第八章 图

- 图的基本概念
- 图的存储表示
- 图的遍历与连通性
- 最小生成树
- 最短路径
- 活动网络
- 小结

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　$V = \{ x \mid x \in$ 某个数据对象$\}$

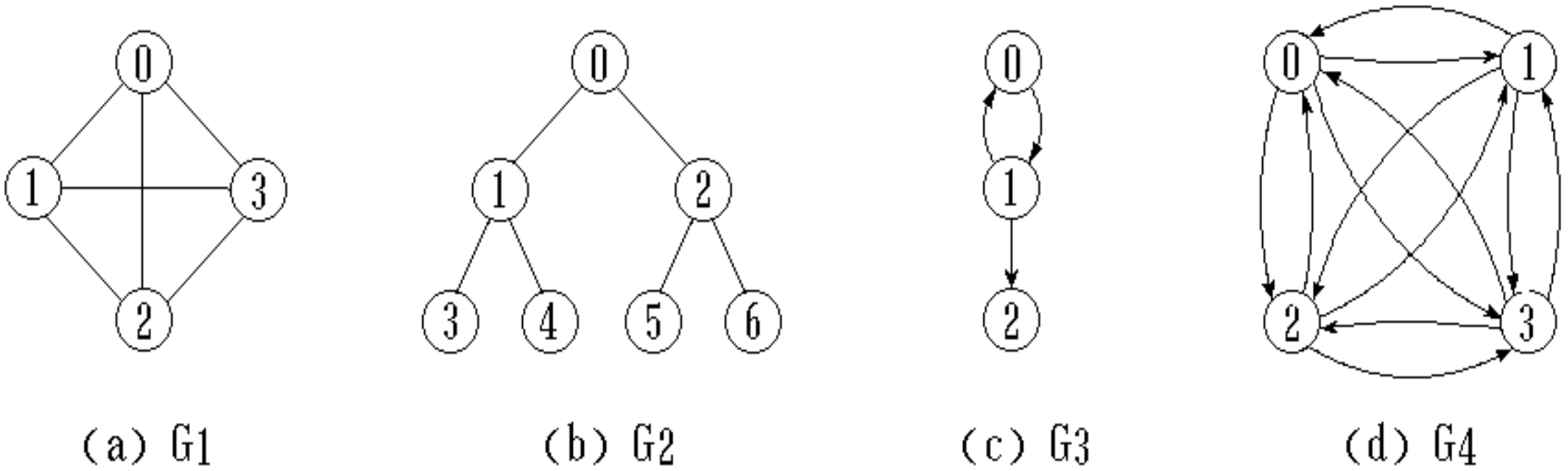
是顶点的有穷非空集合;

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

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

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。

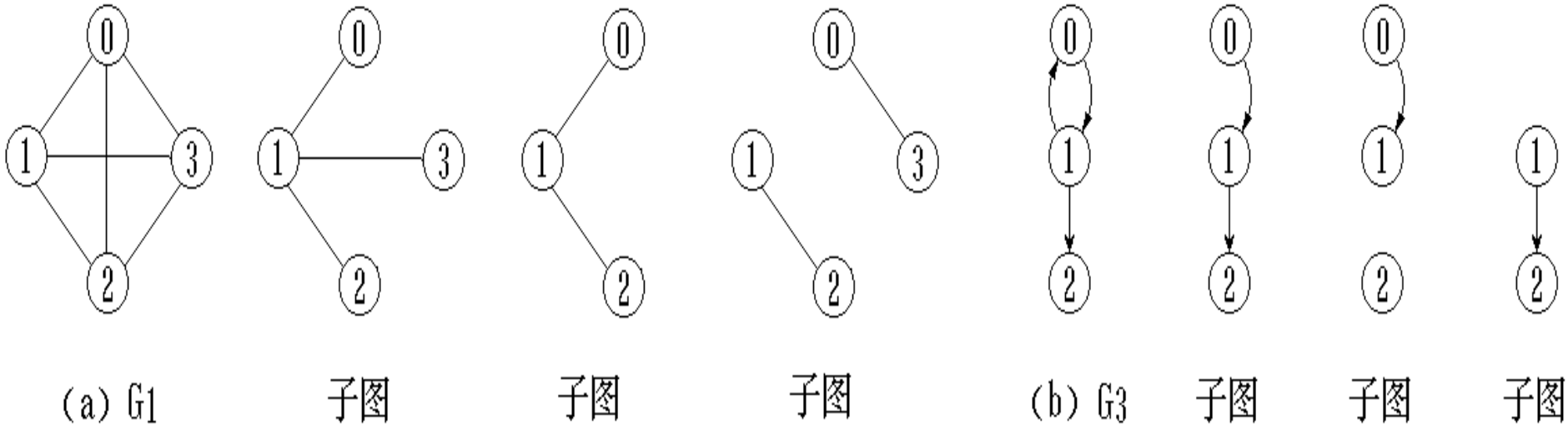
- 有向图与无向图　在有向图中，顶点对<x, y>是有序的。在无向图中，顶点对(x, y)是无序的。
- 完全图　若有 *n* 个顶点的无向图有 *n*(*n*-1)/2 条边，则此图为完全无向图。有 *n* 个顶点的有向图有 *n*(*n*-1) 条边，则此图为完全有向图。



(a) G1　(b) G2　(c) G3　(d) G4

- 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。

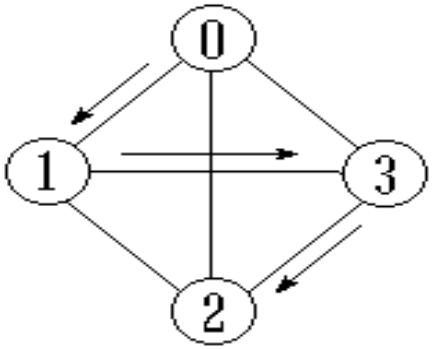
- **权** 某些图的边具有与它相关的数，称之为权。这种带权图叫做网络。
- **子图** 设有两个图 $G=(V, E)$ 和 $G'=(V', E')$。若 $V' \subseteq V$ 且 $E' \subseteq E$，则称 图G' 是 图G 的子图。



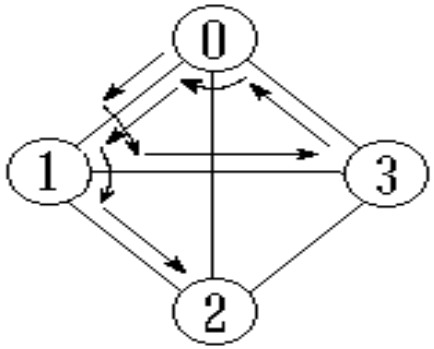
- **顶点的度** 一个顶点v的度是与它相关联的边的条数。记作$TD(v)$。在有向图中，顶点的度等于该顶点的入度与出度之和。

- 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 $v_{p1}, v_{p2}, \cdots, v_{pm}$，到达顶点 v_j。则称顶点序列 $(v_i\ v_{p1}\ v_{p2} \cdots v_{pm}\ v_j)$ 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。
- 路径长度
  - 非带权图的路径长度是指此路径上边的条数。
  - 带权图的路径长度是指路径上各边的权之和。

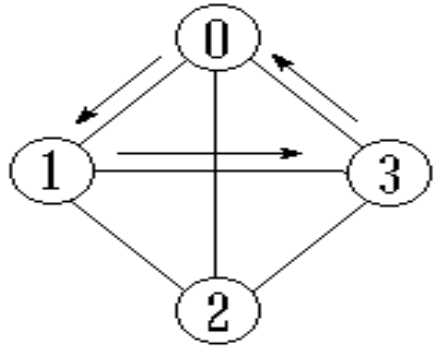
- **简单路径**　若路径上各顶点 $v_1,v_2,\ldots,v_m$ 均不互相重复，则称这样的路径为简单路径。
- **回路**　若路径上第一个顶点 v_1 与最后一个顶点 v_m 重合，则称这样的路径为回路或环。



（a）简单路径



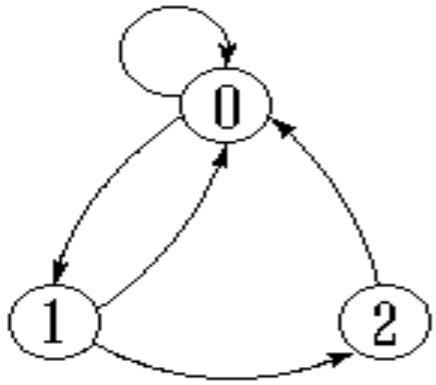
（b）非简单路径



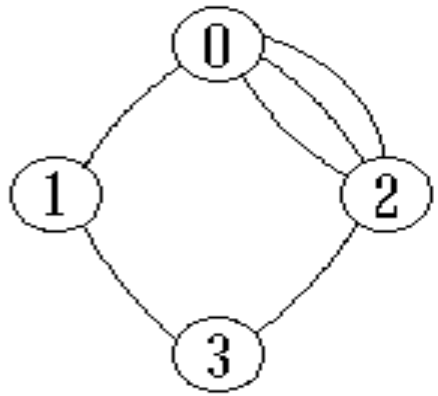
（c） 回路

- **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。

- 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。
- 生成树　一个连通图的生成树是它的极小连通子图，在n个顶点的情形下，有$n-1$条边。但有向图则可能得到它的由若干有向树组成的生成森林。
- 本章不予讨论的图



（a）带自身环的图



（b）多重图

图的抽象数据类型

```
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 );
    int IsEmpty ( );
    Type GetWeight ( int v1,  int v2 );
    int GetFirstNeighbor ( int v );
    int GetNextNeighbor ( int v1,  int v2 );
}
```

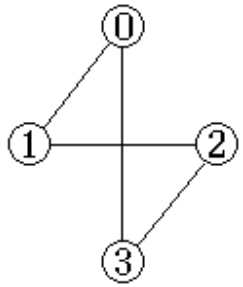
图的存储表示

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，则图的邻接矩阵是一个二维数组 **A.*edge*[*n*][*n*]**，定义：

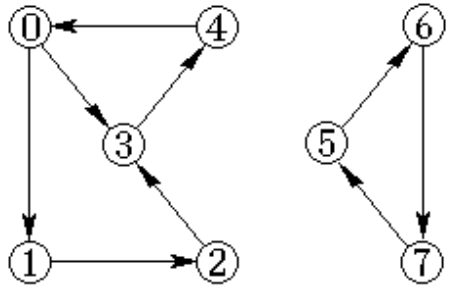
$$\mathbf{A}.Edge[i][j] = \begin{cases} \mathbf{1}, & \text{如果} < i, j > \in E \text{ 或者 } (i, j) \in E \\ \mathbf{0}, & \text{否则} \end{cases}$$

- 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。



G8

G3



G7

$$A.Edge = \begin{array}{c} \begin{matrix} ⓪ & ① & ② & ③ \end{matrix} \\ \begin{pmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{pmatrix} \begin{matrix} ⓪ \\ ① \\ ② \\ ③ \end{matrix} \end{array}$$

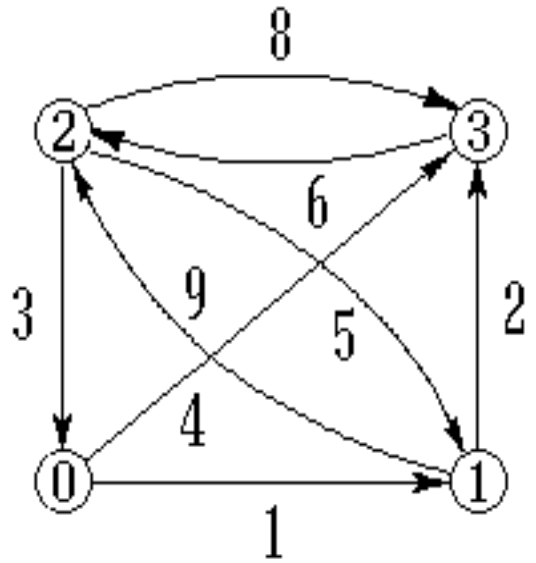
$$A.Edge = \begin{array}{c} \begin{matrix} ⓪ & ① & ② \end{matrix} \\ \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 0 & 0 \end{pmatrix} \begin{matrix} ⓪ \\ ① \\ ② \end{matrix} \end{array}$$

$$A.Edge = \begin{array}{c} \begin{matrix} ⓪ & ① & ② & ③ & ④ & ⑤ & ⑥ & ⑦ \end{matrix} \\ \left(\begin{array}{ccccc:ccc} 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \hdashline 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \end{array}\right) \begin{matrix} ⓪ \\ ① \\ ② \\ ③ \\ ④ \\ \\ ⑤ \\ ⑥ \\ ⑦ \end{matrix} \end{array}$$

- 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* 行 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

$$\mathbf{A}.Edge[i][j]=\begin{cases} W(i,j) & \text{如}i!=j\text{且}<i,j>\in E\text{或}(i,j)\in E \\ \infty & \text{否则,但是}i!=j \\ \mathbf{0} & \text{对角线}i==j \end{cases}$$



$$\mathrm{A}.Edge=\begin{array}{c} \begin{matrix} ⓪ & ① & ② & ③ \end{matrix} \\ \begin{pmatrix} 0 & 1 & \infty & 4 \\ \infty & 0 & 9 & 2 \\ 3 & 5 & 0 & 8 \\ \infty & \infty & 6 & 0 \end{pmatrix} \end{array} \begin{matrix} ⓪ \\ ① \\ ② \\ ③ \end{matrix}$$

用邻接矩阵表示的图的类的定义

```
const int MaxEdges = 50;
const int MaxVertices = 10;

template <class NameType, class DistType>
class Graph {
private:
    SeqList<NameType> VerticesList (MaxVertices);
    DistType Edge[MaxVertices][MaxVertices];
    int CurrentEdges;
    int FindVertex ( SeqList<NameType> & L;
       const NameType &vertex )
           { return L.Find (vertex); }
```

```
    int GetVertexPos ( Const NameType &vertex )
        { return FindVertex (VerticesList, vertex ); }
public:
    Graph ( int sz = MaxNumEdges );
    int GraphEmpty ( )
        const { return VerticesList.IsEmpty ( ); }
    int GraphFull( )
        const { return VerticesList.IsFull( ) ||
                       CurrentEdges ==MaxEdges; }
    int NumberOfVertices ( )
        { return VerticesList.last +1; }
    int NumberOfEdges ( ) { return CurrentEdges; }
```

```
    NameType GetValue ( int i )
        { return i >= 0 && i <= VerticesList.last
                    ? VerticesList.data[i] : NULL; }
    DistType GetWeight ( int v1,  int v2 );
    int GetFirstNeighbor ( int v );
    int GetNextNeighbor ( int v1,  int v2 );
    void InsertVertex ( NameType & vertex );
    void InsertEdge
            ( int v1,  int v2, DistType weight );
    void RemoveVertex ( int v );
    void RemoveEdge ( int v1,  int v2 );
}
```

邻接矩阵实现的部分图操作

```
template <class NameType, class DistType>
Graph<NameType, DistType> :: 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;
    CurrentEdges = 0;
}

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
```

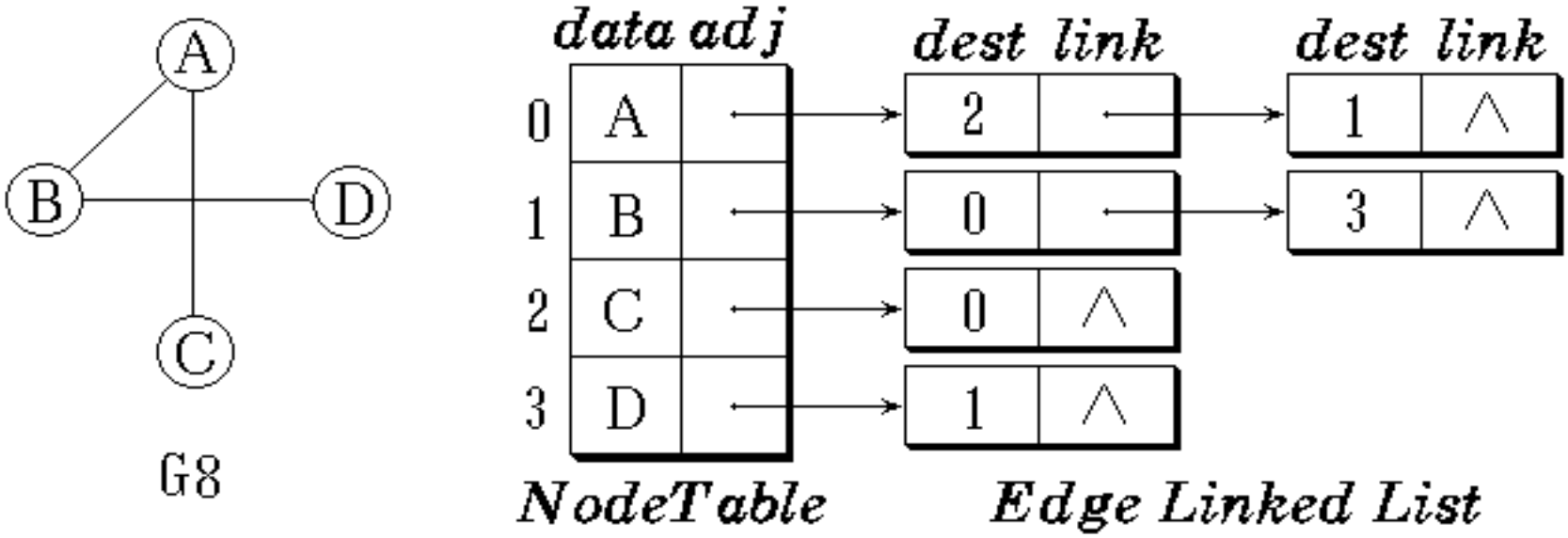
```
    if ( v1 != -1 && v2 != -1 )
       return Edge[v1][v2];
    else return 0;
}

template <class NameType, class DistType>
int Graph<NameType, DistType>::
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 col = 0; col < VerticesList.last; col++ )
          if ( Edge[row][col] > 0 &&
               Edge[row][col] < max )
       return col;
    }
```

```
    return −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
    int col;
    if ( v1 != −1 && v2 != −1 ) {
       for ( col = v2+1; col < VerticesList.last; col++ )
           if ( Edge[v1][col] > 0 &&
                   Edge[v1][col] < max ) return col;
    }
    return −1;
}
```

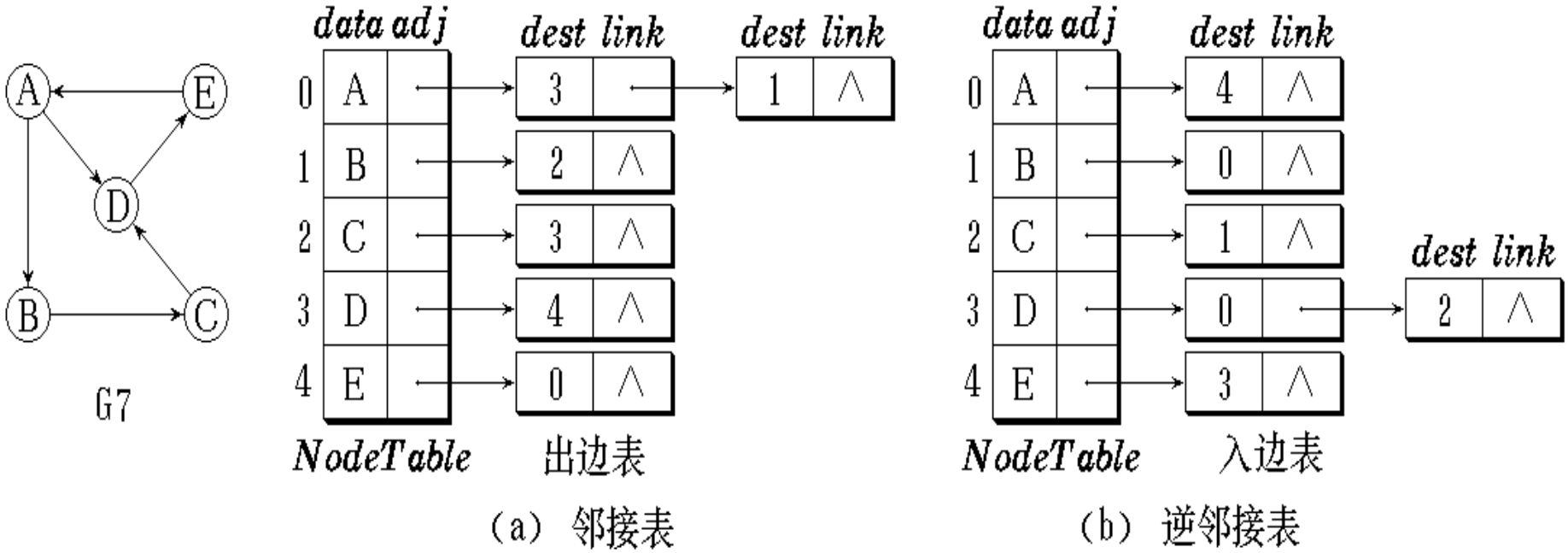
邻接表 (Adjacency List)

- 无向图的邻接表



把同一个顶点发出的边链接在同一个边链表中，链表的每一个结点代表一条边，叫做边结点，结点中保存有与该边相关联的另一顶点的顶点下标 ***dest*** 和指向同一链表中下一个边结点的指针 ***link***。

- 有向图的邻接表和逆邻接表



- 在有向图的邻接表中，第 *i* 个边链表链接的边都是顶点 *i* 发出的边。也叫做出边表。
- 在有向图的逆邻接表中，第 *i* 个边链表链接的边都是进入顶点 *i* 的边。也叫做入边表。

- 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。

邻接表表示的图的类定义

```
const int DefaultSize = 10;
template <class DistType> class Graph;

template <class DistType> struct Edge {
friend class Graph<NameType, DistType>;
    int dest;
    DistType cost;
    Edge<DistType> *link;
    Edge ( ) { }
    Edge ( int D, DistType C ) :
        dest (D), cost (C), link (NULL) { }
    int operator != ( Edge<DistType>& E )
        const { return dest != E.dest; }
}
```

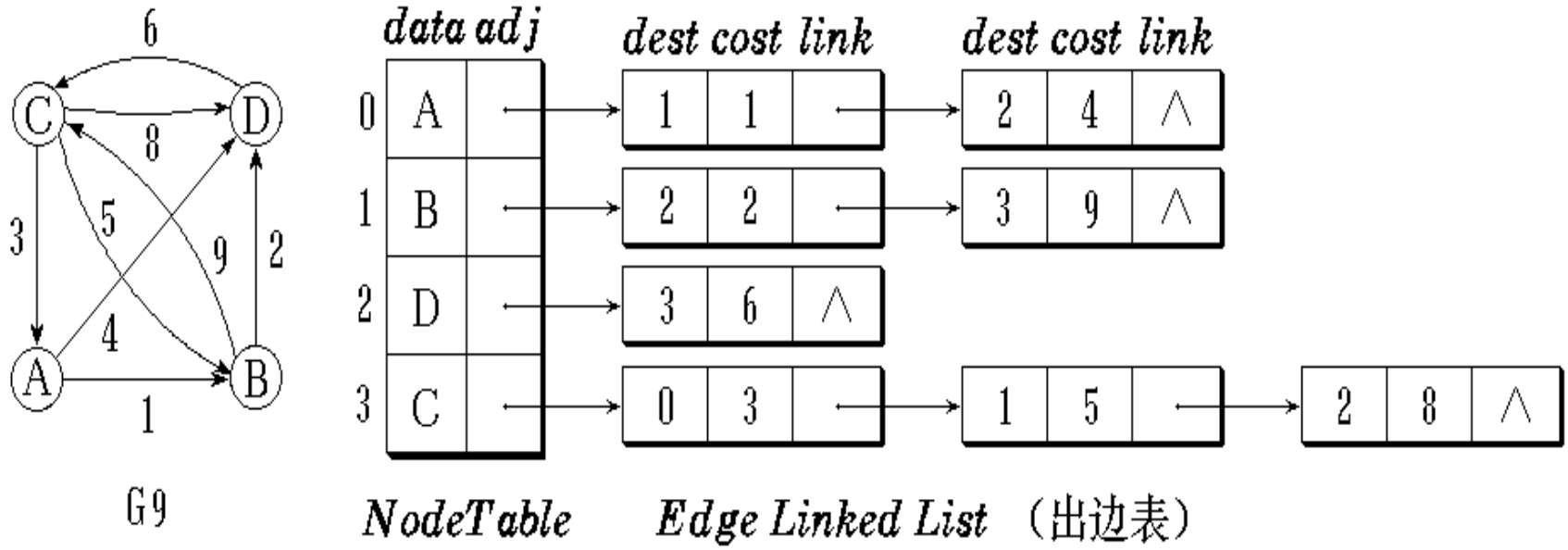
```
template <class NameType, class DistType>
struct Vertex {
friend class Graph<NameType, DistType>;
    NemeType data;
    Edge<DistType> *adj;
}

template <class NameType, class DistType>
class Graph {
private:
    Vertex<NameType, DistType> *NodeTable;
    int NumVertices;
    int MaxVertices;
```

```
    int NumEdges;
    int GetVertexPos ( NameType & vertex );
public:
    Graph ( int sz );
    ~Graph ( );
    int GraphEmpty ( )
        const { return NumVertices == 0; }
    int GraphFull ( ) const
        { return NumVertices == MaxVertices; }
    NameType GetValue ( int i )
         { return i >= 0 && i < NumVertices ?
                          NodeTable[i].data : NULL; }
    int NumberOfVertices ( ) { return NumVertices; }
```

```
    int NumberOfEdges ( ) { return NumEdges; }
    void InsertVertex ( NameType & vertex );
    void RemoveVertex ( int v );
    void InsertEdge ( int v1,  int v2,
                                  DistType weight );
    void RemoveEdge ( int v1,  int v2 );
    DistType GetWeight ( int v1,  int v2 );
    int GetFirstNeighbor ( int v );
    int GetNextNeighbor ( int v1,  int v2 );
}
```

网络 (带权图) 的邻接表



邻接表的构造函数和析构函数

```
template <class NameType, class DistType>
Graph<NameType, DistType> ::
Graph ( int sz = DefaultSize ) :
    NumVertices (0), MaxVertices (sz), NumEdges (0){
```

```
int n, e, k, j;  NameType name, tail, head;
DistType weight;
NodeTable =                                  //创建顶点表
    new Vertex<Nametype>[MaxVertices];
cin >> n;                                    //输入顶点个数
for ( int i = 0; i < n; i++)     //输入各顶点信息
  { cin >> name;  InsertVertex ( name ); }
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 );
}
```

```
}

template <class NameType, class DistType>
Graph<NameType, DistType> ::
~Graph ( ) {
    for ( int i = 0; i < NumVertices; i++ ) {
        Edge<DistType> *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 NameType, class DistType>
int Graph<NameType, DistType> ::
GetVertexPos ( const NameType & vertex ) {
//根据顶点名vertex查找该顶点在邻接表中的位置
    for ( int i =0; i < NumVertices; i++ )
        if ( NodeTable[i].data == vertex ) return i;
    return -1;
}

template <Class NameType, class DistType>
int Graph<NameType, DistType> ::
GetFirstNeighbor ( int v ) {
```

```
//查找顶点 v 第一个邻接顶点在邻接表中的位置
    if ( v != -1 ) {                                    //若顶点存在
        Edge<DistType> *p = NodeTable[v].adj;
        if ( p != NULL ) return p→dest;
    }
    return -1;                                          //若顶点不存在
}

template <Class NameType, class DistTypeType>
int Graph<NameType, DistType> ::
GetNextNeighbor ( int v1,  int v2 ) {
//查找顶点 v1 在邻接顶点 v2 后下一个邻接顶点
    if ( v1 != -1 ) {
        Edge<DistType> *p = NodeTable[v1].adj;
```

```
        while ( p != NULL ) {
            if ( p→dest == v2 && p→link != NULL )
                return p→link→dest;
            //返回下一个邻接顶点在邻接表中的位置
            else p = p→link;
        }
    }
    return -1;  //没有查到下一个邻接顶点返回-1
}
```

```
template <Class NameType, class DistType>
DistType Graph<NameType, DistType> ::
GetWeight ( int v1,  int v2) {
//取两端点为 v1 和 v2 的边上的权值
    if ( v1 != −1 && v2 != −1 ) {
        Edge<DistType> *p = NodeTable[v1].adj;
        while ( p != NULL ) {
            if ( p→dest == v2 ) return p→cost;
            else p = p→link;
        }
    }
    return 0;
}
```

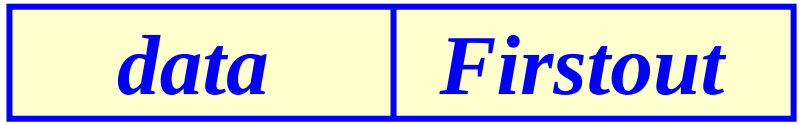
邻接多重表 (Adjacency Multilist)

- 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了方便。
- 无向图的情形
  - ◆边结点的结构

mark	*vertex*1	*vertex*2	*path*1	*path*2

其中，*mark* 是记录是否处理过的标记；*vertex*1和*vertex*2是依附于该边的两顶点位置。*path*1域是链接指针，指向下一条依附于顶点*vertex*1的边；*path*2也是链接指针，指向下一条依附于顶点*vertex*2的边。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

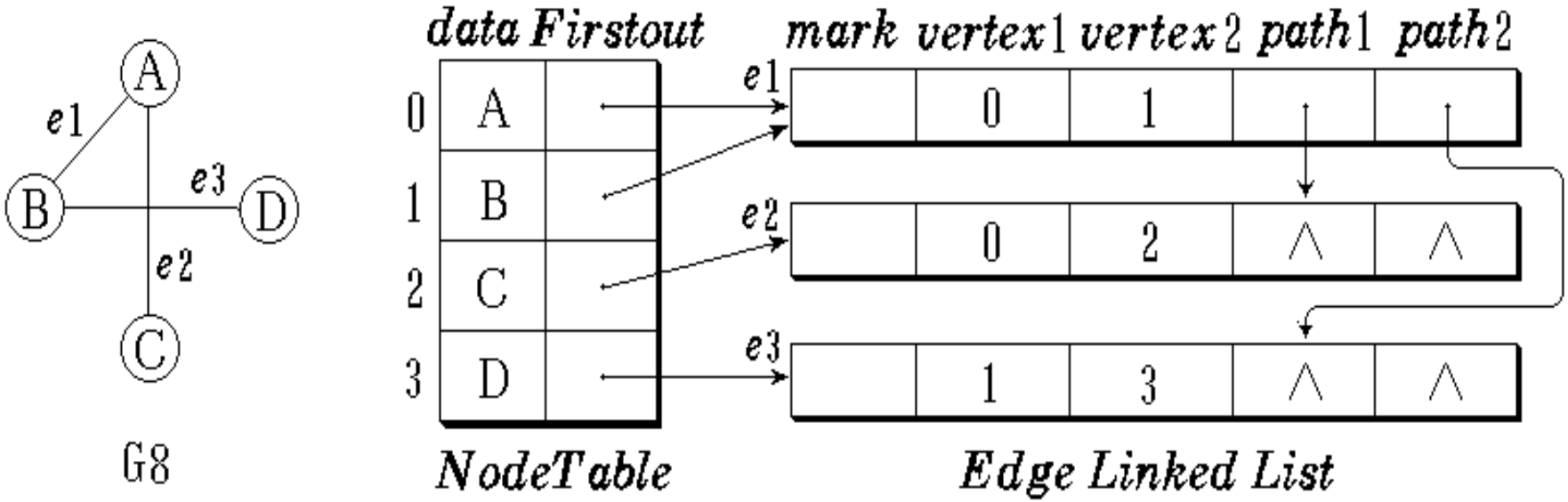


存储顶点信息的结点表以顺序表方式组织，每一个顶点结点有两个数据成员：其中，*data* 存放与该顶点相关的信息，*Firstout* 是指示第一条依附于该顶点的边的指针。

在邻接多重表中，所有依附于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。

◆邻接多重表的结构



- 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)可把这两个表结合起来表示。

  - 边结点的结构

mark	vertex1	vertex2	path1	path2

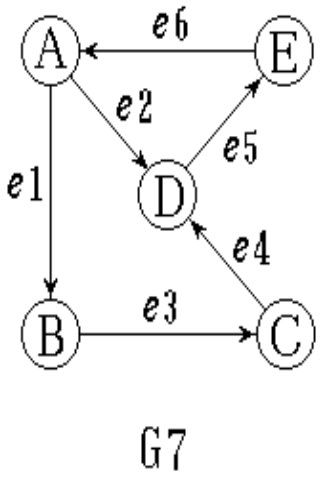
其中，*mark*是处理标记；*vertex*1和*vertex*2指明该有向边始顶点和终顶点的位置。*path*1是指向始顶点与该边相同的下一条边的指针；*path*2是指向终顶点与该边相同的下一条边的指针。需要时还可有权值域*cost*。

- 顶点结点的结构

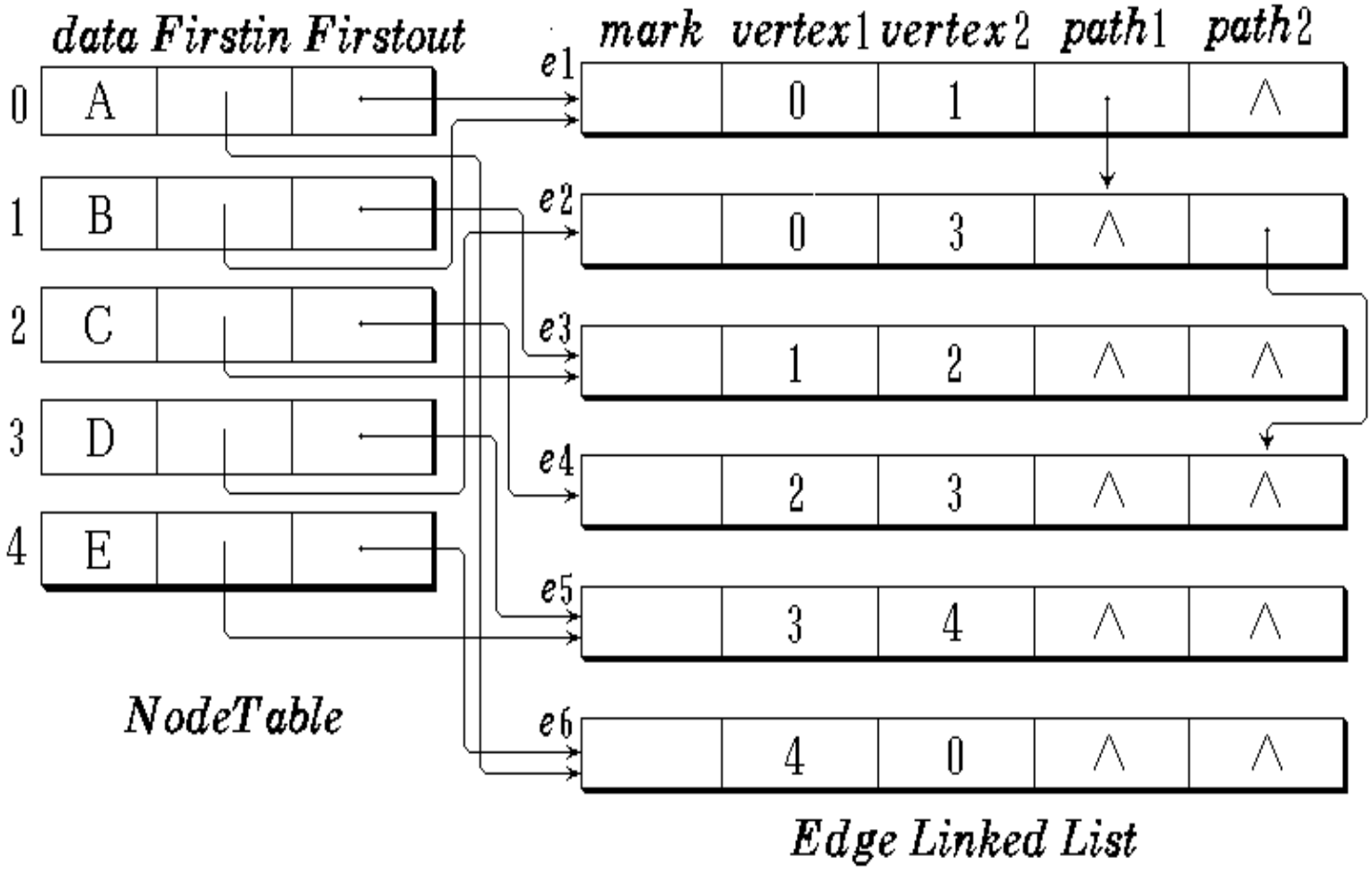
data	*Firstin*	*Firstout*

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*Firstin*指示以该顶点为始顶点的出边表的第一条边，*Firstout*指示以该顶点为终顶点的入边表的第一条边。

◆邻接多重表的结构


A
E
D
B
C
e1
e2
e3
e4
e5
e6

G7

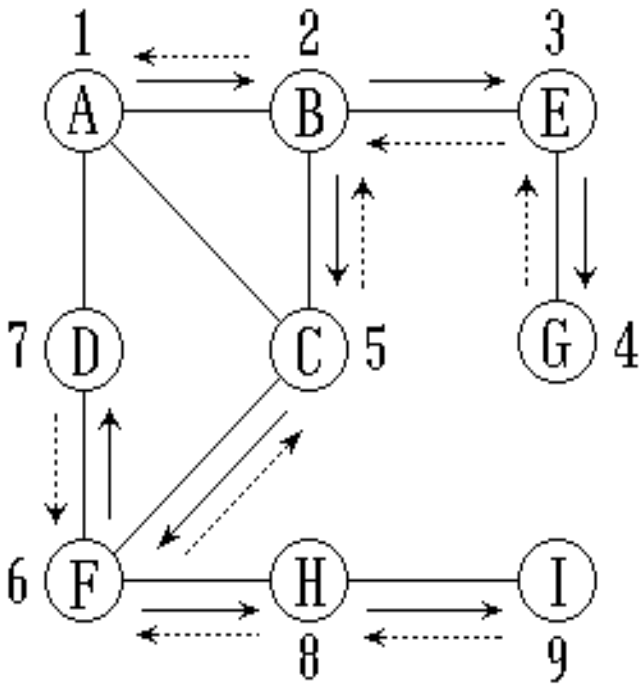

data Firstin Firstout
0 A
1 B
2 C
3 D
4 E
NodeTable
mark vertex1 vertex2 path1 path2
e1 0 1 ∧
e2 0 3 ∧
e3 1 2 ∧ ∧
e4 2 3 ∧ ∧
e5 3 4 ∧ ∧
e6 4 0 ∧ ∧
Edge Linked List

图的遍历与连通性

- 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。
- 为了避免重复访问，可设置一个标志顶点是否被访问过的辅助数组 ***visited* [ ]**，它的初始状态为 **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。

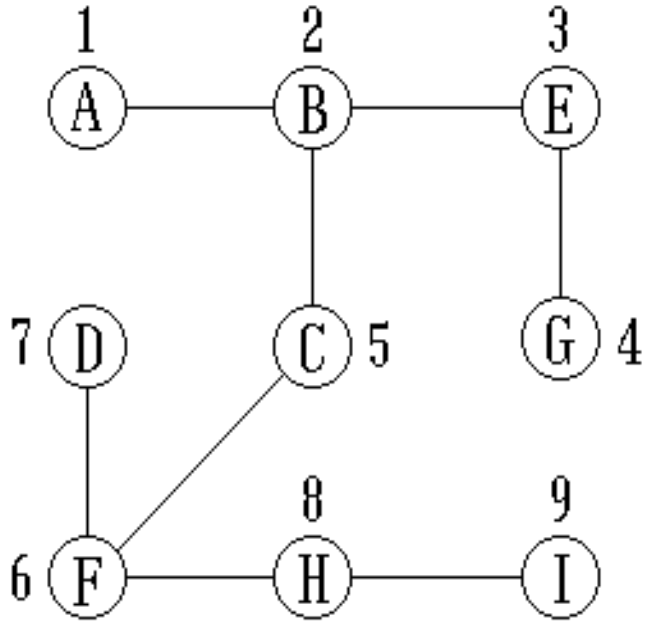
深度优先搜索*DFS* (Depth First Search)

- 深度优先搜索的示例



搜索

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。

图的深度优先搜索算法

```
template<class NameType, class DistType>
void Graph <NameType, DistType> :: DFS ( ) {
    int * visited = new int [NumVertices];
    for ( int i = 0; i < NumVertices; i++ )
        visited [i] = 0;  //访问标记数组 visited 初始化
    DFS (0, visited );
    delete [ ] visited;                    //释放 visited
}

template<class NameType, class DistType>
void Graph<NameType, DistType> ::
DFS ( const int v, int visited [ ] ) {
```

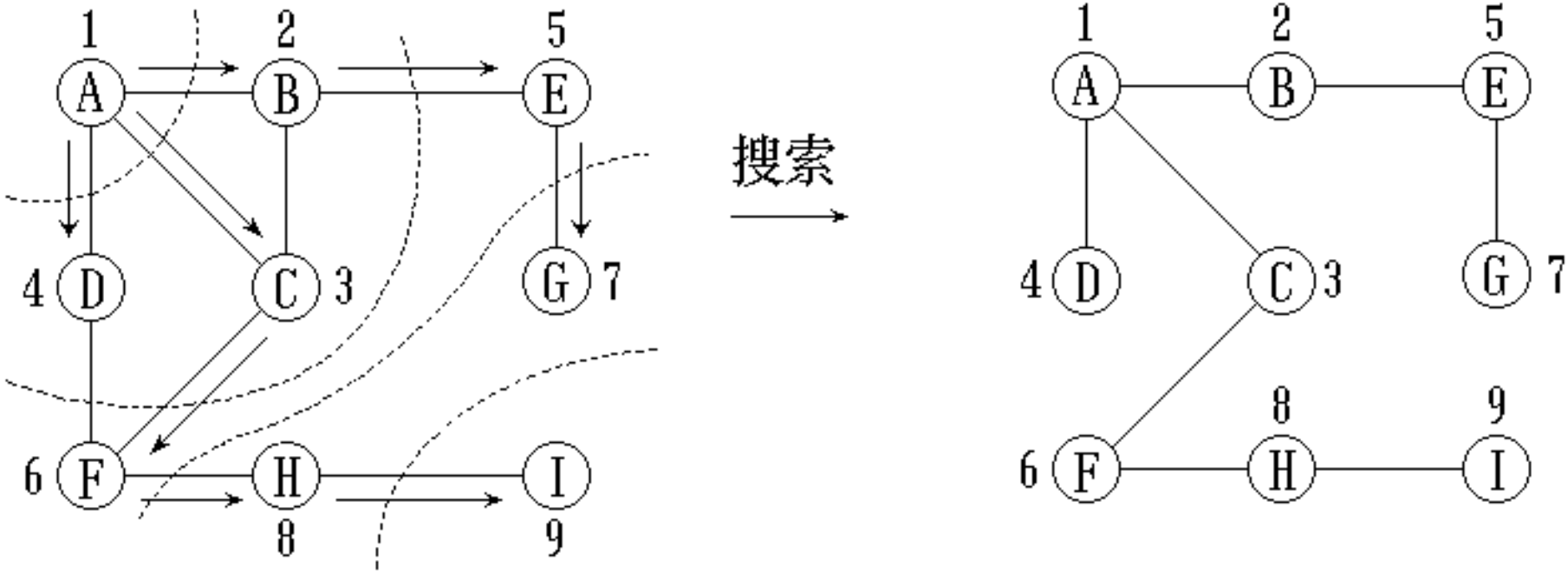
```
    cout << GetValue (v) << ' '; //访问顶点 v
    visited[v] = 1;                    //顶点 v 作访问标记
    int w = GetFirstNeighbor (v);
      //取 v 的第一个邻接顶点 w
    while ( w != -1 ) {                //若邻接顶点 w 存在
        if ( !visited[w] ) DFS ( w, visited );
        //若顶点 w 未访问过, 递归访问顶点 w
        w = GetNextNeighbor ( v, w );
        //取顶点 v 的排在 w 后面的下一个邻接顶点
    }
}
```

算法分析

- 图中有 n 个顶点，e 条边。
- 如果用邻接表表示图，沿 *Firstout* → *link* 链可以找到某个顶点 v 的所有邻接顶点 w。由于总共有 $2e$ 个边结点，所以扫描边的时间为$O(e)$。而且对所有顶点递归访问1次，所以遍历图的时间复杂性为$O(n+e)$。
- 如果用邻接矩阵表示图，则查找每一个顶点的所有的边，所需时间为$O(n)$，则遍历图中所有的顶点所需的时间为$O(n^2)$。

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

- 使用广度优先搜索在访问了起始顶点 v 之后，由 v 出发，依次访问 v 的各个未曾被访问过的邻接顶点 $w_1, w_2, \ldots, w_t$，然后再顺序访问 $w_1, w_2, \ldots, w_t$ 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，其算法也不是递归的。

- 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。

图的广度优先搜索算法

```
template<class NameType, class DistType>
void Graph <NameType, DistType> ::
BFS ( int v ) {
    int * visited = new int[NumVertices];
    for ( int i = 0; i < NumVertices; i++ )
      visited[i] = 0;            //visited 初始化
```

```
cout << GetValue (v) << ' ';   visited[v] = 1;
Queue<int> q;    q.EnQueue (v);  //访问 v, 进队列
while ( !q.IsEmpty ( ) ) {                  //队空搜索结束
    v = q.DeQueue ( );                      //不空, 出队列
    int w = GetFirstNeighbor (v);
     //取顶点 v 的第一个邻接顶点 w
    while ( w != −1 ) {     //若邻接顶点 w 存在
        if ( !visited[w] ) {   //若该邻接顶点未访问过
            cout << GetValue (w) << ‘ ’;          //访问
            visited[w] = 1;   q.EnQueue (w);    //进队
        }
        w = GetNextNeighbor (v, w);
        //取顶点 v 的排在 w 后面的下一邻接顶点
```

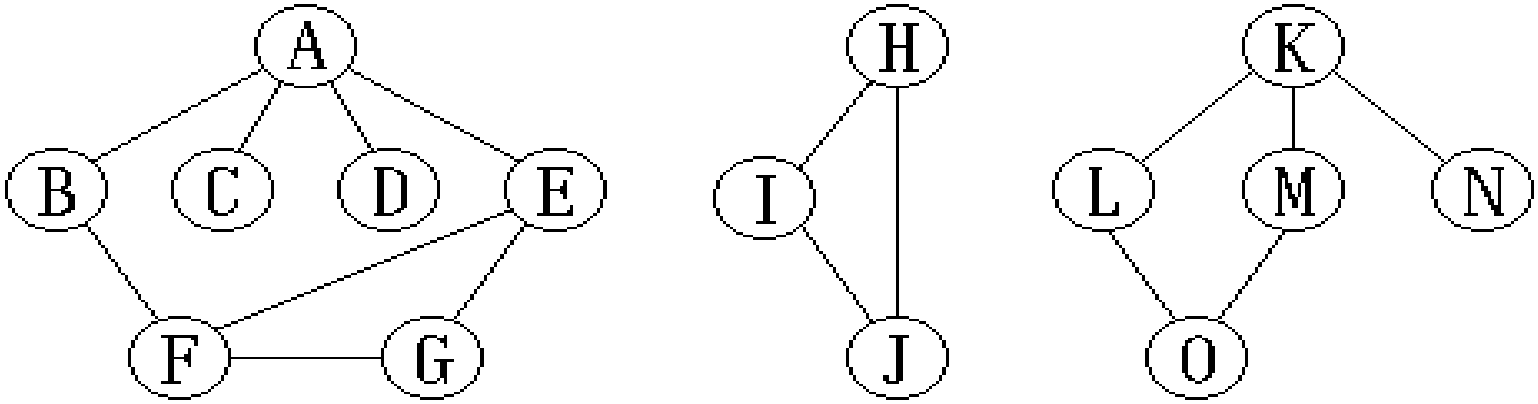
```
    }  //重复检测 v 的所有邻接顶点
  }  //外层循环，判队列空否
  delete [ ] visited;
}
```

算法分析

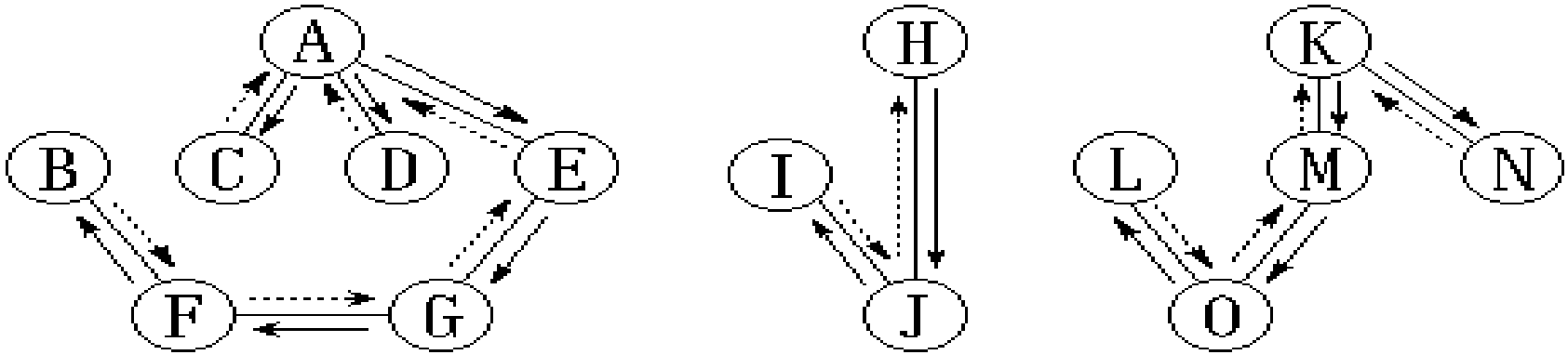
- 如果使用邻接表表示图，则循环的总时间代价为 $\mathbf{d_0 + d_1 + \ldots + d_{n-1} = O(e)}$，其中的 $\mathbf{d_i}$ 是顶点 i 的度。
- 如果使用邻接矩阵，则对于每一个被访问过的顶点，循环要检测矩阵中的 n 个元素，总的时间代价为 $\mathbf{O(n^2)}$。

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。



（a）非连通无向图



（c）非连通图的连通分量

- 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。

确定连通分量的算法

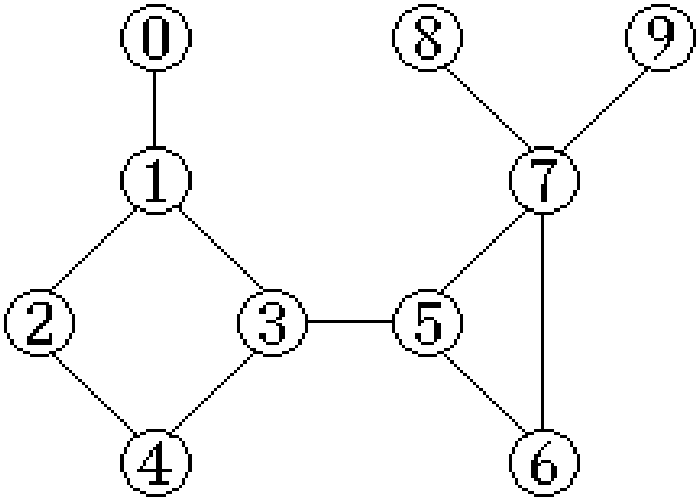
```
template<class NameType, class DistType>
void Graph<NameType, DistType> ::
Components ( ) {
    int *visited = new int[NumVertices];
    for ( int i = 0; i < NumVertices; i++ )
        visited[i] = 0;                          //visited 初始化
    for ( i = 0; i < NumVertices; i++ )
        if ( !visited[i] ) {      //检测所有顶点是否访问过
            DFS ( i, visited ); //从未访问的顶点出发访问
            OutputNewComponent ( );
              //输出一个连通分量
        }
```

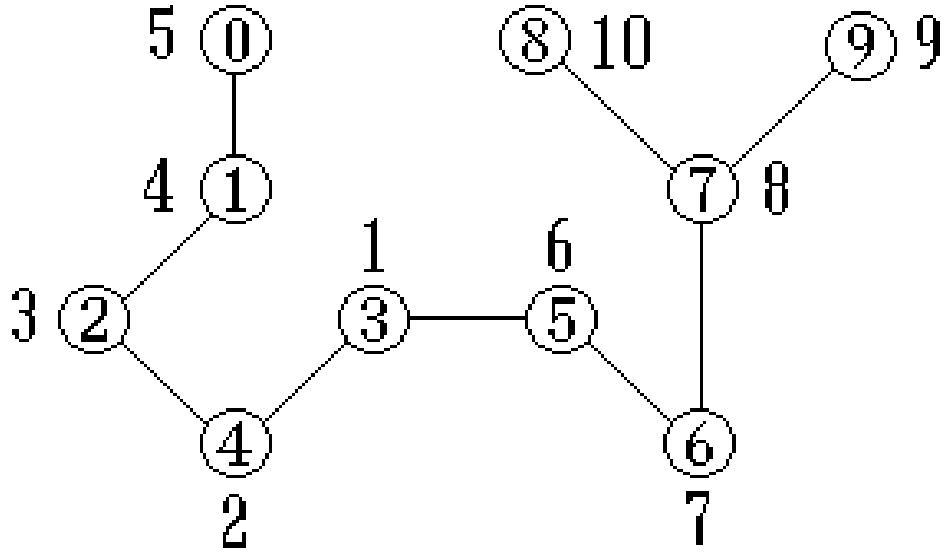
```
    delete [ ] visited;                    //释放visited
}
```

重连通分量 (Biconnected Component)

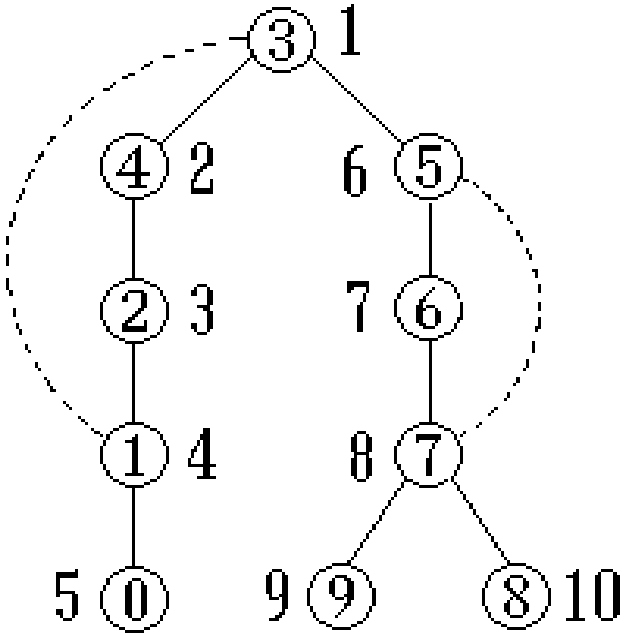
- 在无向连通图G中，当且仅当删去G中的顶点*v*及所有依附于*v*的所有边后，可将图分割成两个或两个以上的连通分量，则称顶点*v*为关节点。
- 没有关节点的连通图叫做重连通图。
- 在重连通图上，任何一对顶点之间至少存在有两条路径，在删去某个顶点及与该顶点相关联的边时，也不破坏图的连通性。

- 一个连通图G如果不是重连通图，那么它可以包括几个重连通分量。
- 在一个无向连通图G中，重连通分量可以利用深度优先生成树找到。
- ***dfn*** 顶点的深度优先数，标明进行深度优先搜索时各顶点访问的次序。
- 如果在深度优先生成树中，顶点 ***u*** 是顶点 ***v*** 的祖先，则有***dfn*[*u*] < *dfn*[*v*]**。
- 深度优先生成树的根是关节点的充要条件是它至少有两个子女。
- 其它顶点 ***u*** 是关节点的充要条件是它至少有一个子女 ***w***，从 ***w*** 出发，不能通过 ***w***、***w*** 的子孙及一条回边所组成的路径到达 ***u*** 的祖先。

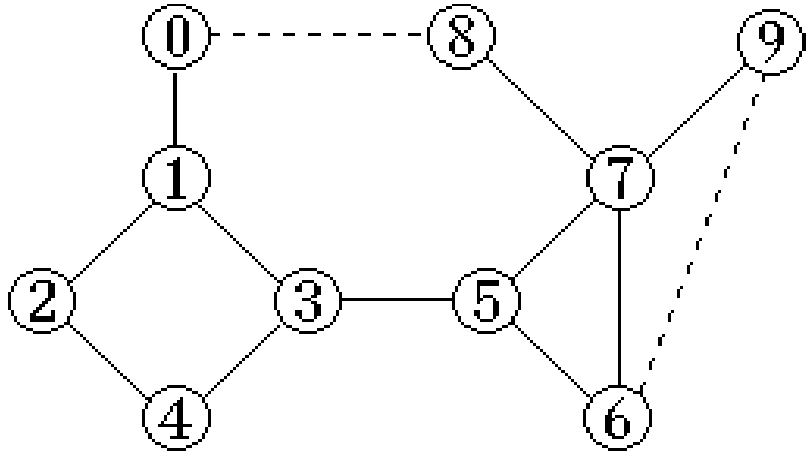




（a）深度优先生成树



）深度优先生成树(加回边)



（c）重连通图

- 在图G的每一个顶点上定义一个 *low* 值，*low*[*u*] 是从 *u* 或 *u* 的子孙出发通过回边可以到达的最低深度优先数。

  $low[u] = \min\{\, dfn[u],$
  $\min\{\, low[w] \mid w \text{ 是 } u \text{ 的一个子女} \},$
  $\min\{\, dfn[x] \mid (u, x) \text{ 是一条回边} \}\,\}$

- *u* 是关节点的充要条件是：
  - *u* 是具有两个以上子女的生成树的根
  - *u* 不是根，但它有一个子女 *w*，使得

    $low[w] \geq dfn[u]$

- 这时 *w* 及其子孙不存在指向顶点*u*的祖先的回边。

计算*dfn*与*low*的算法 (1)

```
template<class NameType, class DistType>
void Graph<NameType, DistType> ::
DfnLow ( const int x ) {
//公有函数: 从顶点x开始深度优先搜索
    int num = 1;                    // num是访问计数器
    static int * dfn = new int[NumVertices];
    static int * low = new int[NumVertices];
    //dfn是深度优先数, low是最小祖先访问顺序号
    for ( int i = 0; i < NumVertices; i++ )
      { dfn[i] = low[i] = 0; }
     //给予访问计数器num及dfn[u], low[u]初值
    DfnLow ( x, -1 );          //从根x开始
```

```
    delete [ ] dfn;   delete [ ] low;
}
```

计算dfn与low的算法 (2)

```
template<class NameType, class DistType>
void Graph<NameType, DistType> ::
DfnLow ( int u,  int v ) {
//私有函数：从顶点 u 开始深度优先搜索计算dfn
// 和low。在产生的生成树中 v 是 u 的双亲。
    dfn[u] = low[u] = num++;
    int w = GetFirstNeighbor (u);
    while ( w != -1 ) {      //对u所有邻接顶点w循环
```

```
    if ( dfn[w] == 0 ) {  //未访问过, w是u的孩子
      DfnLow ( w, u );  //从w递归深度优先搜索
      low[u] = min2 ( low[u], low[w] );
        //子女w的low[w]先求出, 再求low[u]
    }
    else if ( w != v )  //w访问过且w不是v, 是回边
      low[u] = min2 ( low[u], dfn[w] );
        //根据回边另一顶点w调整low[u]
     w = GetNextNeighbor ( u, w );
        //找顶点u在w后面的下一个邻接顶点
  }
}
```

在算法*DfnLow*增加一些语句，可把连通图的边划分到各重连通分量中。首先，根据 *DfnLow (w, u)* 的返回，计算*low*[*w*]。如果*low*[*w*] ≥ *dfn*[*u*]，则开始计算新的重连通分量。

在算法中利用一个栈，在遇到一条边时保存它。在函数***Biconnected***中就能输出一个重连通分量的所有的边。

当 *n* > 1 时输出重连通分量（1）

```
template<class NameType, class DistType>
void Graph<NameType, DistType> ::
Biconnected ( ) {
```

```
//公有函数：从顶点0开始深度优先搜索
    int num = 1;              //访问计数器num
    static int * dfn = new int[NumVertices];
    static int * low = new int[NumVertices];
    // dfn是深度优先数, low是最小祖先号
    for ( int i = 0; i < NumVertices; i++ )
       { dfn[i] = low[i] = 0; }
    DfnLow ( 0, -1 );     //从顶点 0 开始
    delete [ ] dfn;  delete [ ] low;
}
```

当 **n > 1** 时输出重连通分量（2）

```
template<class NameType, class DistType>
void Graph<NameType, DistType> ::
Biconnected ( int u,  int v ) {
//计算dfn与low，根据其重连通分量输出Graph的
//边。在产生的生成树中，v 是 u 的双亲结点，S
//是一个初始为空的栈，应声明为图的数据成员。
    int x, y, w;
    dfn[u] = low[u] = num++;
    w = GetFirstNeighbor (u);
     //找顶点u的第一个邻接顶点w
    while ( w != -1 ) {
        if ( v != w && dfn[w] < dfn[u] ) S.Push ((u,w));
        //w不是u的双亲且w先于u被访问，(u,w)进栈
```

```
if ( dfn[w] == 0 ) {      //未访问过, w是u的孩子
   Biconnected (w, u);  //从w递归深度优先访问
   low[u] = min2 ( low[u], low[w] );
    //根据先求出的low[w], 调整low[u]
   if ( low[w] >= dfn[u] ) {
    //无回边, 原来的重连通分量结束
      cout << "新重连通分量: " << end1;
      do {  (x, y) = S.Pop ( );
          cout << x << "," << y << endl;
      } while ( (x, y) 与 (u, w) 不是同一条边 );
   }    //输出该重连通分量的各边
}
else if ( w != v )          //有回边,计算
```

```
        low[u] = min2 ( low[u], dfn[w] );
        //根据回边另一顶点w调整low[u]
        w = GetNextNeighbor ( u, w );
        //找顶点u的邻接顶点w的下一个邻接顶点
    }
}
```

算法 **Biconnected** 的时间代价是 **O(*n*+*e*)**。其中 ***n*** 是该连通图的顶点数，***e*** 是该连通图的边数。

此算法的前提条件是连通图中至少有两个顶点，因为正好有一个顶点的图连一条边也没有。

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
  - 必须只使用该网络中的边来构造最小生成树；
  - 必须使用且仅使用 *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

克鲁斯卡尔 (Kruskal) 算法

- 克鲁斯卡尔算法的基本思想：

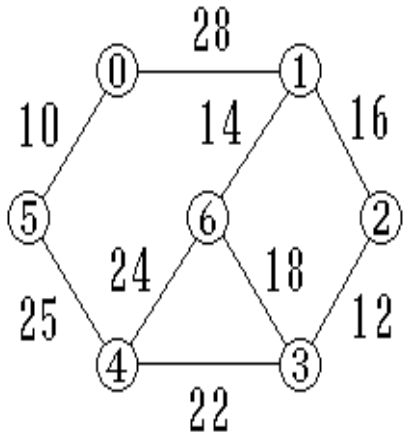
  设有一个有 n 个顶点的连通网络 $N=\{V, E\}$，最初先构造一个只有 n 个顶点，没有边的非连通图 $T=\{V, \varnothing\}$，图中每个顶点自成一个连通分量。当在 E 中选到一条具有最小权值的边时，若该边的两个顶点落在不同的连通分量上，则将此边加入到 T 中；否则将此边舍去，重新选择一条权值最小的边。如此重复下去，直到所有顶点在同一个连通分量上为止。

- 算法的框架　我们利用最小堆(MinHeap)和并查集(DisjointSets)来实现克鲁斯卡尔算法。

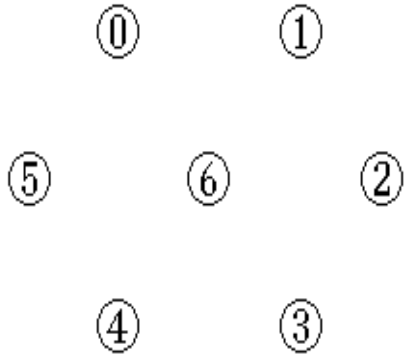
- 首先，利用最小堆来存放E中的所有的边，堆中每个结点的格式为

- 在构造最小生成树的过程中，最小堆中存放剩余的边，并且利用并查集的运算检查依附于一条边的两个顶点 ***tail***、***head*** 是否在同一个连通分量 (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个连通分量上。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。

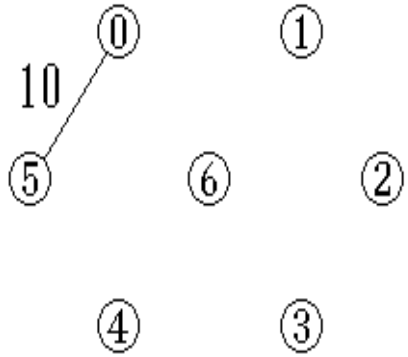
应用克鲁斯卡尔算法构造最小生成树的过程



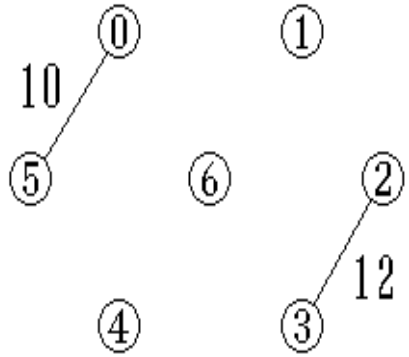
(a)



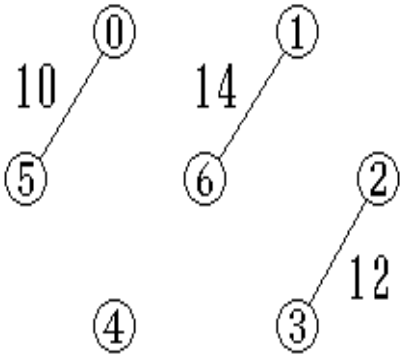
(b)



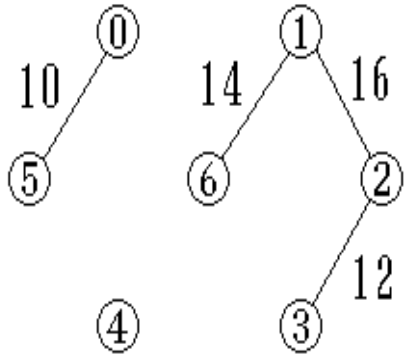
(c)



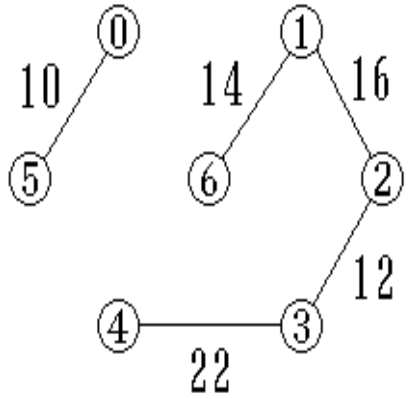
(d)



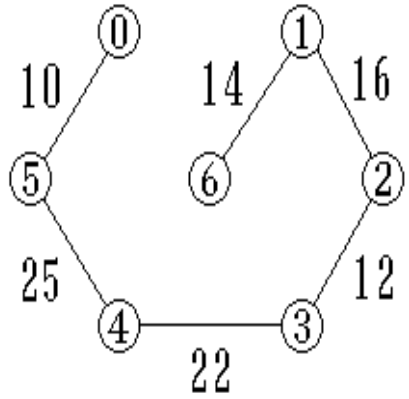
(e)



(f)



(g)



(h)

最小生成树类定义

```
const int MAXINT = 机器可表示的, 问题中不可
                   能出现的大数

class MinSpanTree;

class MSTEdgeNode {          //生成树边结点类定义
friend class MinSpanTree;
private:
    int tail, head;          //生成树各边的两顶点
    float cost;              //生成树各边的代价
};
```

```
class MinSpanTree {          //生成树的类定义
public:
    MinSpanTree ( int sz = NumVertices −1 ) :
          MaxSize (sz), n (0)
      { edgevalue = new MSTEdgeNode[MaxSize]; }
    int Insert ( MSTEdgeNode & item );
      //将item加到最小生成树中
protected:
    MSTEdgeNode *edgevalue;          //边值数组
    int MaxSize, n;                  //最大边数,当前边数
};
```

利用克鲁斯卡尔算法建立最小生成树

```
void Graph<string, float> ::
Kruskal ( MinSpanTree& T ) {
    MSTEdgeNode  e;                          //边结点辅助单元
    MinHeap<MstEdgeNode> H(CurrentEdges);
    int NumVertices = VerticesList.last;     //顶点个数
    UFSets F (NumVertices);                  //并查集F 并初始化
    for ( int u = 0; u < NumVertices; u++ ) //邻接矩阵
       for ( int v = u +1; v < NumVertices; v++ )
          if ( Edge[u][v] != MAXNUM ) {    //所有边
             e.tail = u;  e.head = v;
             e.cost = Edge[u][v];  H.Insert (e);  //插入堆
          }
```

```
    int count = 1;      //最小生成树加入边数的计数
    while ( count < NumVertices ) {
        H.RemoveMin ( e );      //从堆中退出一条边
        u = F.Find ( e.tail );      //检测两端点的根
        v = F.Find ( e.head );
        if ( u != v ) {      //根不同不在同一连通分量
            F.Union ( u, v ); //合并
            T.Insert ( e );      //该边存入最小生成树 T
            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

- 在建立最小堆时需要检测邻接矩阵，计算的时间代价为 $O(n^2)$。且做了 e 次堆插入操作，每次插入调用了一个堆调整算法*FilterUp*( )算法，因此堆插入需要的时间代价为 $O(e\log_2 e)$。
- 在构造最小生成树时，最多调用了 e 次出堆操作*RemoveMin*( )，$2e$ 次并查集的*Find*( )操作，n-1次*Union*( )操作，计算时间分别为 $O(e\log_2 e)$，$O(e\log_2 n)$和$O(n)$。
- 总的计算时间为 $O(e\log_2 e + e\log_2 n + n^2 + n)$

普里姆(Prim)算法

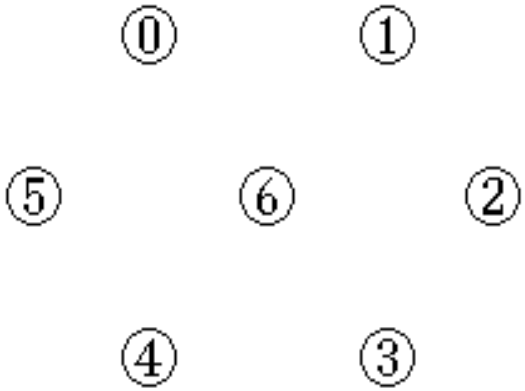
- 普里姆算法的基本思想：

  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的顶点集合U中。

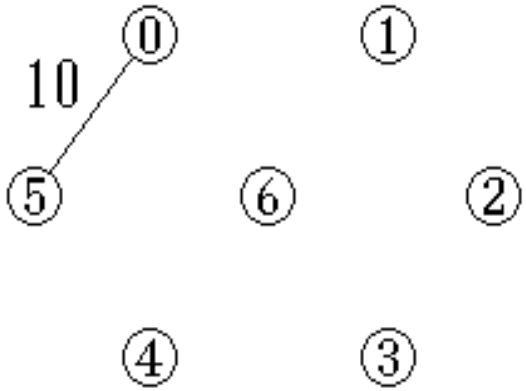
  以后每一步从一个顶点在U中，而另一个顶点不在U中的各条边中选择权值最小的边(u, v)，把它的顶点加入到集合U中。如此继续下去，直到网络中的所有顶点都加入到生成树顶点集合U中为止。

- 采用邻接矩阵作为图的存储表示。

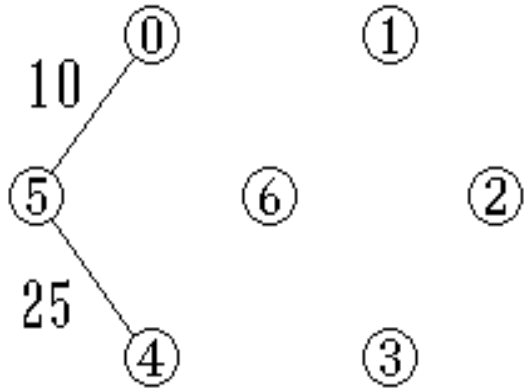
用普里姆算法构造最小生成树的过程



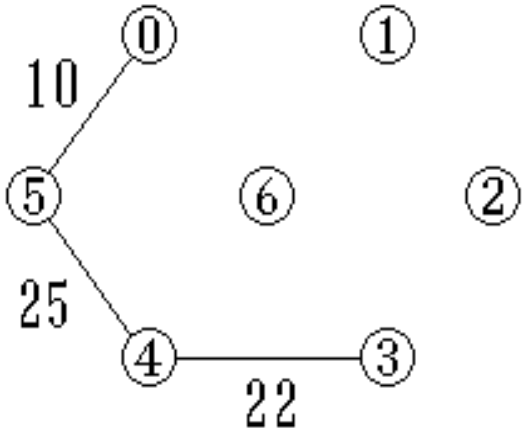
(a)



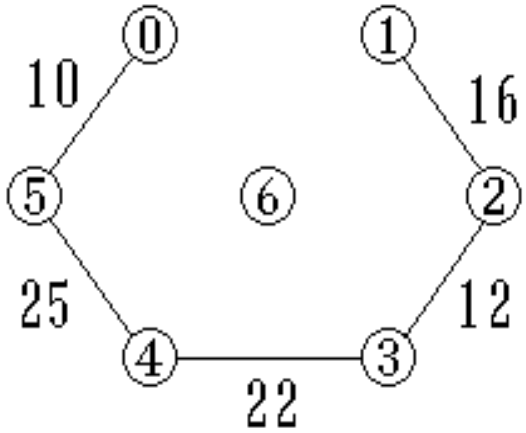
(b)



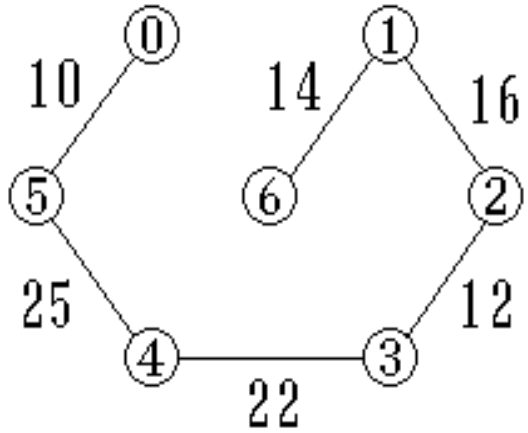
(c)



(d)

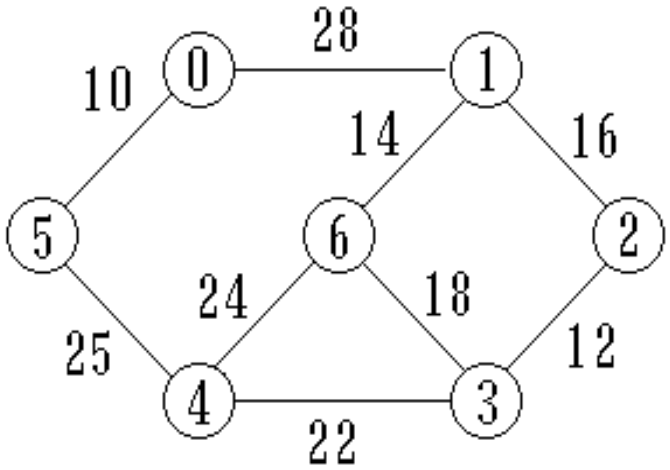


(e)



(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

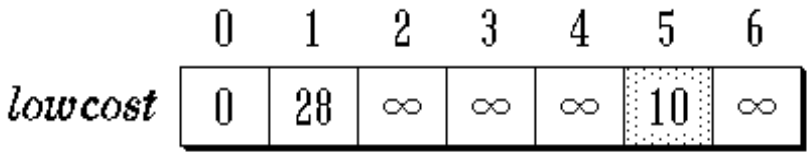


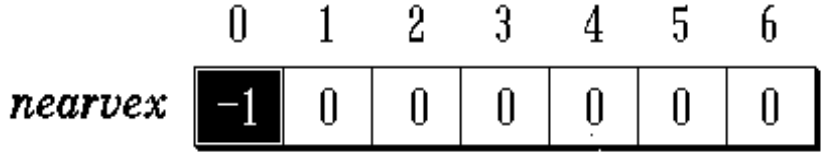
（a）连通网络

	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

（b）邻接矩阵表示

- 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：



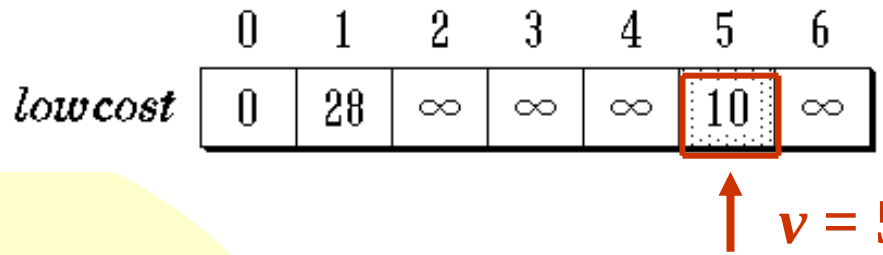


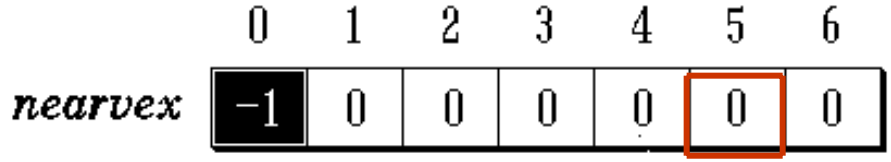
- 然后反复做以下工作：
  - 在 *lowcost* [ ]中选择

    nearvex[*i*] ≠ −1 && *lowcost*[*i*]最小的边 *i*

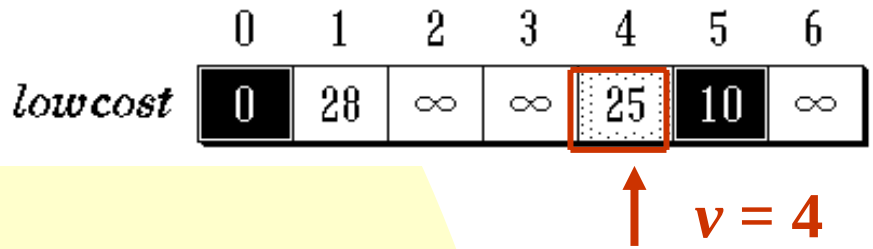
    用 *v* 标记它。则选中的权值最小的边为 (*nearvex*[*v*], *v*)，相应的权值为 *lowcost*[*v*]。
  - 将 *nearvex*[*v*] 改为 −1，表示它已加入生成树顶点集合。将边 (*nearvex*[*v*], *v*, *lowcost*[*v*]) 加入生成树的边集合。

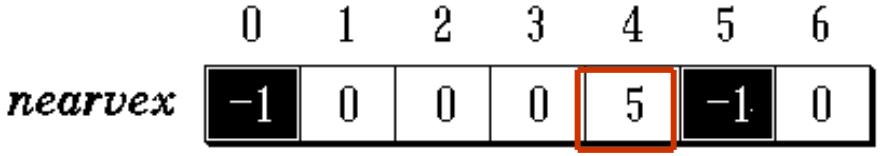
- 取 *lowcost*[*i*] = min{ *lowcost*[*i*], *Edge*[*v*][*i*] }，即用生成树顶点集合外各顶点 *i* 到刚加入该集合的新顶点 *v* 的距离 *Edge*[*v*][*i*] 与原来它们到生成树顶点集合中顶点的最短距离 *lowcost*[*i*] 做比较，取距离近的作为这些集合外顶点到生成树顶点集合内顶点的最短距离。
- 如果生成树顶点集合外顶点 *i* 到刚加入该集合的新顶点 *v* 的距离比原来它到生成树顶点集合中顶点的最短距离还要近，则修改 ***nearvex*[*i*]**：*nearvex*[*i*] = *v*。表示生成树外顶点*i*到生成树内顶点*v*当前距离最近。



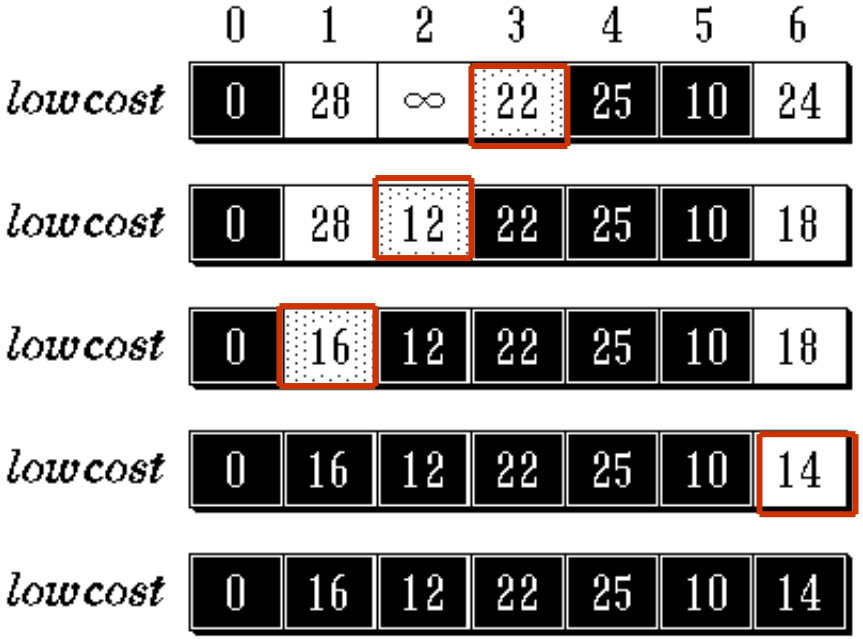


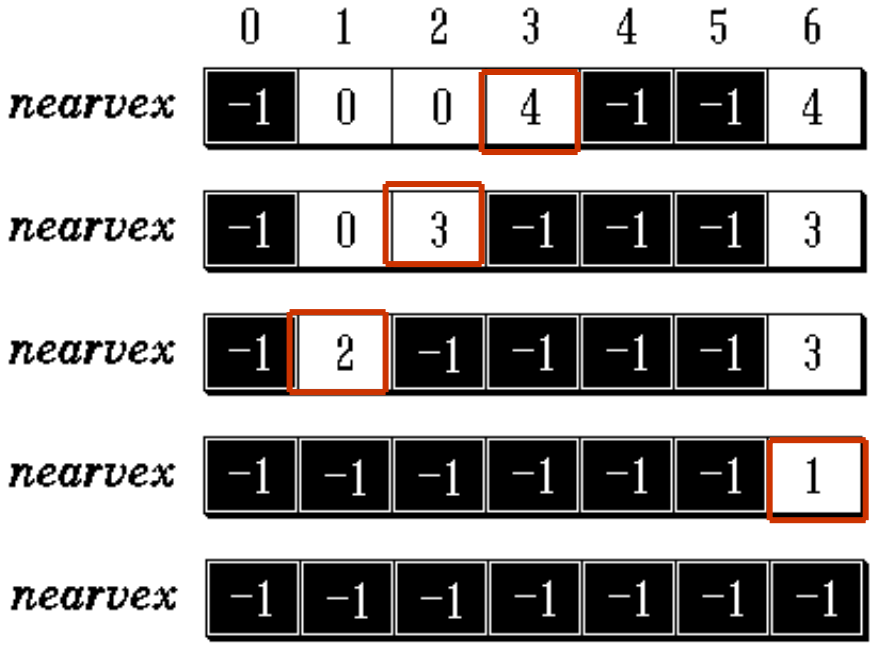
顶点5加入生成树顶点集合:





继续重复得:





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 Graph<string, float> ::
Prim ( MinSpanTree &T ) {
    int NumVertices = VerticesList.last;   //图顶点数
    float * lowcost = new float[NumVertices];
    int * nearvex = new int[NumVertices];
    for ( int i = 1; i < NumVertices; i++ ) {
        lowcost[i] = Edge[0][i]; //顶点0到各边的代价
        nearvex[i] = 0;                  //及最短带权路径
    }
```

```
nearvex[0] = -1;      //顶点0加到生成树顶点集合
MSTEdgeNode e;    //最小生成树结点辅助单元
for ( i = 1; i < NumVertices; i++ ) {
 //循环n-1次, 加入n-1条边
    float min = MAXNUM;  int v = 0;
    for ( int j = 0; j < NumVertices; 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, v是当前具最小权值的边的位置
    if ( v ) {    //v==0表示再也找不到要求顶点了
       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;  //作该边已加入生成树标记
        for ( j = 1; j < NumVertices; j++ )
            if ( nearvex[j] != −1 &&   // j 不在生成树中
                 Edge[v][j] < lowcost[j] ) {    //需要修改
              lowcost[j] = Edge[v][j];
              nearvex[j] = v;
            }
    }
  }
}
```

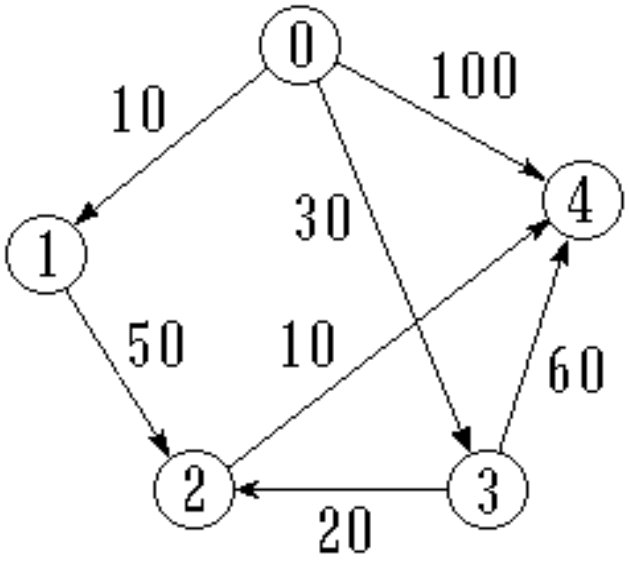
分析以上算法，设连通网络有 n 个顶点，则该算法的时间复杂度为 $O(n^2)$，它适用于边稠密的网络。

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，依次类推，直到从顶点**v**到其它各顶点的最短路径全部求出为止。
- 举例说明



（a）带权有向图

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

（b）邻接矩阵

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 0 60

- 引入一个辅助数组*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没有边，则*dist*[*i*]为+∞。
- 一般情况下，假设*S*是已求得的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 ($v_x \in V\text{-}S$)的路径中的一条。
- 每次求得一条最短路径之后，其终点v_k加入集合*S*，然后对所有的$v_i \in V\text{-}S$，修改其*dist*[*i*]值。

Dijkstra算法可描述如下:

☆ 初始化: $S \leftarrow \{ v_0 \}$;

$dist[j] \leftarrow Edge[0][j]$, $j = 1, 2, \ldots, n\text{-}1$;

// n为图中顶点个数

求出最短路径的长度:

$dist[k] \leftarrow \min\{ dist[i] \}$, $i \in V\text{-}S$;

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

修改:

$dist[i] \leftarrow \min\{ dist[i], dist[k] + Edge[k][i] \}$,

对于每一个 $i \in V\text{-}S$;

判断: 若$S = V$, 则算法结束, 否则转。

用于计算最短路径的图邻接矩阵类的定义

```
const int NumVertices = 6;   //图中最大顶点个数
class Graph {                //图的类定义
private:
    float Edge[NumVertices][NumVertices];
    float dist[NumVertices];  //最短路径长度数组
    int path[NumVertices];    //最短路径数组
    int S[NumVertices];       //最短路径顶点集
public:
    void ShortestPath ( int,  int );
    int choose ( int );
};
```

计算从单个顶点到其它各个顶点的最短路径

```
void Graph :: ShortestPath ( int n,  int v ){
//Graph是一个具有 n 个顶点的带权有向图, 各边
//上的权值由Edge[i][j]给出。本算法建立起一个
//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。
    for ( int i = 0; i < n; i++) {
        dist[i] = Edge[v][i];     //dist数组初始化
        S[i] = 0;
        if ( i != v && dist[i] < MAXNUM) path[i] = v;
        else path[i] = −1;        //path数组初始化
    }
```

```
S[v] = 1;   dist[v] = 0;      //顶点v加入顶点集合
//选择当前不在集合S中具有最短路径的顶点u
for ( i = 0; i < n−1; i++ ) {
    float min = MAXNUM;  int u = v;
    for ( int 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 ( int w = 0; w < n; w++ )    //修改
        if ( !S[w] && Edge[u][w] < MAXNUM &&
                    dist[u] + Edge[u][w] < dist[w] ) {
            dist[w] = dist[u] + Edge[u][w];
            path[w] = u;
        }
```

```
    }
}
```

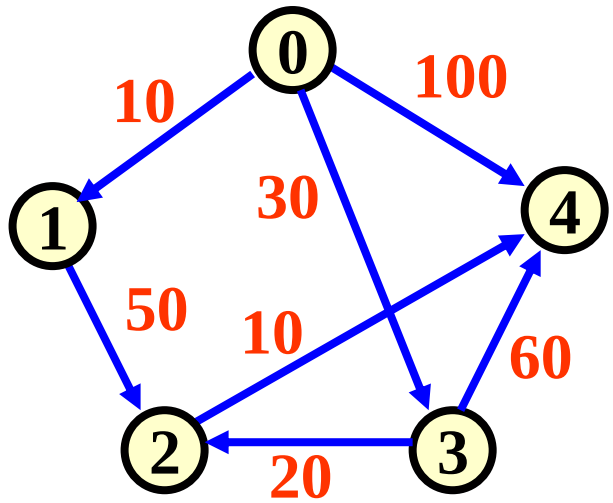
算法的时间复杂度：$O(n^2)$

Dijkstra算法中各辅助数组的变化

选取终点	顶点1			顶点2			顶点3			顶点4		
	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**

如何从表中读取源点**0**到终点**v**的最短路径？举顶点**4**为例

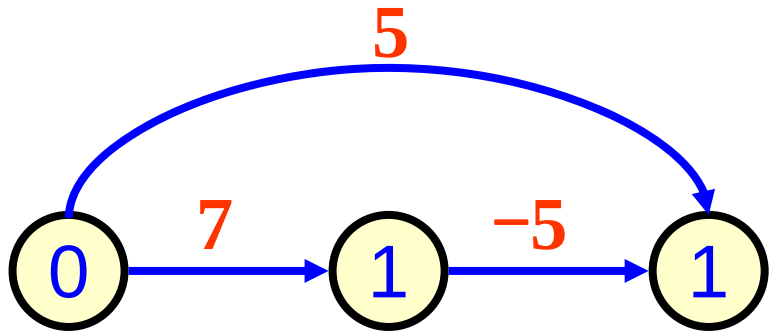
path[4] = 2 → path[2] = 3 → path[3] = 0，反过来排列，得到路径**0, 3, 2, 4**，这就是源点**0**到终点**4**的最短路径。



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。

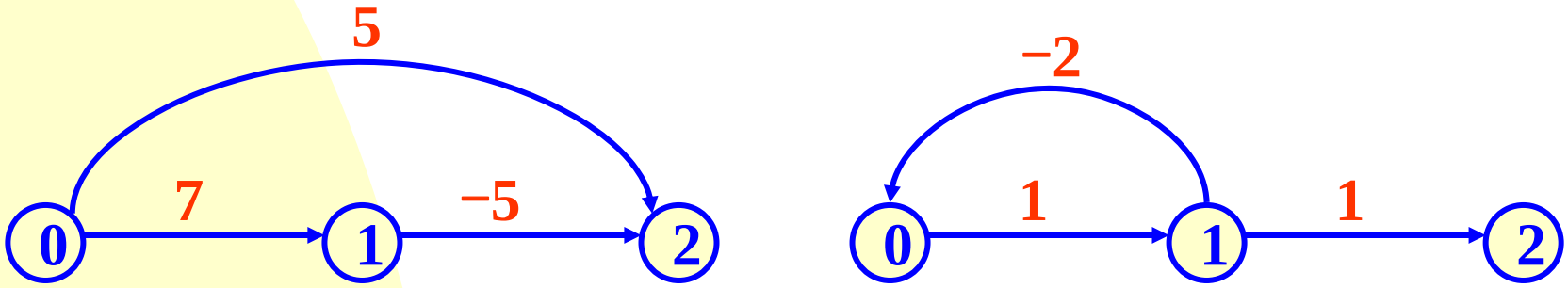
  若设源点$v=0$，使用**Dijkstra**算法所得结果



选取顶点	顶点 0			顶点 1			顶点 2		
	$S[0]$	$d[0]$	$p[0]$	$S[1]$	$d[1]$	$p[1]$	$S[2]$	$d[2]$	$p[2]$
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**，它小于算法中计算出来的***dist*[2]**的值。

- **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有由带负权值的边组成的回路。**



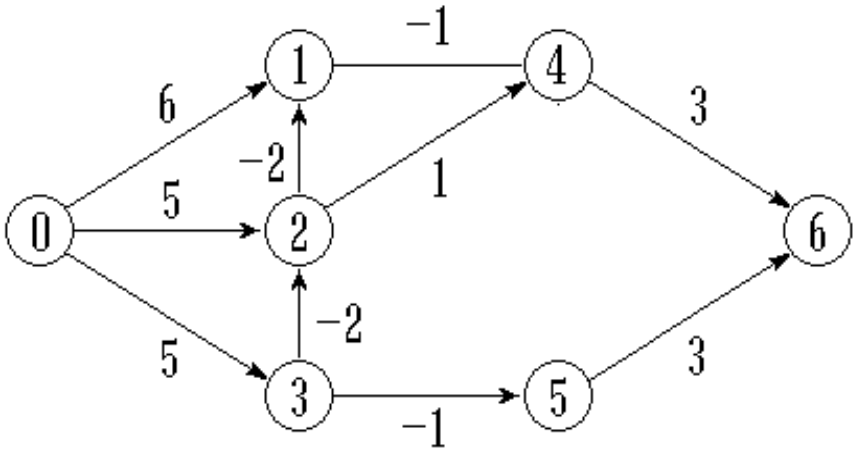
- 当图中没有由带负权值的边组成的回路时，有***n***个顶点的图中任意两个顶点之间如果存在最短路径，此路径最多有***n*-1**条边。
- 我们将以此为依据考虑计算从源点***v***到其它顶点***u***的最短路径的长度***dist*[*u*]**。

- **Bellman-Ford**方法构造一个最短路径长度数组序列$dist^1[u]$，$dist^2[u]$，…，$dist^{n-1}[u]$。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的长度，$dist^1[u] = Edge[v][u]$；$dist^2[u]$是从源点 *v* 最多经过两条边到达终点 *u* 的最短路径的长度，$dist^3[u]$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 $dist^k[u]$。

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

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

- 设已经求出 $dist^{k-1}[j]$, $j = 0, 1, \ldots, n-1$, 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 的最短路径的长度。
- 从图的邻接矩阵中可以找到各个顶点 j 到达顶点 u 的距离 $Edge[j][u]$，计算 $\min\{ dist^{k-1}[j] + Edge[j][u] \}$，可得从源点 v 绕过各个顶点，最多经过不构成带负长度边回路的 k 条边到达终点u的最短路径的长度。
- 用它与$dist^{k-1}[u]$比较，取小者作为$dist^k[u]$的值。



(a)

$$\begin{array}{c} \begin{matrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 \end{matrix} \\ \begin{pmatrix} 0 & 6 & 5 & 5 & \infty & \infty & \infty \\ \infty & 0 & \infty & \infty & -1 & \infty & \infty \\ \infty & -2 & 0 & \infty & 1 & \infty & \infty \\ \infty & \infty & -2 & 0 & \infty & -1 & \infty \\ \infty & \infty & \infty & \infty & 0 & \infty & 3 \\ \infty & \infty & \infty & \infty & \infty & 0 & 3 \\ \infty & \infty & \infty & \infty & \infty & \infty & 0 \end{pmatrix} \begin{matrix} 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{matrix} \end{array}$$

(b)

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 Graph::BellmanFord ( int n,  int v ) {
//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。
    for ( int i = 0; i < n; i++ ) {
        dist[i] = Edge[v][i];
        if ( i != v && dist[i] < MAXINT )
            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++ )
                if ( Edge[i][u] > 0 &&
                        Edge[i][u] < MAXNUM &&
                        dist[u] > dist[i] + Edge[i][u] ) {
                    dist[u] = dist[i] + Edge[i][u];
                    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。

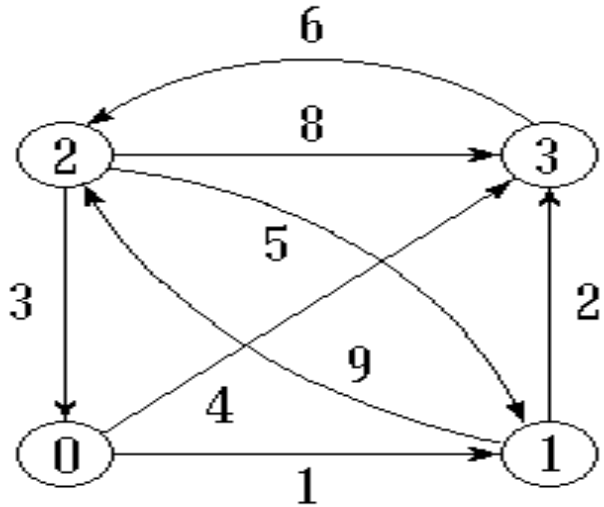
所有各对顶点之间的最短路径

```
void Graph :: AllLengths ( int n ) {
//Edge[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 的前一顶点的顶点号，在求
//最短路径时图的类定义中要修改path为
//path[NumVertices][NumVertices]。
```

```
    for ( int i = 0; i < n; i++ )    //矩阵a与path初始化
        for ( int j = 0; j < n; j++ ) {
            a[i][j] = Edge[i][j];
            if ( i <> j && a[i][j] < MAXINT )
                path[i][j] = i;              // i 到 j 有路径
            else path[i][j] = 0;             // i 到 j 无路径
        }
    for ( int k = 0; k < n; 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
}
```

	$A^{(-1)}$				$A^{(0)}$				$A^{(1)}$				$A^{(2)}$				$A^{(3)}$			
	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



$$\begin{array}{c} \begin{matrix} 0 & 1 & 2 & 3 \end{matrix} \\ \begin{pmatrix} 0 & 1 & \infty & 4 \\ \infty & 0 & 9 & 2 \\ 3 & 5 & 0 & 8 \\ \infty & \infty & 6 & 0 \end{pmatrix} \begin{matrix} 0 \\ 1 \\ 2 \\ 3 \end{matrix} \end{array}$$

以 $Path^{(3)}$为例，对最短路径的读法加以说明。从$A^{(3)}$知，顶点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>。

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。

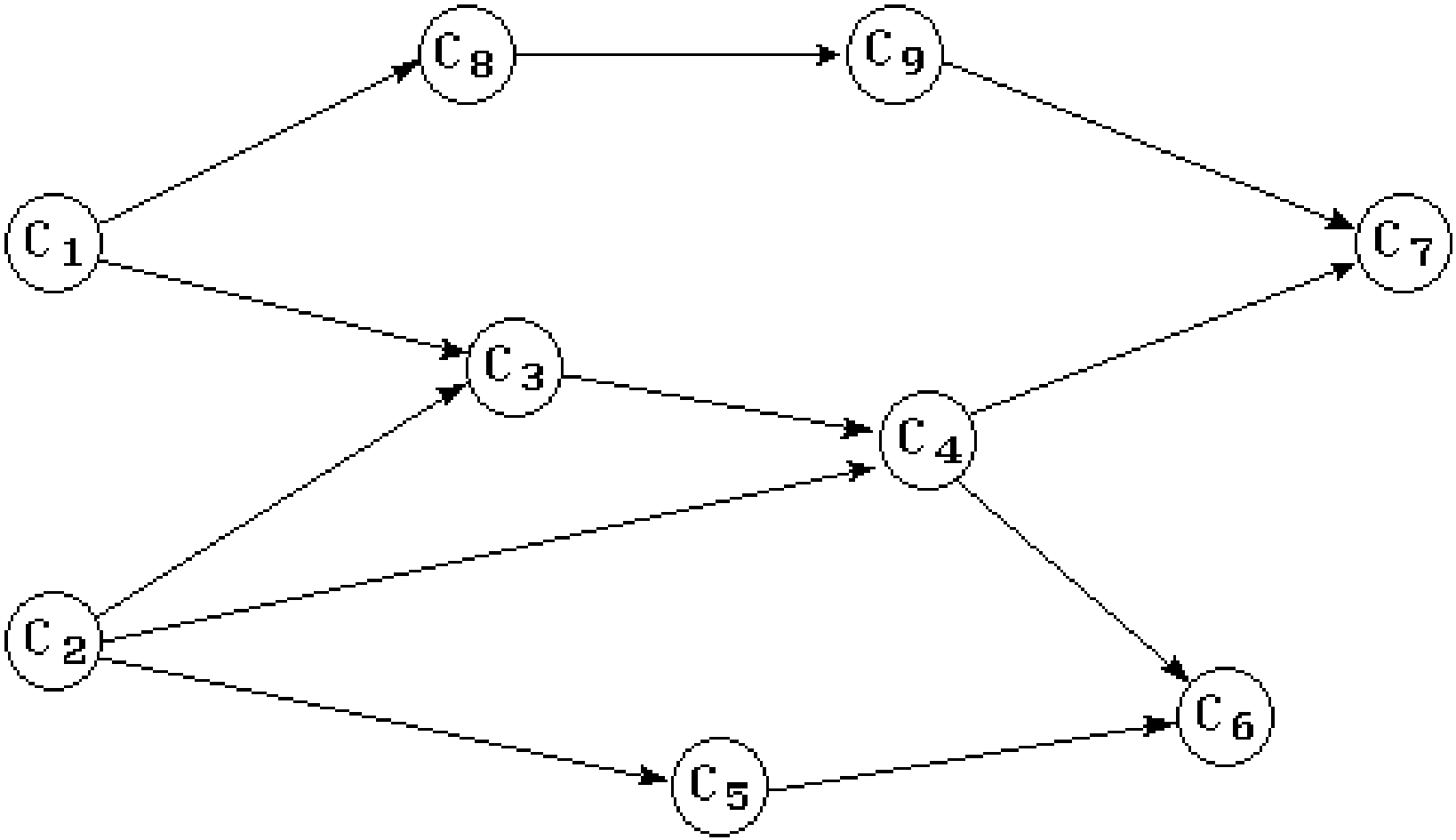
本章给出的求解最短路径的算法不仅适用于带权有向图，对带权无向图也可以适用。因为带权无向图可以看作是有往返二重边的有向图，只要在顶点v_i与v_j之间存在无向边(v_i, v_j)，就可以看成是在这两个顶点之间存在权值相同的两条有向边$<v_i, v_j>$和$<v_j, v_i>$。

活动网络 (Activity Network)

用顶点表示活动的网络 (AOV网络)

- 计划、施工过程、生产流程、程序流程等都是“工程”。除了很小的工程外，一般都把工程分为若干个叫做“活动”的子工程。完成了这些活动，这个工程就可以完成了。
- 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

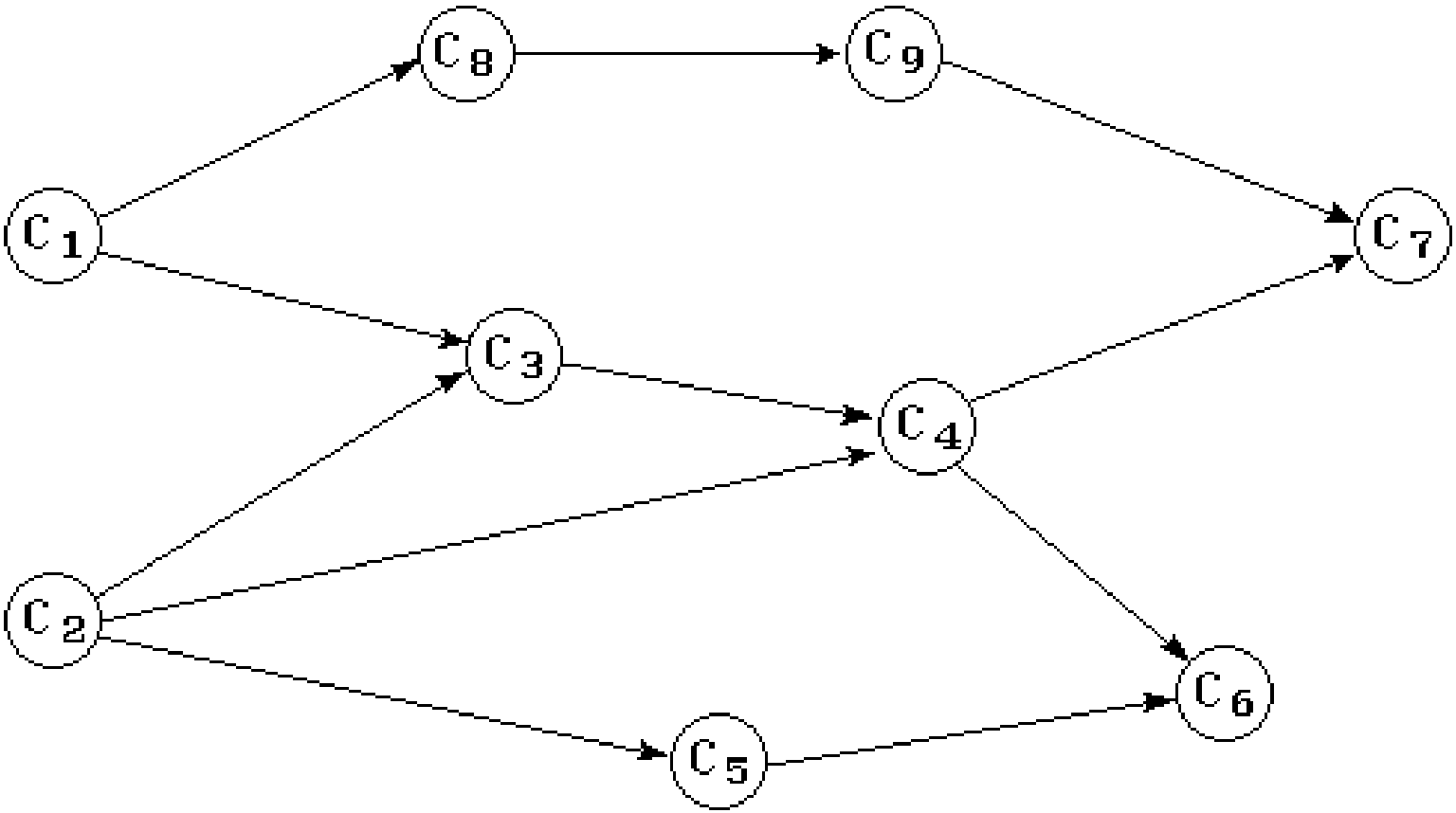


学生课程学习工程图

- 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，如果活动V_i必须在活动V_j之前进行，则存在有向边<V_i, V_j>， **AOV网络**中不能出现有向回路，即有向环。在**AOV网络**中如果出现了有向环，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AOV网络中必定不会出现有向环；相反，如果得不到满足要求的拓扑有序序列，则说明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

☆ 输入AOV网络。令 *n* 为顶点个数。

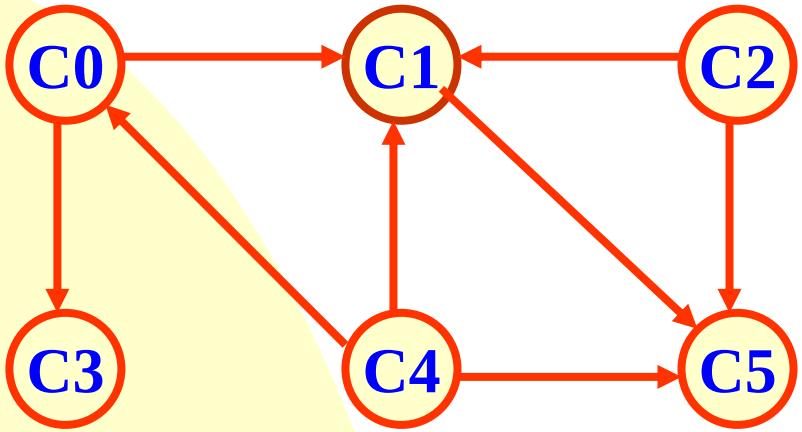
① 在AOV网络中选一个没有直接前驱的顶点，并输出之；

② 从图中删去该顶点，同时删去所有它发出的有向边；

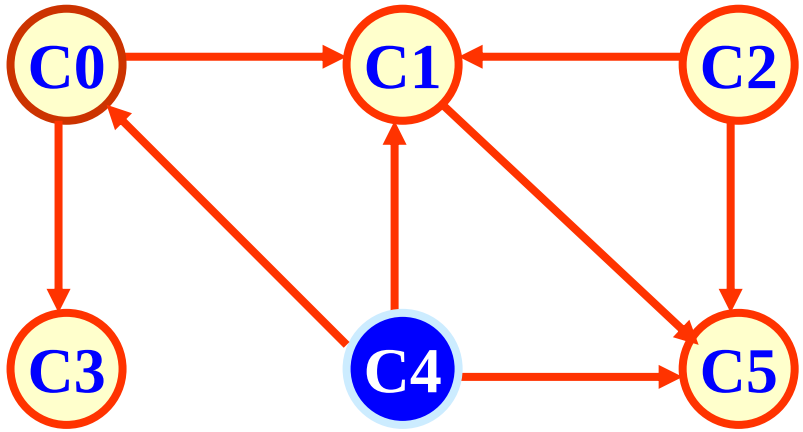
③ 重复以上①、②步，直到

- 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。这说明图中还剩下一些顶点，它们都有直接前驱，再也找不到没有前驱的顶点了。这时AOV网络中必定存在有向环。

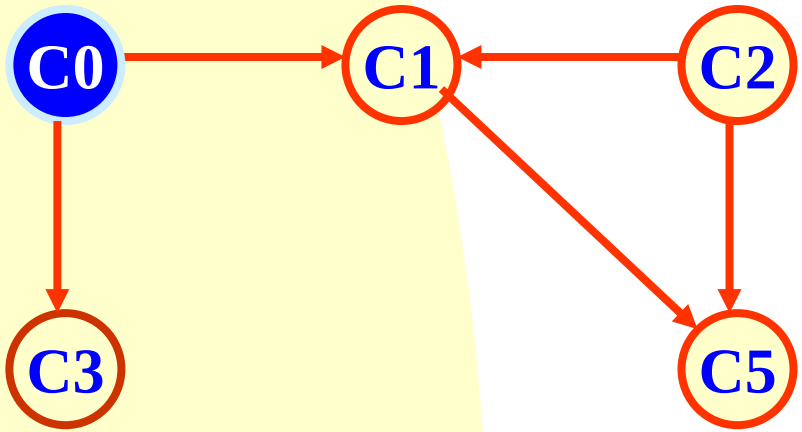
拓扑排序的过程



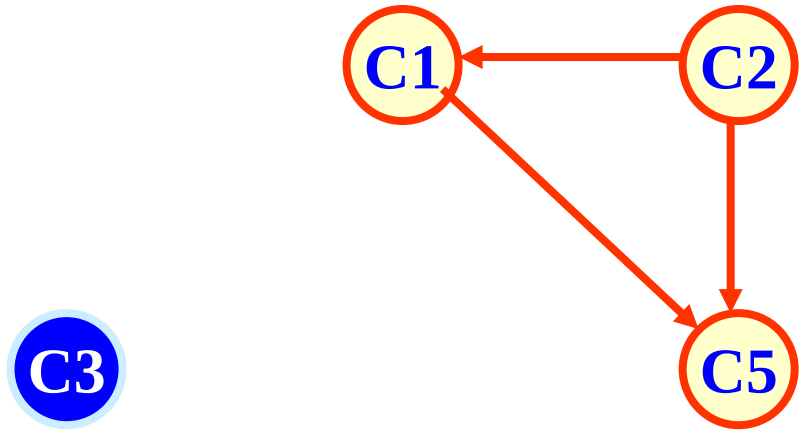
(a) 有向无环图



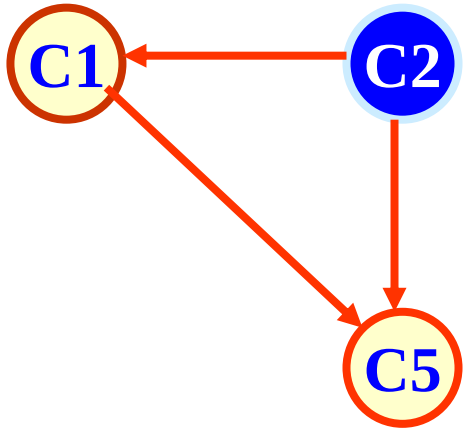
(b) 输出顶点C4



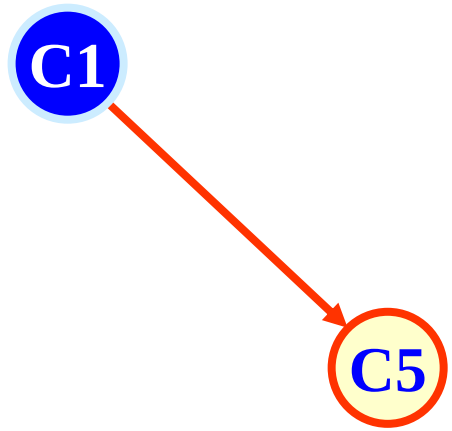
(c) 输出顶点C0



(d) 输出顶点C3



(e) 输出顶点C2



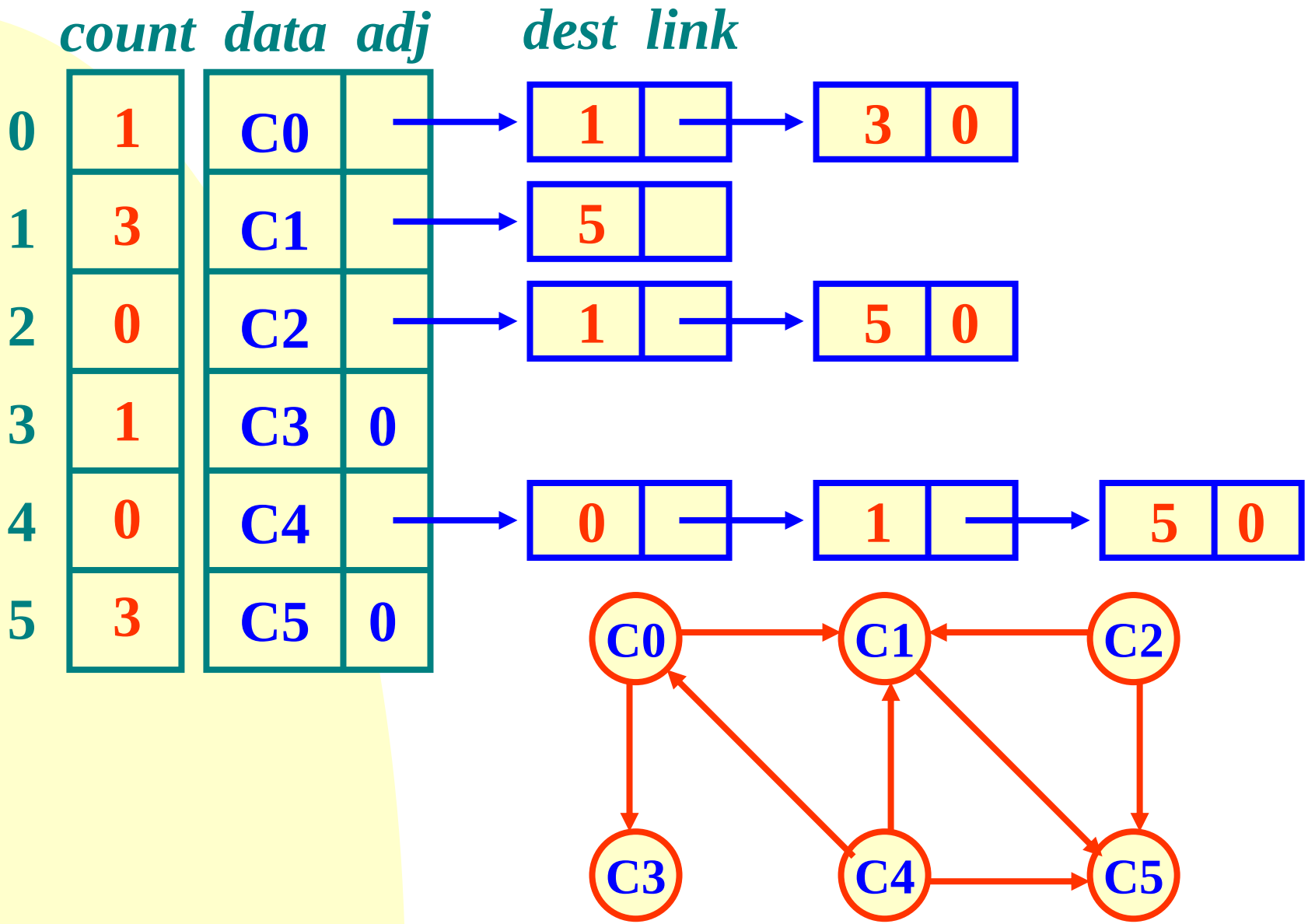
(f) 输出顶点C1

(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，也排出了先后次序关系。

AOV网络及其邻接表表示



- 在邻接表中增设了一个数组*count*[ ]，记录各个顶点入度。入度为零的顶点即无前驱的顶点。
- 在输入数据前，邻接表的顶点表*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<int> * 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
```

- 在拓扑排序算法中，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。只要出现入度为零的顶点，就将它加入栈中。
- 使用这种栈的拓扑排序算法可以描述如下:

(1) 建立入度为零的顶点栈;

(2) 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;

- (3) 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。

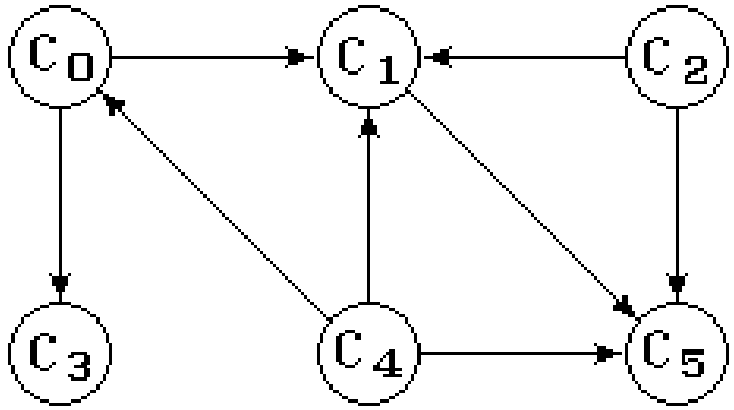
- 在算法实现时，为了建立入度为零的顶点栈，可以不另外分配存储空间，直接利用入度为零的顶点的*count*[ ]数组元素。我们设立了一个栈顶指针*top*，指示当前栈顶的位置，即某一个入度为零的顶点位置。栈初始化时置*top* = -1，表示空栈。
- 将顶点 *I* 进栈时，执行以下指针的修改:

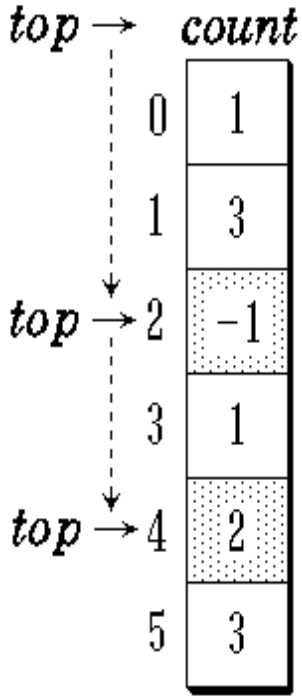
  count[*i*] = *top*; *top* = *i*;

  // *top*指向新栈顶*i*, 原栈顶元素放在*count*[*i*]中
- 退栈操作可以写成:

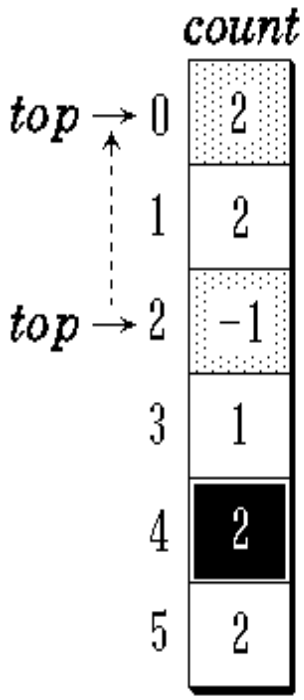
  j = *top*; *top* = *count*[*top*];

  //位于栈顶的顶点的位置记于 *j*, *top*退到次栈顶

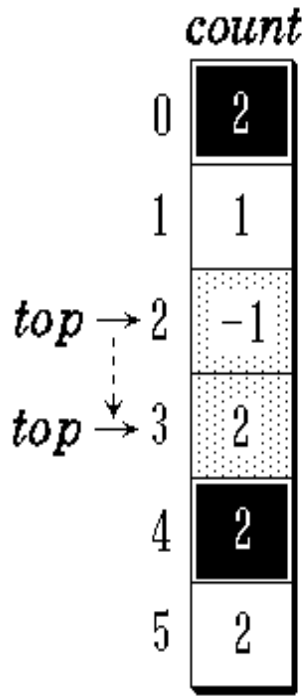




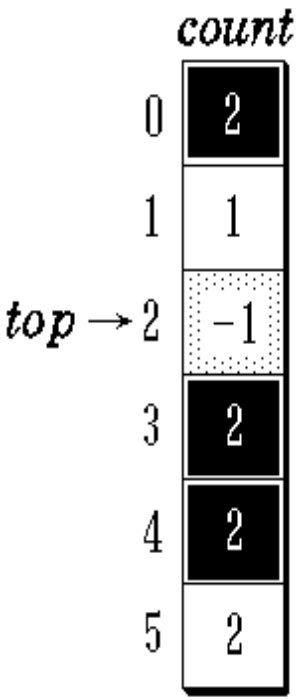
建顶点栈



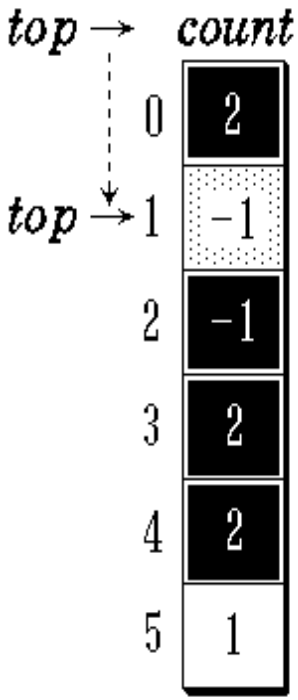
顶点4出栈



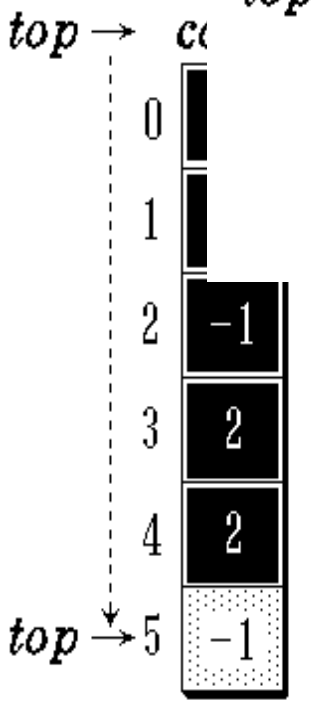
顶点0出栈



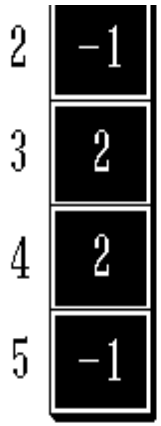
顶点3出栈



顶点2出栈



顶点1出栈



顶点5出栈

拓扑排序时入度为零的顶点栈的变化

```
class Graph {                                    //图的邻接表的类定义
friend class <int, float>Vertex;
friend class <float>Edge;
private:
    Vertex<int, float> *NodeTable;
    int *count;
    int n;
public:
    Graph ( const int vertices = 0 ) : n ( vertices ) {
        NodeTable = new vertex<int, float>[n];
        count = new int[n];
    };
    void TopologicalOrder ( );
}
```

拓扑排序的算法

```
void Graph :: TopologicalSort ( ) {
    int top = −1;                    //入度为零的顶点栈初始化
    for ( int i = 0; i < n; i++ )    //入度为零顶点进栈
        if ( count[i] == 0 ) { count[i] = top;  top = i; }
    for ( i = 0; i < n; i++ )        //期望输出n个顶点
        if ( top == −1 ){            //中途栈空,转出
            cout << "网络中有回路(有向环)" << endl;
            return;
        }
        else {                       //继续拓扑排序
            int j = top;  top = count[top];    //退栈
```

```
        cout << j << endl;                  //输出
        Edge<float> * l = NodeTable[j].adj;
        while ( l ) {                       //扫描该顶点的出边表
            int k = l→dest;                 //另一顶点
            if ( --count[k] == 0 )          //该顶点入度减一
               { count[k] = top;  top = k; }    //减至零
            l = l→link;
        }
    }
}
```

- 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了 e 次。所以总的时间复杂度为$O(n+e)$。

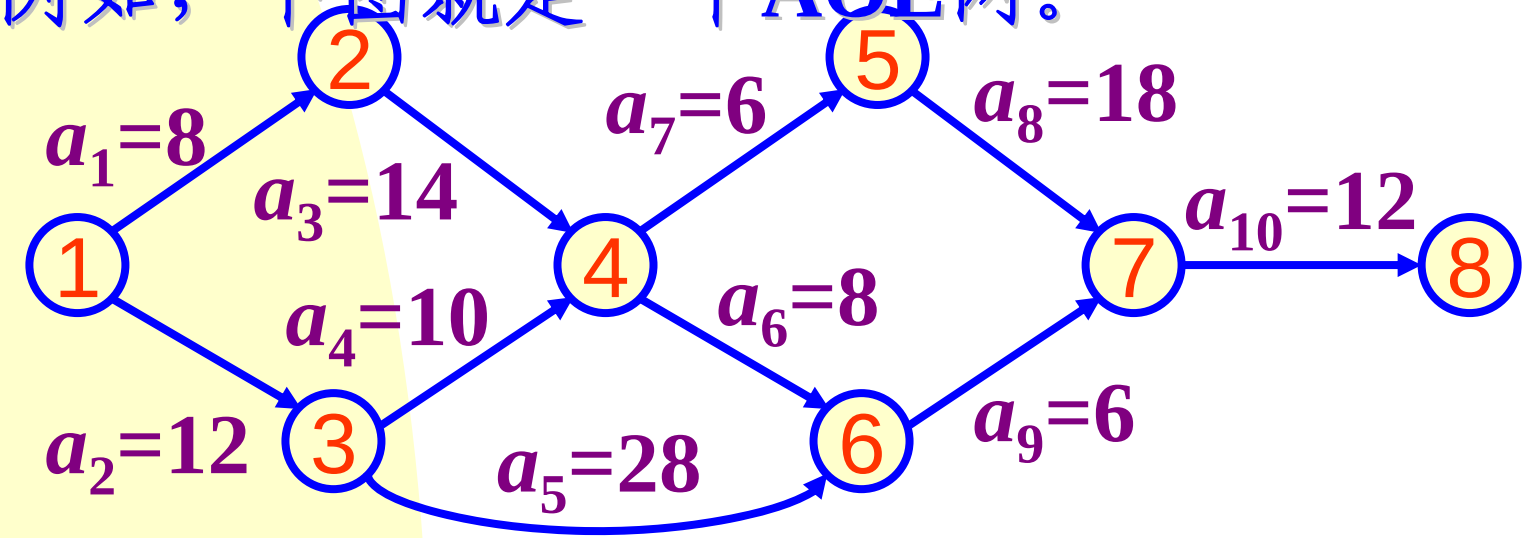
用边表示活动的网络(AOE网络)

- 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:
  (1) 完成整个工程至少需要多少时间(假设网络中没有环)?

(2) 为缩短完成工程所需的时间, 应当加快哪些活动?

- 在**AOE**网络中, 有些活动顺序进行，有些活动并行进行。
- 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，只要找到了关键活动，就可以找到关键路径.
- 例如，下图就是一个AOE网。



定义几个与计算关键活动有关的量:

☆ 事件V_i的最早可能开始时间$Ve(i)$
是从源点V_0到顶点V_i的最长路径长度。

事件V_i的最迟允许开始时间$Vl[i]$
是在保证汇点V_{n-1}在$Ve[n-1]$时刻完成的前提下，事件V_i的允许的最迟开始时间。

活动a_k的最早可能开始时间 $e[k]$
设活动a_k在边$< V_i, V_j >$上，则$e[k]$是从源点V_0到顶点V_i的最长路径长度。因此, $e[k] = Ve[i]$。

活动a_k的最迟允许开始时间 $l[k]$
$l[k]$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。

$l[k] = Vl[j] - dur(<i, j>)$。

其中，$dur(<i, j>)$是完成 a_k 所需的时间。

- 时间余量 $l[k] - e[k]$

  表示活动 a_k 的最早可能开始时间和最迟允许开始时间的时间余量。$l[k] == e[k]$ 表示活动 a_k 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 $e[k]$ 与 $l[k]$，以判别是否 $l[k] == e[k]$.
- 为求得 $e[k]$与 $l[k]$，需要先求得从源点 V_0 到各个顶点 V_i 的 $Ve[i]$ 和 $Vl[i]$。
- 求$Ve[i]$的递推公式

- 从 $Ve[0]=0$开始，向前递推

$$Ve[i]=\max_{j}\{Ve[j]+dur(<V_j,V_i>)\},$$
$$<V_j,V_i>\in S2,\quad i=1,2,\ldots,n-1$$

其中，$S2$是所有指向顶点V_i的有向边$<V_j,V_i>$的集合。

- 从$Vl[n-1]=Ve[n-1]$开始，反向递推

$$Vl[i]=\min_{j}\{Vl[j]-dur(<V_i,V_j>)\},$$
$$<V_i,V_j>\in S1,\ i=n-2,n-3,\ldots,0$$

其中，$S1$是所有从顶点V_i发出的有向边$<V_i,V_j>$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, 它的持续时间用dur (<V_i, V_j>) 表示，则有

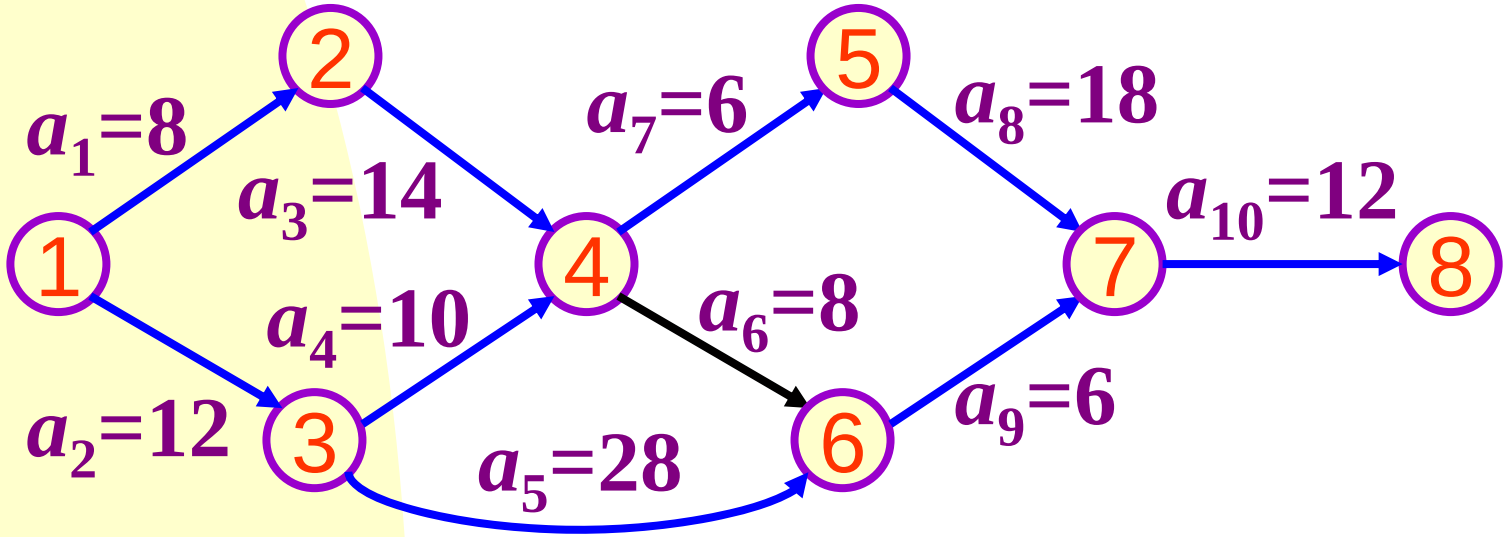
  $e[k] = Ve[i]$;

  $l[k] = Vl[j] - dur(<V_i, V_j>)$; k = 1, 2, …, e。

  这样就得到计算关键路径的算法。
- 计算关键路径时，可以一边进行拓扑排序一边计算各顶点的$Ve[i]$。为了简化算法，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。算法在求$Ve[i]$, i=0, 1, …, n-1时按拓扑有序的顺序计算，在求$Vl[i]$, i=n-1, n-2, …, 0时按逆拓扑有序的顺序计算。

	1	2	3	4	5	6	7	8
Ve	0	8	12	22	28	40	46	58
Vl	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



利用关键路径法求AOE网的各关键活动

```
void graph :: CriticalPath ( ) {
//在此算法中需要对邻接表中单链表的结点加以
//修改, 在各结点中增加一个int域 cost, 记录该结
//点所表示的边上的权值。
    int i, j;  int p, k;  float e, l;
    float * Ve = new float[n];
    float * Vl = new float[n];
    for ( i = 0; i < n; i++ ) Ve[i] = 0;
    for ( i = 0; i < n; i++ ) {
        Edge<float> *p = NodeTable[i].adj;
        while ( p != NULL ) {
            k = p→dest;
```

```
            if ( Ve[i] + p→cost > Ve[k] )
                Ve[k] = Ve[i] + p→cost;
            p = p→link;
        }
    }
    for ( i = 0; i < n; i++ ) Vl[i] = Ve[n−1];
    for ( i = n−2; i; i−− ) {
        p = NodeTable[i].adj;
        while ( p != NULL ) {
            k = p→dest;
            if ( Vl[k] − p→cost < Vl[i])
                Vl[i] = Vl[k] − p→cost;
            p = p→link;
```

```
        }
    }
    for ( i = 0; i < n; i++ ) {
        p = NodeTable[i].adj;
        while ( p != NULL ) {
            k = p→dest;
            e = Ve[i];  l = Vl[k] − p→cost;
            if ( l == e ) cout << "<" << i << "," << k
                << “>” << “是关键活动” << endl;
            p = p→link;
        }
    }
}
```

在拓扑排序求 $Ve[i]$ 和逆拓扑有序求 $Vl[i]$ 时，所需时间为$O(n+e)$，求各个活动的 $e[k]$和 $l[k]$时所需时间为$O(e)$，总共花费时间仍然是$O(n+e)$。

注意

- 所有顶点按拓扑有序的次序编号
- 仅计算 $Ve[i]$ 和 $Vl[i]$ 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。

小结　需要复习的知识点

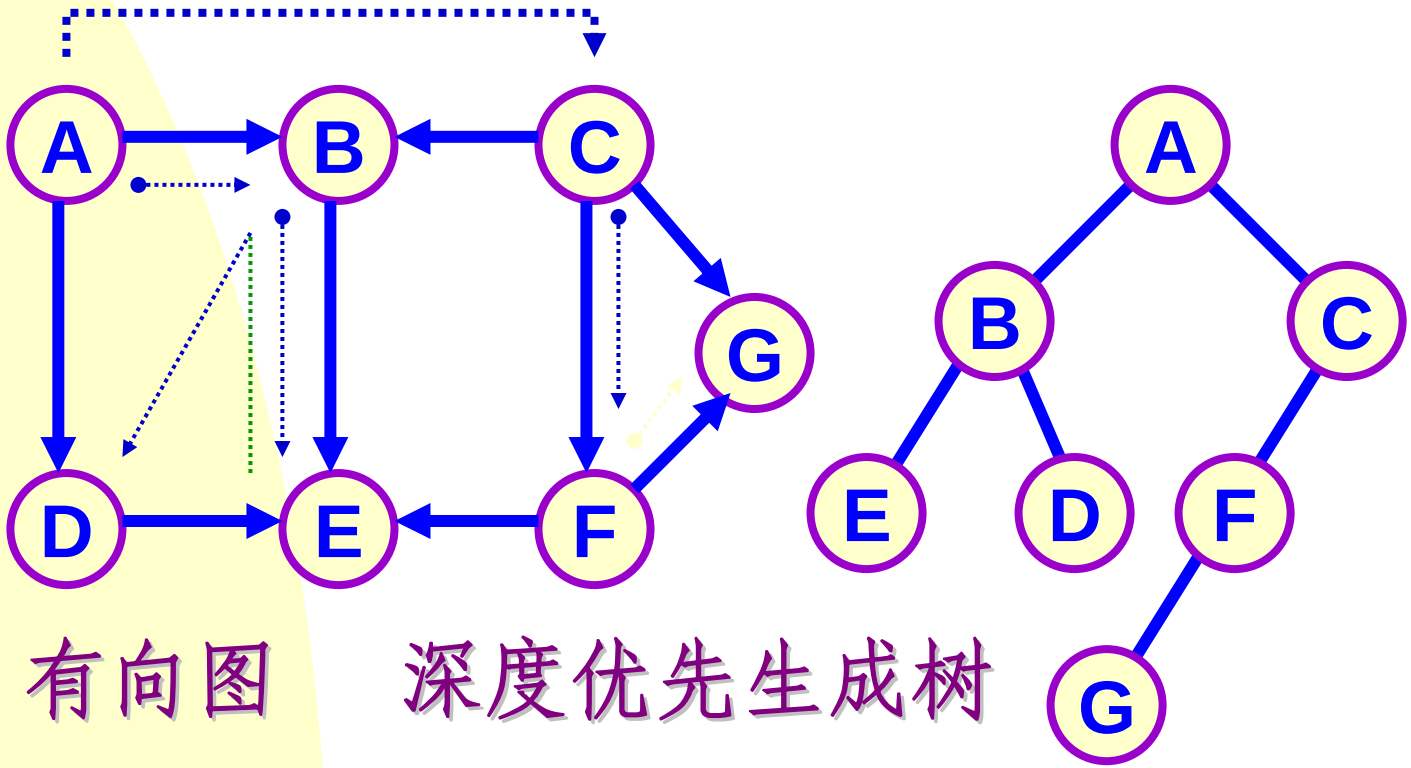
- 理解：图的基本概念
  - 图的顶点集合是非空有限集合
  - 图的边集合可以是空集合
  - 图的顶点序号不是固有的
- 掌握：图的存储表示
  - 图的邻接矩阵元素个数与顶点有关，非零元素个数与边有关
  - 图的邻接表既与顶点个数 n 有关，又与边数有关

- 连通无向图最多有 $n(n-1)/2$ 条边，最少有$n-1$条边。
- 强连通有向图最多有 $n(n-1)$ 条边，最少有 n 条边。$n=1$无边。
- 图中顶点的度有别于树中结点的度。

- 图的遍历与连通性
  - 深度优先搜索算法 递归或用栈
  - 广度优先搜索算法 用队列
  - 生成树或生成森林

- 连通分量
  - 重连通分量与关节点的概念
  - 求解关节点及构造重连通图的方法
- 掌握：构造最小生成树的方法
  - *Prim* 算法
  - *Kruskal* 算法
- 掌握：活动网络的拓扑排序算法
- 掌握：求解关键路径的方法
- 理解：求解最短路径的*Dijkstra*方法(不要求算法)

【例1】以深度优先搜索方法从 A 出发遍历图，建立深度优先生成森林。



有向图　　深度优先生成树

```
template<class Type>
void Graph<Type> ::
DFS_Forest ( Tree<Type> &T ) {
    TreeNode<Type> *rt, *subT;
    int *visited = new int[n];   //创建访问标志数组
    for ( int i = 0; i < n; i++ )
        visited [i] = 0;              //初始化, 都未访问过
    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 ( GetValue(i) );
                //顶点 i 的值成为根 rt 的值
```

```
        else
            subT = T.InsertRightSibling
                        ( subT, GetValue(i) );
                //顶点 i 的值成为 subT 右兄弟的值
            DFS_Tree ( T, subT, i, visited );
                //从顶点 i 出发深度优先遍历
                //建立以 subT为根的 T 的子树
    }
}

template<class Type>
void Graph<Type> :: DFS_Tree
( Tree<Type> &T,  TreeNode<Type>*RT,
      int i,  int visited [ ] ) {
```

```
TreeNode<Type> *p;
visited [i] = 1;        //顶点 i 作访问过标志
int w = First-Adjvertex(i);
//取顶点 i 的第一个邻接顶点 w
int FirstChild = 1;
// i 第一个未访问子女应是 i 的左子女
while ( w ) {           //邻接顶点 w 存在
      if ( ! visited [w] ) {
            // w未访问过, 将成为 i 的子女
          if ( FirstChild ) {
              p = T.InsertLeftChild ( RT,
                    GetValue(w) );
            // p 插入为 RT 的左子女
```

```
            FirstChild = 0;   //建右兄弟
        }
        else p = T.InsertRightSibling
                 ( p, GetValue(w) );
            // p 插入为 p 的左子女
        DFS_Tree ( T, p, w );
            //递归建立 w 的以 p 为根的子树
    }
    //邻接顶点 w 处理完
    w = Next-AdjVertex ( i, w );
    //取 i 的下一个邻接顶点 w
}
//回到 while 判邻接顶点 w 存在
}
```