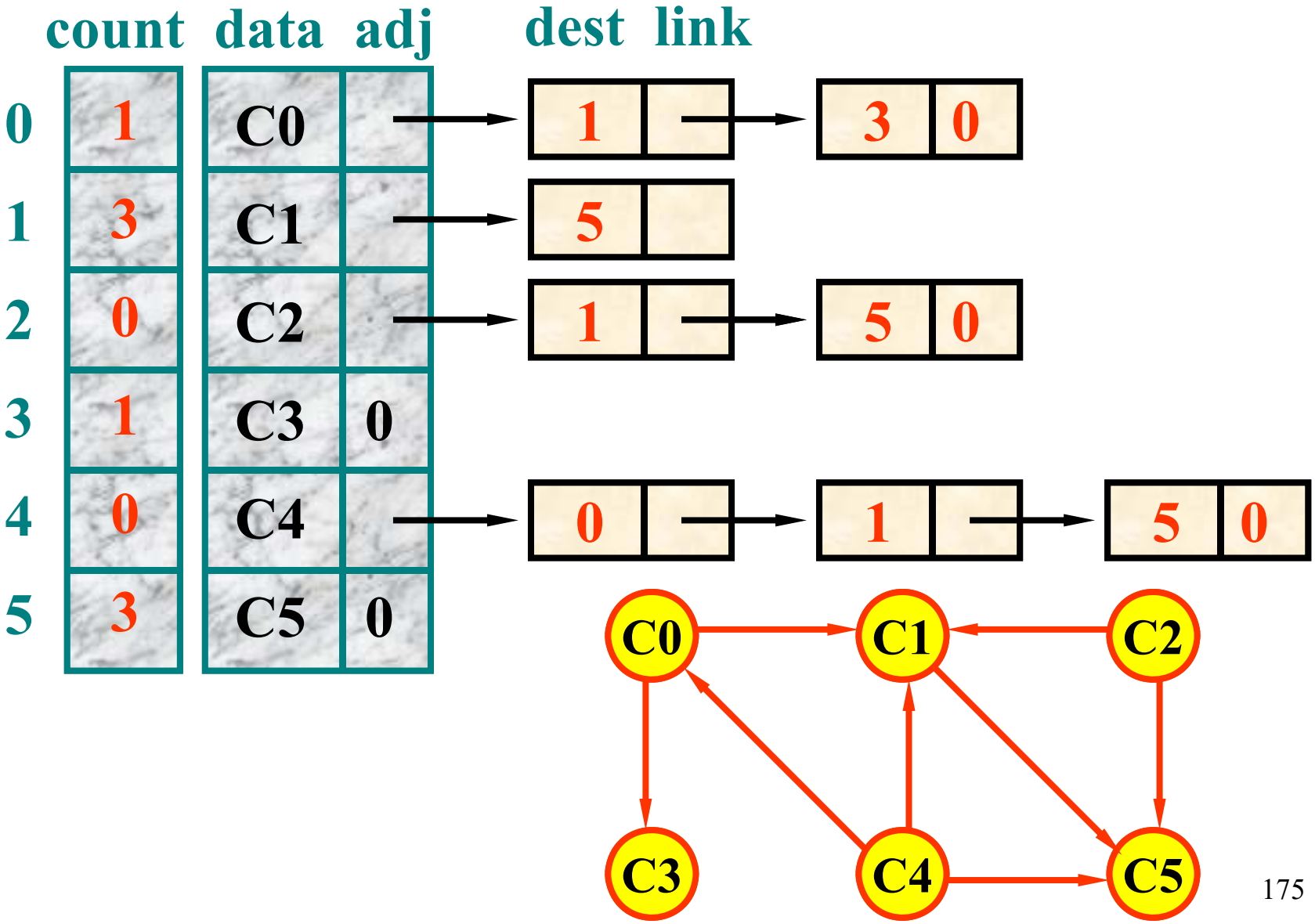


图8-29 产生G_6的拓扑序列的过程

AOV网络及其邻接表表示



- 在邻接表中增设一个数组**count[ ]**，记录各顶点入度，入度为零的顶点即无前驱顶点。
- 在输入数据前，顶点表**NodeTable[ ]**和入度数组**count[ ]**全部初始化。在输入数据时，每输入一条边**<j, k>**，就需要建立一个边结点，并将它链入相应边链表中，统计入度信息：

```
Edge *p=new Edge <int> (k);
//建立边结点，dest域赋为k
p->link=NodeTable[j].adj;
NodeTable[j].adj=p;
//链入顶点j的边链表的前端
count[k]++; //顶点k入度加一
```

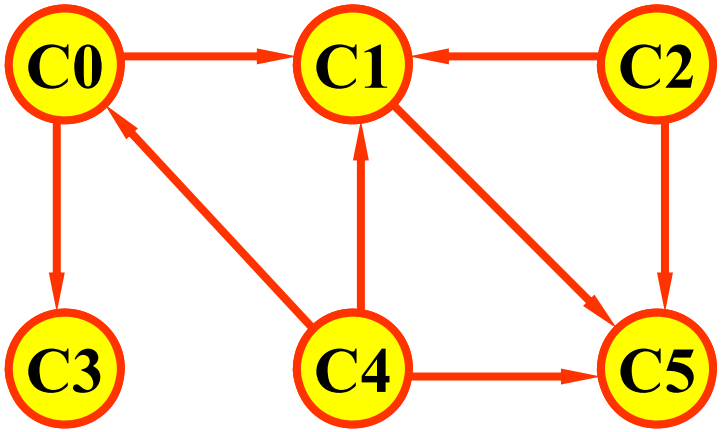
- 在算法中，使用一个存放入度为零的顶点的链式栈，供选择和输出无前驱的顶点。
- 拓扑排序算法可描述如下：
  - 建立入度为零的顶点栈；
  - 当入度为零的顶点栈不空时，重复执行：
    - 从顶点栈中退出一个顶点，并输出之；
    - 从**AOV**网络中删去这个顶点和它发出的边，边的终顶点入度减一；
    - 如果边的终顶点入度减至**0**，则该顶点进入度为零的顶点栈。
  - 如果输出顶点个数少于**AOV**网络的顶点个数，则报告网络中存在有向环。

- 在算法实现时，为建立入度为零的顶点栈，可以不另外分配存储空间，直接利用入度为零的顶点的**count[ ]**数组元素。设立一个栈顶指针**top**指示当前栈顶位置，即某一个入度为零的顶点。栈初始化时，置 **top=-1** 。
- 将顶点**i**进栈时执行以下指针的修改：

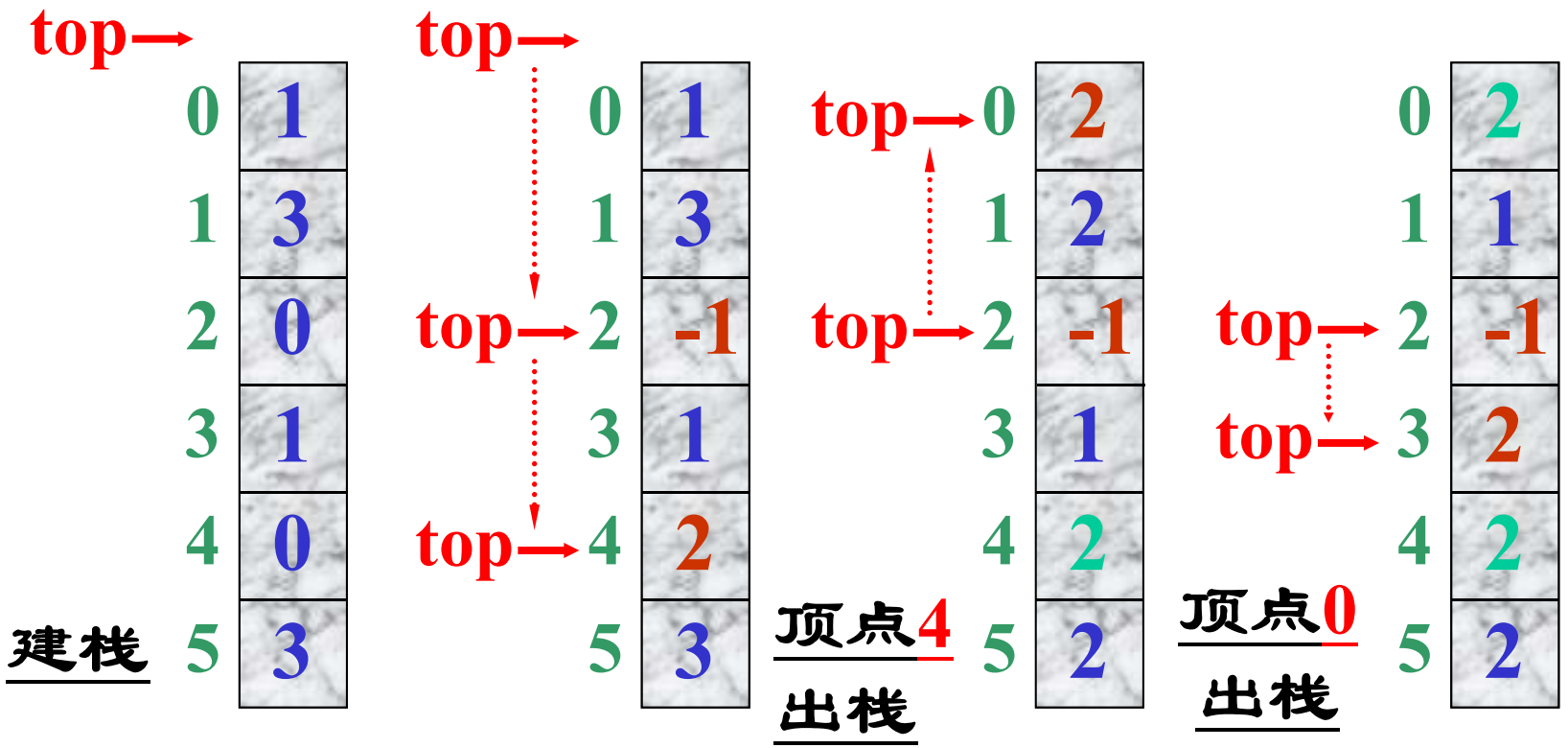
```
count[i]=top; top=i;
//top指向新栈顶i, 原栈顶元素在count[i]中
```

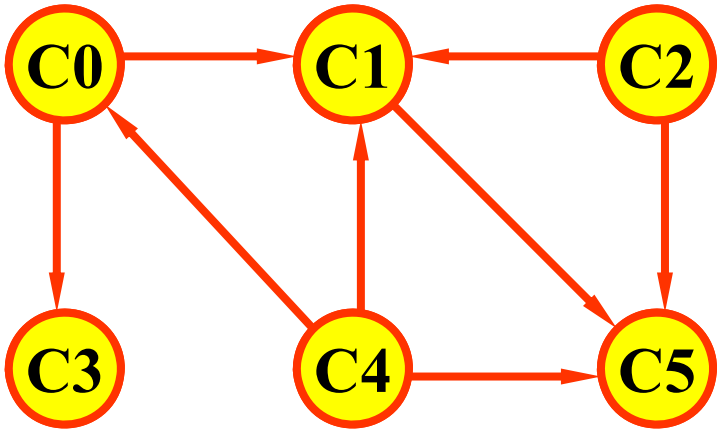
- 退栈操作可以写成：

```
j=top; top=count[top];
//位于栈顶的顶点位置记为 j, top退到次栈顶
```

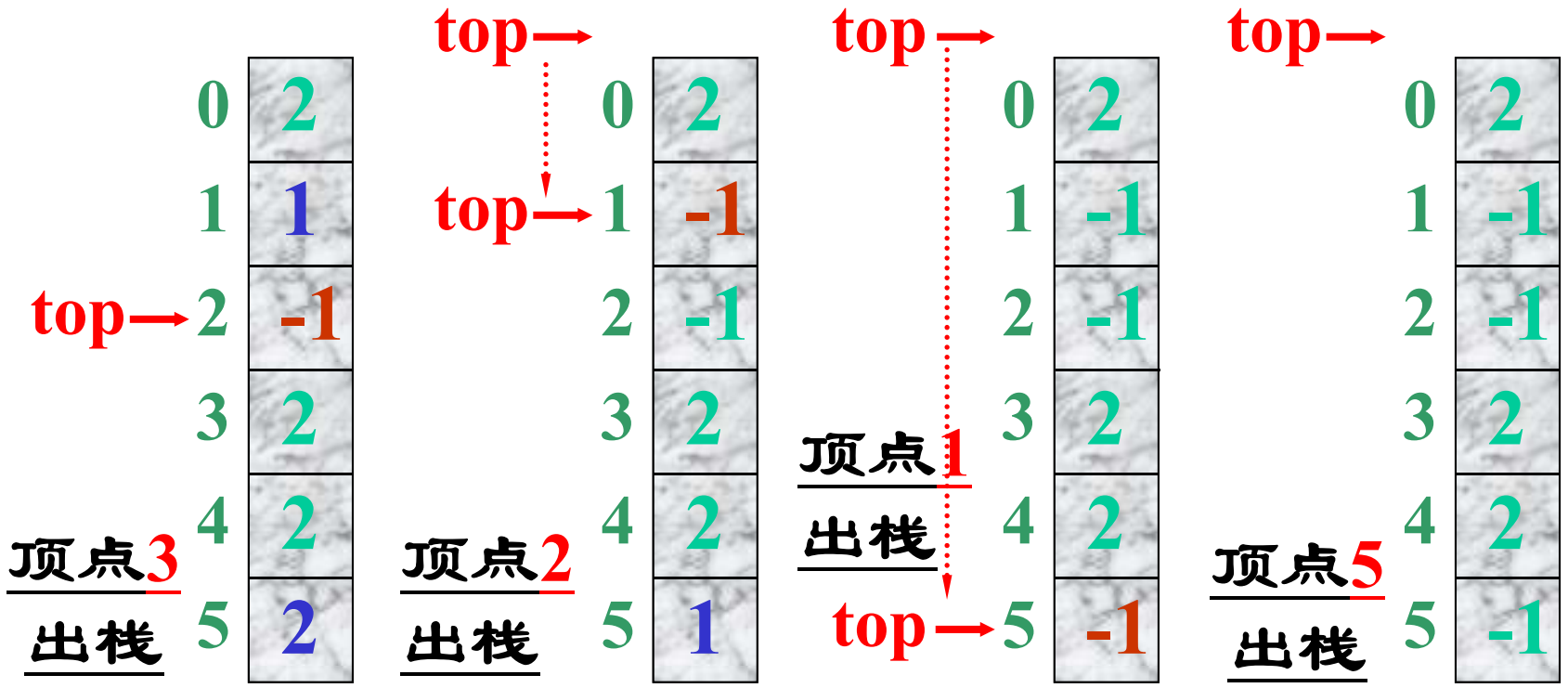


拓扑排序时入度为零的顶点栈在count[ ]中的变化





拓扑排序时入度为零的顶点栈在count[ ]中的变化



程序8-9 拓扑排序

```
class Graph {
    friend class <int, float> VertexNode;
    friend class <float> EdgeNode;
    private:
        VertexNode <int, float> *NodeTable;
        //邻接表的顶点表，设顶点数据及边上的权均为整数
        int *count; //入度数组，用于记录各顶点的入度
        int n; //顶点个数
    public:
        Graph(const int vertices=0): n(vertices)
        {
            NodeTable=new VertexNode <int, float> [n];
            //建立顶点表数组
            count=new int [n]; //建立入度数组
         }
         void Topppol_Order( );
};
```

```
void Graph :: Toppol_Order( )
{ //对n个顶点的AOV网络进行拓扑排序
  //其中top是入度为零顶点栈的栈顶指针
    int top=-1; //建立空栈
    for (int i=0; i<n; i++) //建立入度为零顶点的链接栈
        if (count[i]==0)
        { //入度为零的顶点依次进栈
            count[i]=top;
            top=i;
        }
     for (i=0; i<n; i++)
        if (top==-1)
        { //存在有向环，未能形成拓扑序列，直接返回
                cout<<"Network has a cycle!"<<endl;
                return;
         }
```

```
        else {
            int j=top;
            top=count[top]; //入度为零的顶点出栈
            cout<<j<<endl; //输出入度为零的顶点信息
            EdgeNode <float> *l=NodeTable[j].adj;
            //顶点j的边链表的头指针
            while (l)
            { //l≠0, 存在出边
                int k=l.dest;
                //取该边的终止顶点k, 该顶点的入度减1
                if (--count[k]==0)
                { //入度减1为零的顶点进栈
                    count[k]=top;
                    top=k;
                }
                l=l->link; //取顶点j的下一条边出边
            }
        }
}
```

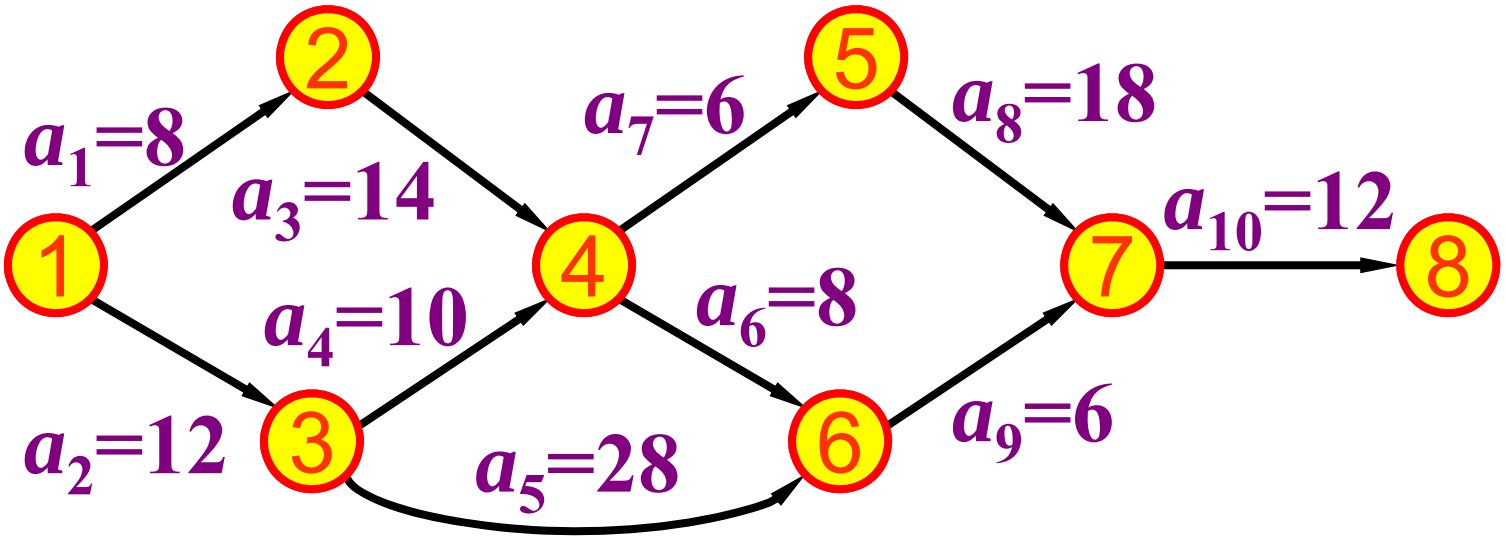
- 分析此拓扑排序算法可知，如果AOV网络有n个顶点，e条边，在拓扑排序的过程中，搜索入度为零的顶点，建立链式栈所需要的时间是O(n)。在正常的情况下，有向图有n个顶点，每个顶点进一次栈，出一次栈，共输出n次。顶点入度减一的运算共执行e次。所以总的时间复杂度为O(n+e)。

8.7 关键路径

- 如果在无有向环的带权有向图中，用有向边表示一个工程中的活动(Activity)，用边上权值表示活动持续时间(Duration)，用顶点表示事件(Event)，则这样的有向图叫做用边表示活动的网络，简称AOE (Activity On Edges)网络。
- AOE网络在某些工程估算方面非常有用。例如，可以使人们了解：
  - 完成整个工程至少需要多少时间（假设网络中没有环）？
  - 为缩短完成工程所需的时间，应当加快哪些活动？

- 从源点到各个顶点，以至从源点到汇点的有向路径可能不止一条，这些路径的长度也可能不同。完成不同路径的活动所需的时间虽然不同，但只有各条路径上所有活动都完成了，整个工程才算完成。
- 因此，完成整个工程所需的时间取决于从源点到汇点的最长路径长度，即在这条路径上所有活动的持续时间之和。这条路径长度最长的路径就叫做关键路径**(Critical Path)**。

- 要找出关键路径，必须找出关键活动，即不按期完成就会影响整个工程完成的活动。
- 关键路径上的所有活动都是关键活动，因此只要找到关键活动，就可以找到关键路径。
  例如，下图就是一个**AOE**网。



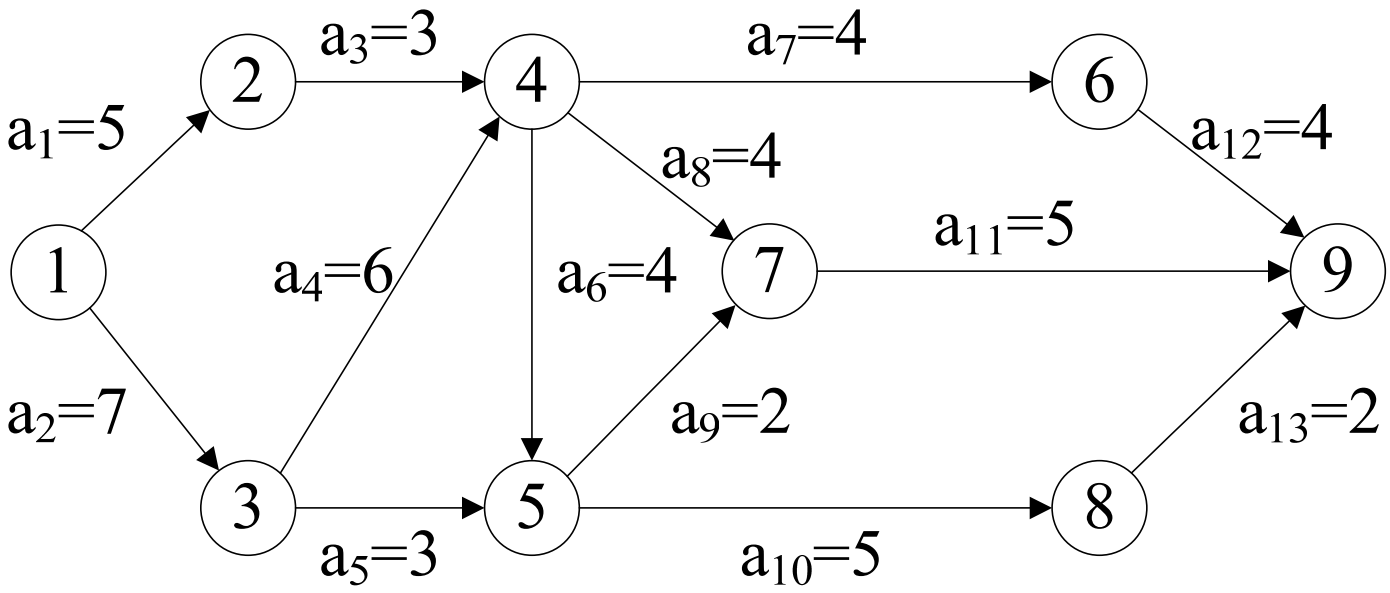


图8-31 一个假象工程的AOE-网络

v_1——表示工程开始
v_2——表示活动a_1完成
v_3——表示活动a_2完成
v_4——表示活动a_3和a_4完成
v_5——表示活动a_5和a_6完成

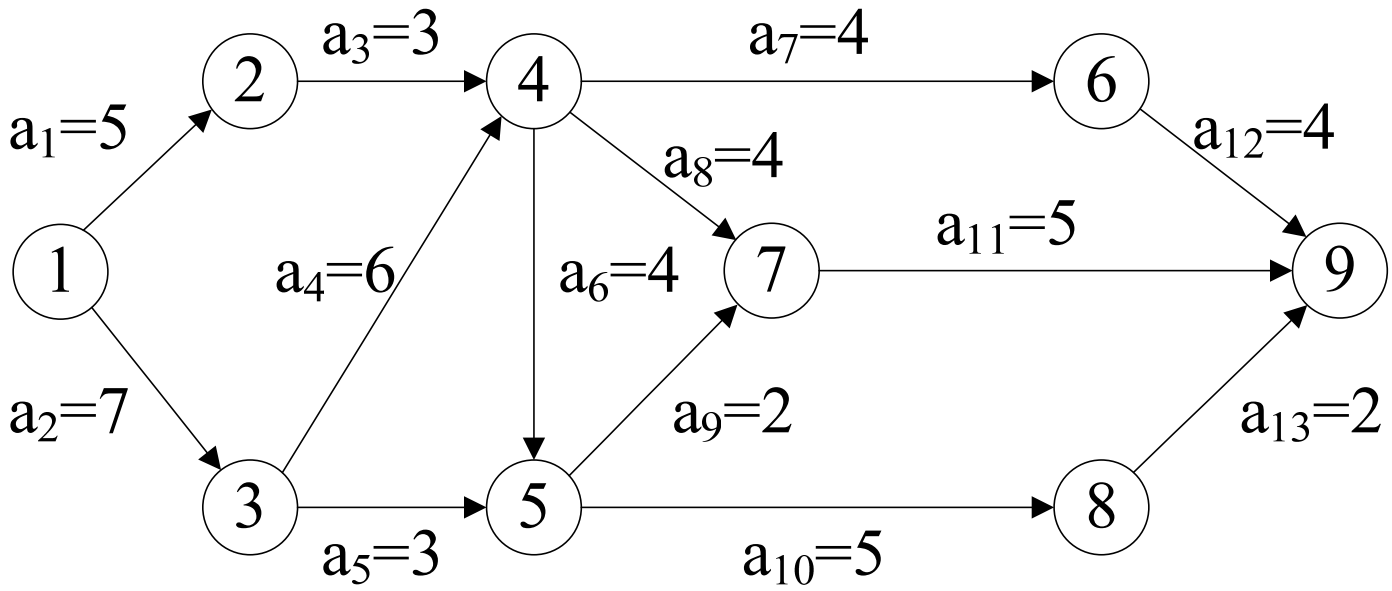


图8-31 一个假象工程的AOE-网络

v_6——表示活动a_7完成

v_7——表示活动a_8和a_9完成

v_8——表示活动a_{10}完成

v_9——表示活动a_{11}, a_{12}和a_{13}完成，此时表示整个工程完成。

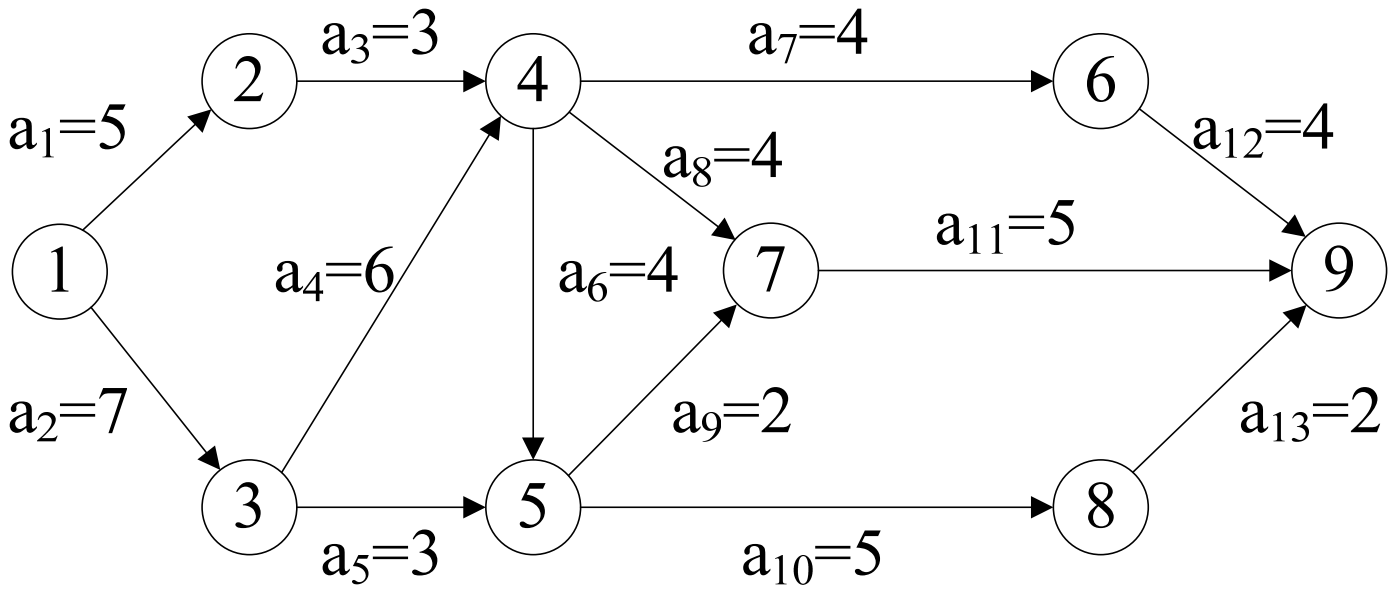


图8-31 一个假象工程的AOE-网络

因为边上的权表示完成该活动所需的时间，所以边a_1的权5，表示活动a_1需要5天（假设以天为单位）完成。类似地，活动a_4需要6天完成，等等。通常，这些时间仅仅是估计的。当工程开始后，活动a_1和a_2可以同时进行，而活动a_3要等到事件v_2发生后才能进行，a_4和a_5要在事件v_3发生后（即在活动a_2完成后）才能同时进行，当活动a_{11}, a_{12}和a_{13}完成后，整个工程就完成了。

定义几个与计算关键活动有关的量：

①事件V_i的最早可能开始时间ee(i)

是从源点V_1到顶点V_i的最长路径长度。

②事件V_i的最迟允许开始时间le[i]

是在保证汇点V_n在ee[n-1]时刻完成的前提下，事件V_i的允许的最迟开始时间。le[i]等于ee[n]减去顶点i到顶点n的最长路径长度。

③活动a_i的最早可能开始时间e[k]

设活动a_i在边$<V_j, V_k>$上，则e[i]是从源点V_1到顶点V_j的最长路径长度。因此，

e[i]=ee[j]

④活动a_i的最迟允许开始时间 l[i]

l[i]是在不会引起时间延误的前提下，该活动允许的最迟开始时间。

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

其中，dur(<j, k>)是完成a_i所需的时间。

⑤时间余量 l[i]-e[i]

表示活动a_i的最早可能开始时间和最迟允许开始时间的时间余量，l[i]==e[i]表示活动a_i是没有时间余量的关键活动。

- 为找出关键活动，需要求各个活动的e[i]与l[i]，以判别是否 l[i]==e[i]。

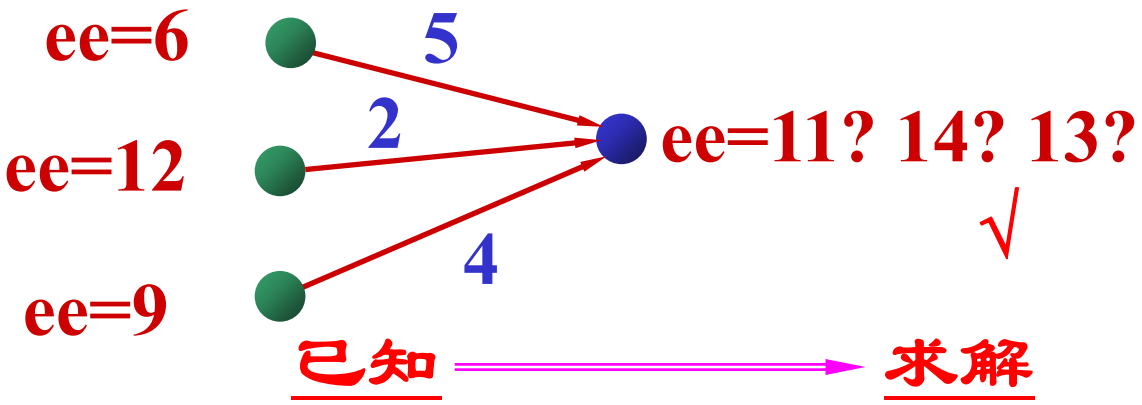
- 称e[i]=l[i]的活动a_i为关键活动。
  - 若a_i拖延时间，则整个工程也要拖延时间。
  - l[i]-e[i]为活动a_i的最大可利用时间（即时间余量），它就是在不增加完成工程所需的总时间的情况下，活动a_i可以拖延的时间。
  - l[i]-e[i]>0的活动a_i不是关键活动，即使提早完成a_i的时间小于等于l[i]-e[i]天，也不能加快整个工程的进度。
  - 若a_i提早完成的时间大于l[i]-e[i]天，则关键路径就可能发生变化。

- 为求得e[k]与l[k]，需要先求得从源点V_1到各个顶点V_i的ee[i]和le[i]。
- 求ee[k]的递推公式
  - 从ee[1]=0开始，向前递推

$$ee[k]=\max_{j}\{ee[j]+dur(<V_j,V_k>)\}$$

$<V_j, V_k>\in S2$，　j=1, 2, …, n-1

S2是所有指向V_k的有向边$<V_j, V_k>$的集合。

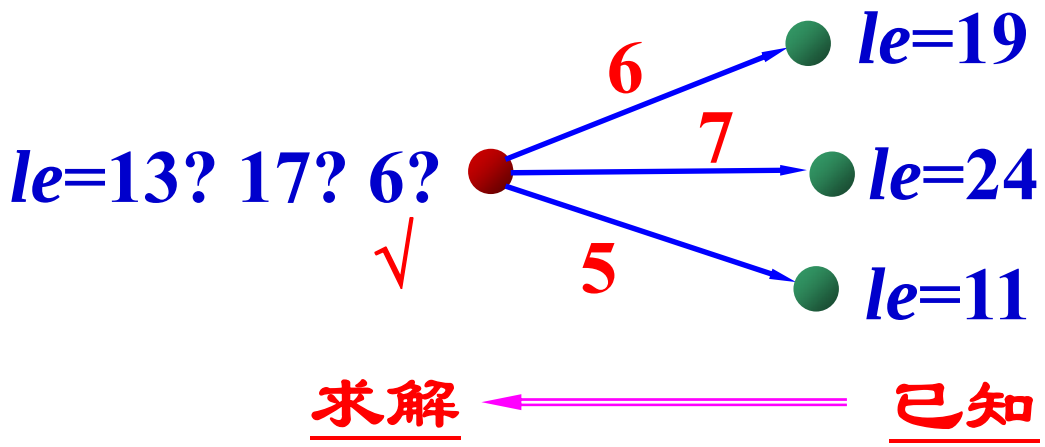


- 从le[n]=ee[n]开始，反向递推

$$le[k]=\min_{j}\{le[j]-dur(<V_k,V_j>)\}$$

$<V_k, V_j>\in S1$, k=n-2, n-3, …, 0

S1是所有源自V_k的有向边 $<V_k, V_j>$的集合。



- 这两个递推公式的计算必须分别在拓扑有序及逆拓扑有序的前提下进行。

- 设活动a_k (k=1, 2, …, e)在带权有向边<V_i, V_j>上，其持续时间用dur(<V_i , V_j>)表示，则有

  e[k]=ee[i]；

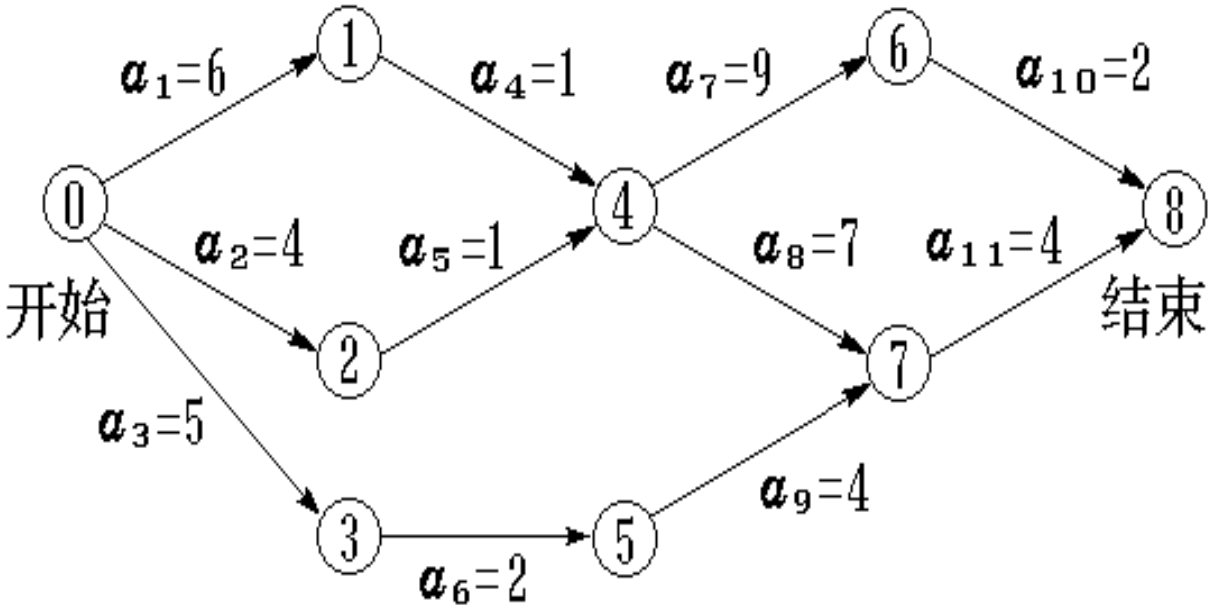
  l[k]=le[j]-dur(<V_i , V_j>)；

  k=1, 2, …, e。

  这样就得到计算关键路径的算法。

- 为简化算法，假定在求关键路径之前已经对各顶点实现了拓扑排序，并按拓扑有序的顺序对各顶点重新进行了编号。

事件	ee[i]	le[i]
ee_0	0	0
ee_1	6	6
ee_2	4	6
ee_3	5	8
ee_4	7	7
ee_5	7	10
ee_6	16	16
ee_7	14	14
ee_8	18	18

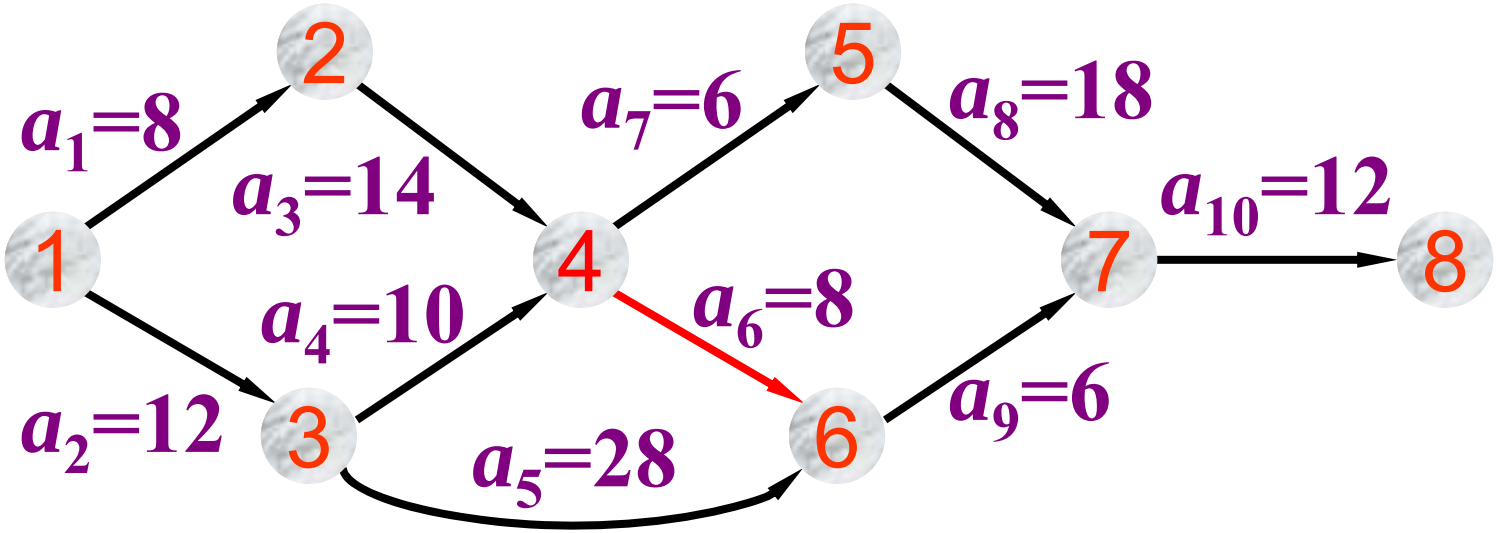


(a)

边	<0,1>	<0,2>	<0,3>	<1,4>	<2,4>	<3,5>	<4,6>	<4,7>	<5,7>	<6,8>	<7,8>
活动	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9	a_{10}	a_{11}
e	0	0	0	6	4	5	7	7	7	16	14
l	0	2	3	6	6	8	7	7	10	16	14
l - e	0	2	3	0	2	3	0	0	3	0	0
关键	是			是			是	是		是	是

	1	2	3	4	5	6	7	8
ee	0	8	12	22	28	40	46	58
le	0	8	12	22	28	40	46	58

	1	2	3	4	5	6	7	8	9	10
e	0	0	8	12	12	22	22	28	40	46
l	0	0	8	12	12	32	22	28	40	46



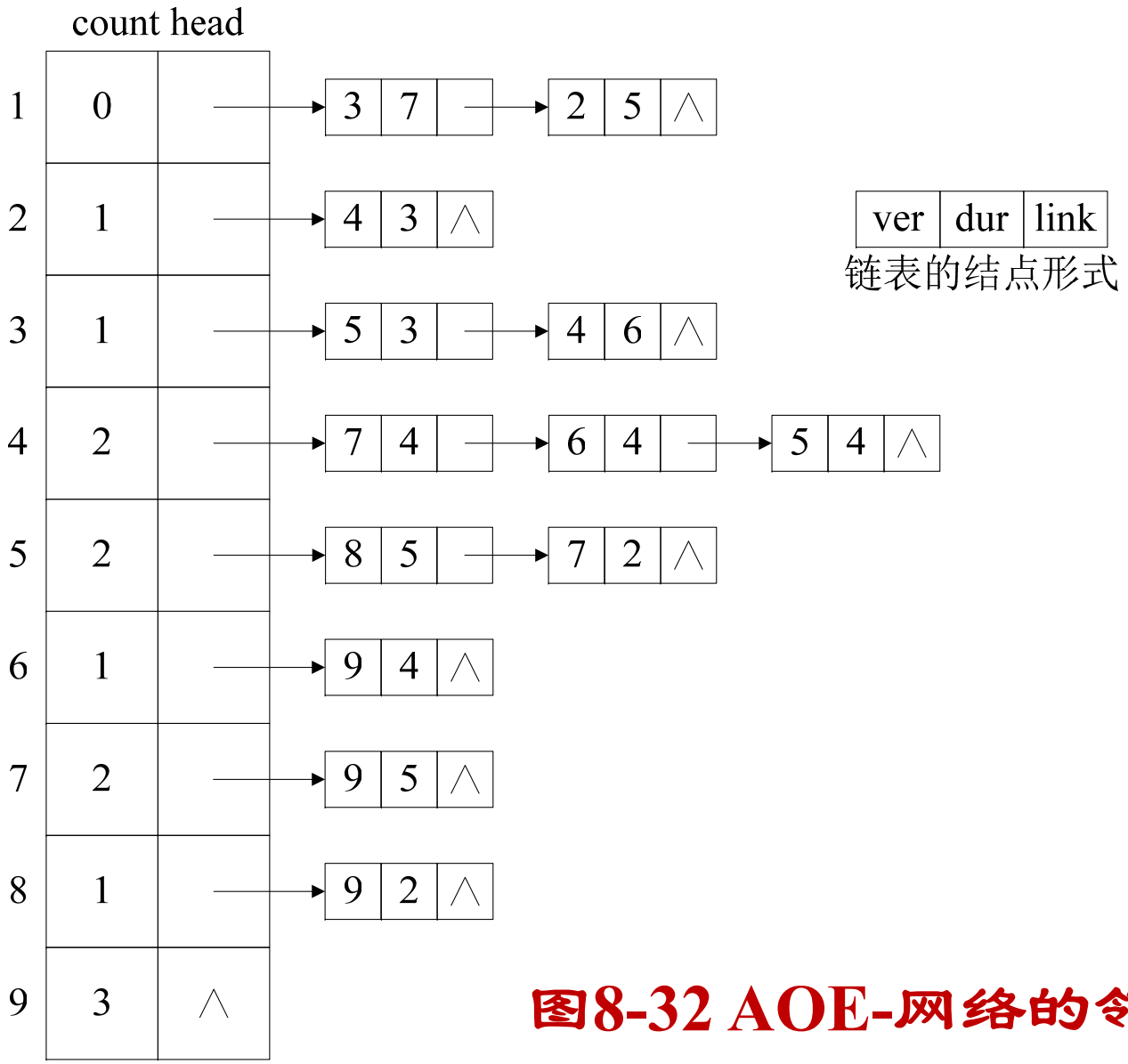


图8-32 AOE-网络的邻接表

i / ee[i]	1	2	3	4	5	6	7	8	9	栈
初态	0	0	0	0	0	0	0	0	0	1
输出顶点1	0	5	7	0	0	0	0	0	0	3, 2
输出顶点2	0	5	7	8	0	0	0	0	0	3
输出顶点3	0	5	7	13	10	0	0	0	0	4
输出顶点4	0	5	7	13	17	17	17	0	0	6, 5
输出顶点5	0	5	7	13	17	17	19	22	0	6, 8, 7
输出顶点7	0	5	7	13	17	17	17	22	24	6, 8
输出顶点8	0	5	7	13	17	17	17	22	24	6
输出顶点6	0	5	7	13	17	17	17	22	24	9
输出顶点9	0	5	7	13	17	17	17	22	24	

图8-33 求解ee的过程

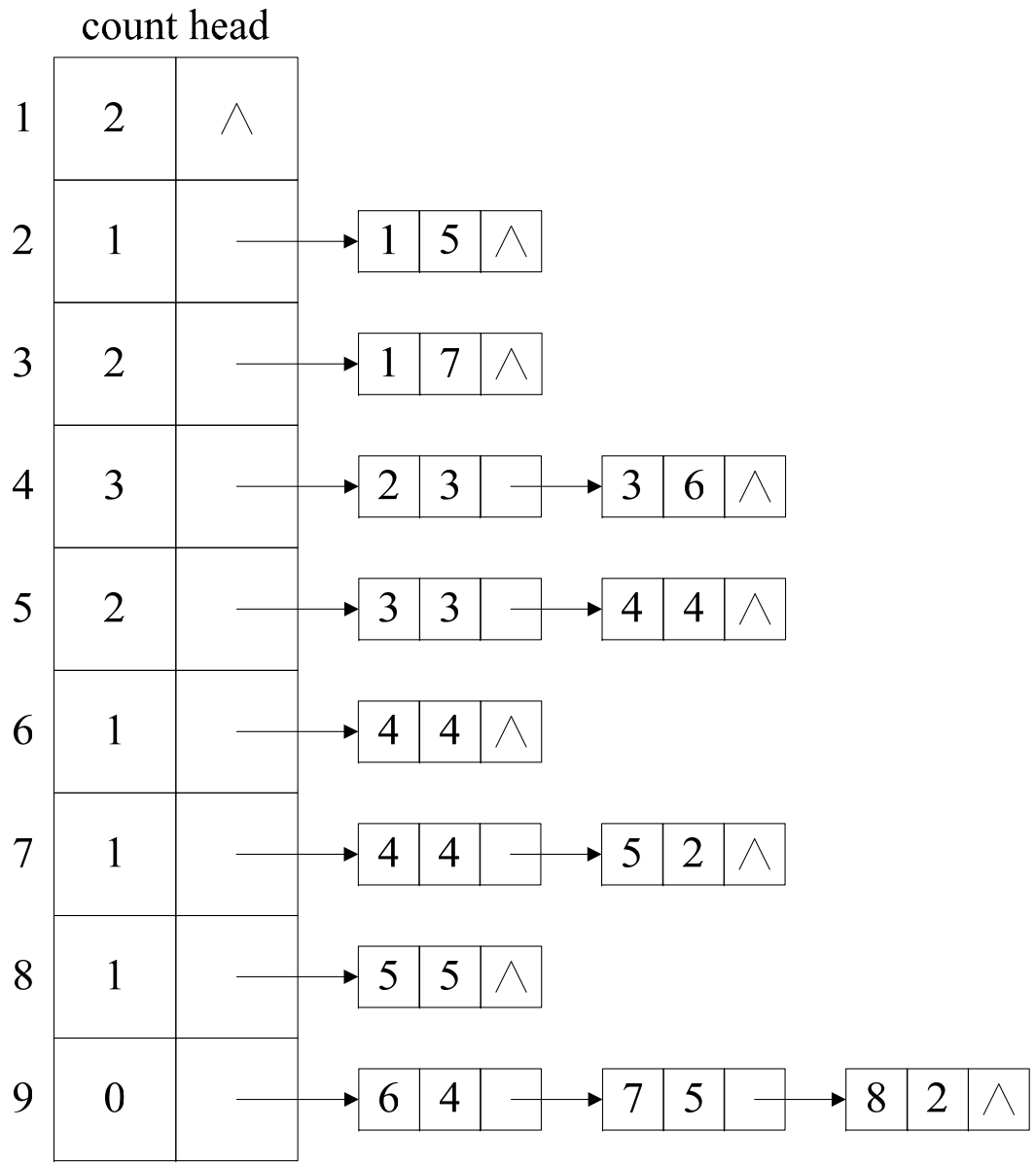


图8-34 图8-31的逆邻接表

e[1]=ee[1]=0	l[1]=le[2]-5=10-5=0	l[1]-e[1]=5
e[2]=ee[1]=0	l[2]=le[3]-7=7-7=0	l[2]-e[2]=0
e[3]=ee[2]=5	l[3]=le[4]-3=13-3=10	l[3]-e[3]=5
e[4]=ee[3]=7	l[4]=le[4]-6=13-6=7	l[4]-e[4]=0
e[5]=ee[3]=7	l[5]=le[5]-3=17-3=14	l[5]-e[5]=7
e[6]=ee[4]=13	l[6]=le[4]-4=17-4=13	l[6]-e[6]=0
e[7]=ee[4]=13	l[7]=le[6]-4=20-4=16	l[7]-e[7]=3
e[8]=ee[4]=13	l[8]=le[7]-4=19-4=15	l[8]-e[8]=2
e[9]=ee[5]=17	l[9]=le[7]-2=19-2=17	l[9]-e[9]=0
e[10]=ee[5]=17	l[10]=le[8]-5=22-5=17	l[10]-e[10]=0
e[11]=ee[7]=19	l[11]=le[9]-5=24-5=19	l[11]-e[11]=0
e[12]=ee[6]=17	l[12]=le[9]-4=24-4=20	l[12]-e[12]=3
e[13]=ee[8]=0	l[13]=le[9]-2=24-2=22	l[13]-e[13]=0

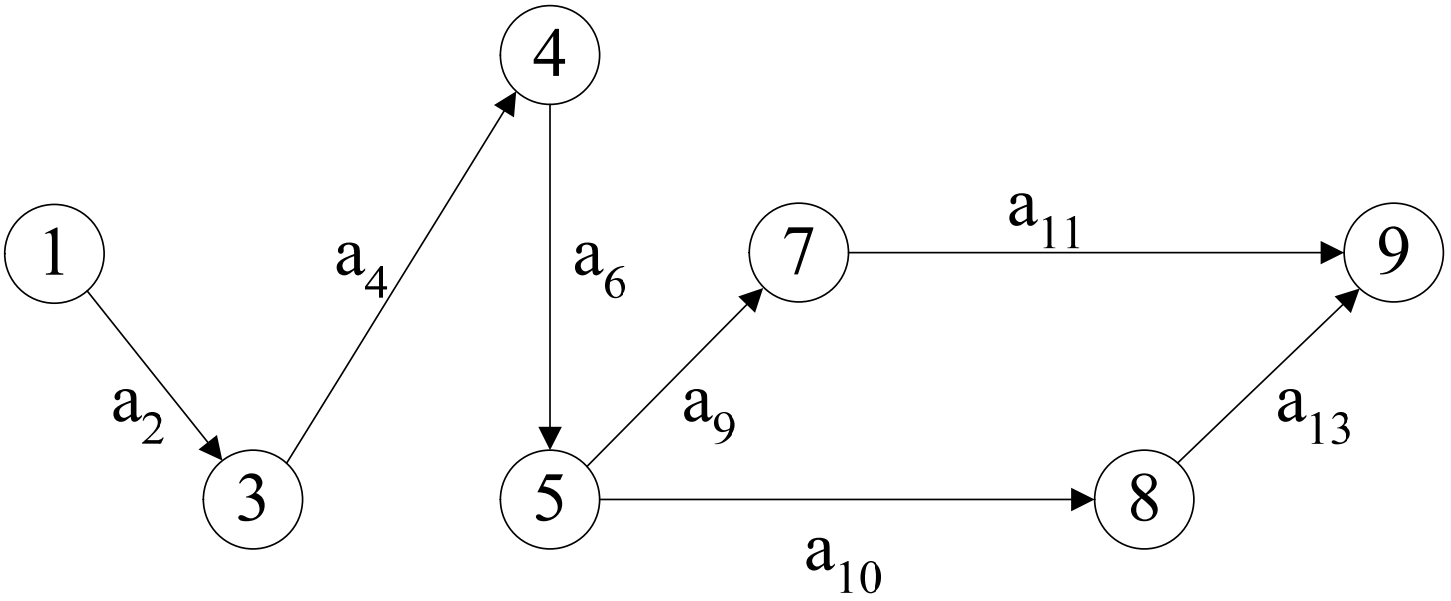


图8-35 两条关键路径

利用关键路径法求AOE网的各关键活动

```
void CriticalPath(Graph <Type> &G) {
//在此算法中需要在邻接表中单链表的结点内
//增加一个int型cost域，记录该边上的权值
    int i, j, k;  float e, l, w;
    int n=G.NumberOfVertices( );
    float *ee=new float [n];
    float *le=new float [n];
    for (i=0; i<n; i++)  ee[i]=0;
    for (i=0; i<n; i++) {  //顺向计算ee[ ]
        j=G.GetFirstNeighbor(i);
        while (j!=-1) {
```

```
        w=G.GetWeight(i, j);
        if (ee[i]+w>ee[j])  ee[j]=ee[i]+w;
        j=G.GetNextNeighbor(i, j);
    }
}
le[n-1]=ee[n-1];
for (j=n-2; j>0; j--) {  //逆向计算le[ ]
    k=G.GetFirstNrighbor(j);
    while (k!=-1) {
        w=G.GetWeight(j, k);
        if (le[k]-w<le[j])  le[j]=le[k]-w;
        k=G.GetNextNeighbor(j, k);
```

```
        }
    }
    for (i=0; i<n; i++) {  //求各活动的e、l
        j=G.GetFirstNeighbor(i);
        while (j!=-1) {
            e=ee[i];  l=le[j]-G.GetWeight(i, j);
            if (l==e)
                cout<<"<"<<i<<","<< j
                        <<">"<<"是关键活动"<<endl;
            j=G.GetNextNeighbor(i, j);
        }
    }
}
```

注意

- 所有顶点按拓扑有序的次序编号。
- 仅计算**ee[i]**和**le[i]**是不够的，还须计算**e[k]**和 **l[k]**。
- 不是任一关键活动加速一定能使整个工程提前。
- 想使整个工程提前，要考虑各个关键路径上所有关键活动。

随堂练习

例1：

(1) 如果含n个顶点的图形成一个环，则它有____棵生成树。

(2) 有10个顶点的无向图，边的总数最多为_____。

(3) G是一个非连通无向图，共有28条边，则该图至少有

_____个顶点。

例2：某乡有A, B, C, D四个村庄，图中边上的权值Wij即为从i村庄到j村庄间的距离。现在要在乡里建立中心俱乐部，其选址应使得离中心最远的村庄离俱乐部最近。

(1) 请写出各村庄之间的最短距离矩阵；

(2) 写出该中心俱乐部应设在哪个村庄，以及各村庄到中心俱乐部的路径和路径长度。

例3：试给出判定一个图是否存在回路的方法。。

例4：设计算法，求出无向连通图中距离顶点v0的最短路径长度（最短路径长度以边数为单位计算）为k的所有结点，要求尽可能地节省时间。

例1：

(1) 如果含n个顶点的图形成一个环，则它有 n 棵生成树。

(2) 有10个顶点的无向图，边的总数最多为 45 。

(3) G是一个非连通无向图，共有28条边，则该图至少有

9 个顶点。

图G是非连通无向图，至少有两个连通分量；一个连通分量最少的顶点数是由28条边组成的无向完全图，其顶点数n可由 $e \le n(n-1)/2$，解之得 $n \ge 8$；另外一个顶点自成一个连通分量，所以该图至少有9个顶点。

例2：某乡有A, B, C, D四个村庄，图中边上的权值Wij即为从i村庄到j村庄间的距离。现在要在乡里建立中心俱乐部，其选址应使得离中心最远的村庄离俱乐部最近。

(1) 请写出各村庄之间的最短距离矩阵；

(2) 写出该中心俱乐部应设在哪个村庄，以及各村庄到中心俱乐部的路径和路径长度。

弗洛伊德算法

例3：试给出判定一个图是否存在回路的方法。。

(1) 利用拓扑排序算法可以判定图G是否存在回路。即，在拓扑排序输出结束后所余下的顶点均有前驱，则说明只得到了部分顶点的拓扑有序序列，AOV网中存在着有向回路。

(2) 设G是n个顶点的无向连通图，若G的边数e>=n，则G中一定有回路存在。因此，只要计算出G的边数，就可判定图G中是否存在回路。

(3) 设G是n个顶点的无向连通图，若G的每个顶点的度大于或等于2，则图中一定存在回路。

(4) 利用深度优先遍历算法可以判定图G中是否存在回路。对无向图来说，若深度优先遍历过程中遇到了回边则必定存在环；对有向图来说，这条回边可能是指向深度优先森林中另一棵生成树上顶点的弧；但是，如果从有向图上的某个顶点v出发进行深度优先遍历，若在DFS(v)结束之前出现一条从顶点u到顶点v的回边，因u在生成树上是v的子孙，则有向图必定存在包含顶点v和顶点u的环。

例4：设计算法，求出无向连通图中距离顶点v0的最短路径长度（最短路径长度以边数为单位计算）为k的所有结点，要求尽可能地节省时间。

算法中必须用广度优先遍历的层次性特性来求解，也即要在以v0为起点调用BFS算法输出第k+1层上的所有顶点。因此，在访问顶点时需要知道层数，而每个顶点的层数是由其前驱决定的（起点除外）。所以，可以从第一个顶点开始，每访问到一个顶点就根据其前驱的层次计算该顶点的层次，并将层数值与顶点编号一起入队、出队。实际上可增加一个队列来保存顶点的层数值，并且将层数的相关操作与对应顶点的操作保持同步，即一起置空、出队和入队。

本章小结

- 知识点
  - 图的定义及存储表示
  - 图的遍历
  - 最小生成树
  - 最短路径
  - 活动网络

常见题型

- 基本概念
- 各种算法
  - 给定图，要求写出深度优先遍历的顺序，广度优先遍历的顺序，对应的邻接表表示。
    - 深度优先、广度优先算法的应用，例如求经过k条边的路径，经过最少边的路径等。
  - 给定图，求最小生成树，或者要求用Prim和Kruskal算法的生成过程。用表格表示Dijkstra或者Floyd算法执行过程中状态的改变。
    - 选址问题，例如给定几个城市之间的路线，要求选一个城市建立一个商店，与各城市的总距离最短，或者具有最小的最远距离。（可以逐点用Dijkstra算法，也可以用Floyd算法，然后求出每两个城市之间的最短路径后再对每个城市找出其最远的最短路径，从中选出最小的，就可以得到结果）
  - 给定图，求拓扑排序，求关键路径

- 课程习题
  - 笔做题——**8.11, 8.15, 8.17**
    （以作业形式提交）
  - 上机题——**8.14**
  - 思考题——剩余的其它习题

Any Suggestion or Question

联系方式：

- 张玥杰

Email: yjzhang@fudan.edu.cn