第八章 图

- 图的基本概念
- 图的存储结构
- 图的遍历
- 最小生成树
- 最短路径
- 用顶点表示活动的网络（AOV网络）
- 用边表示活动的网络（AOE网络）
- 本章小结

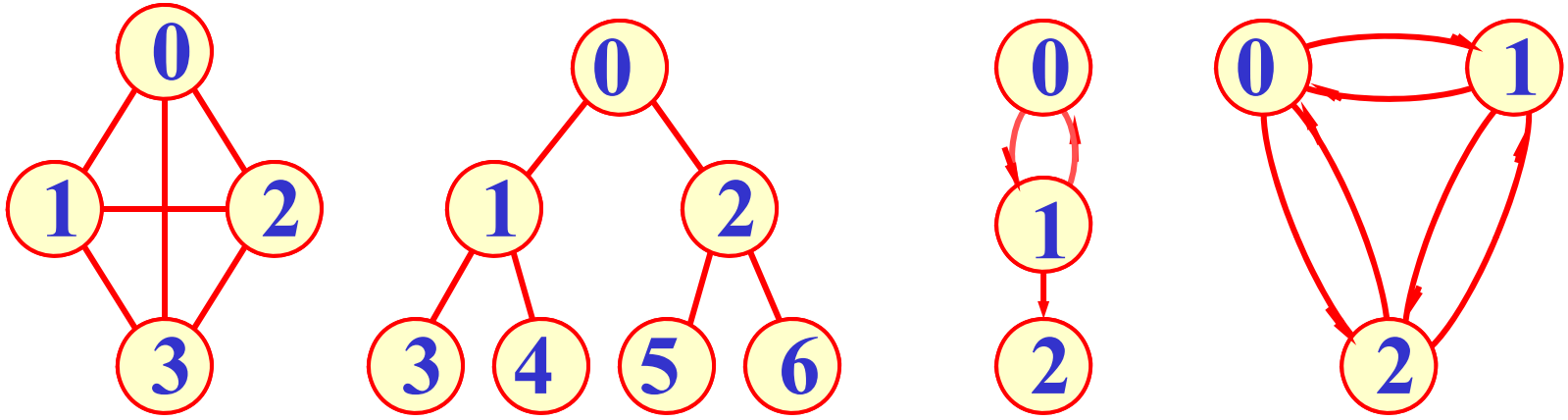
8.1 图的基本概念

- **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：

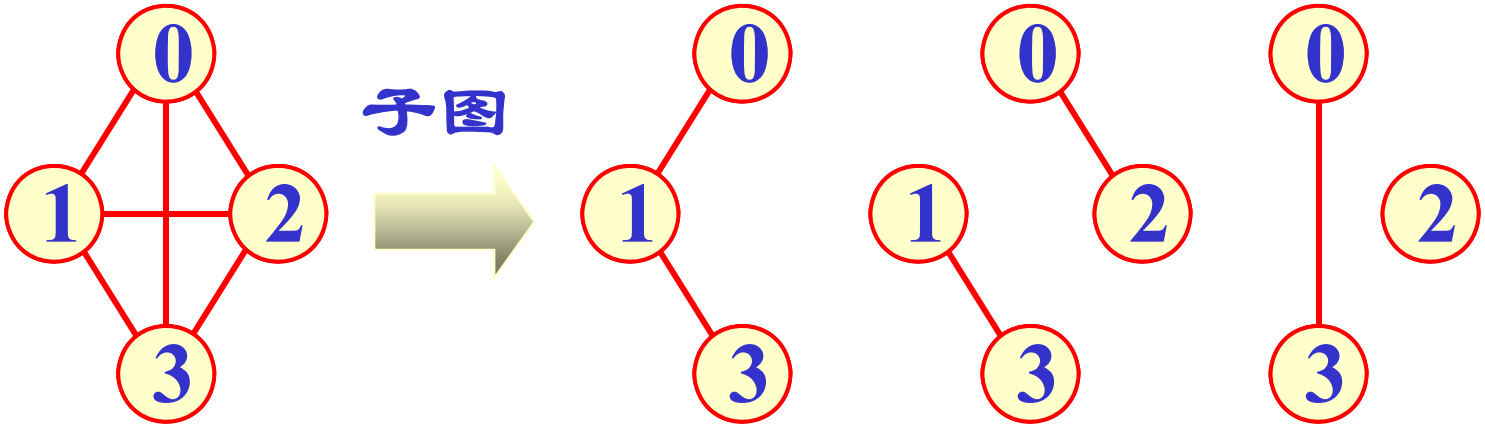
  Graph = (V, E)

  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。



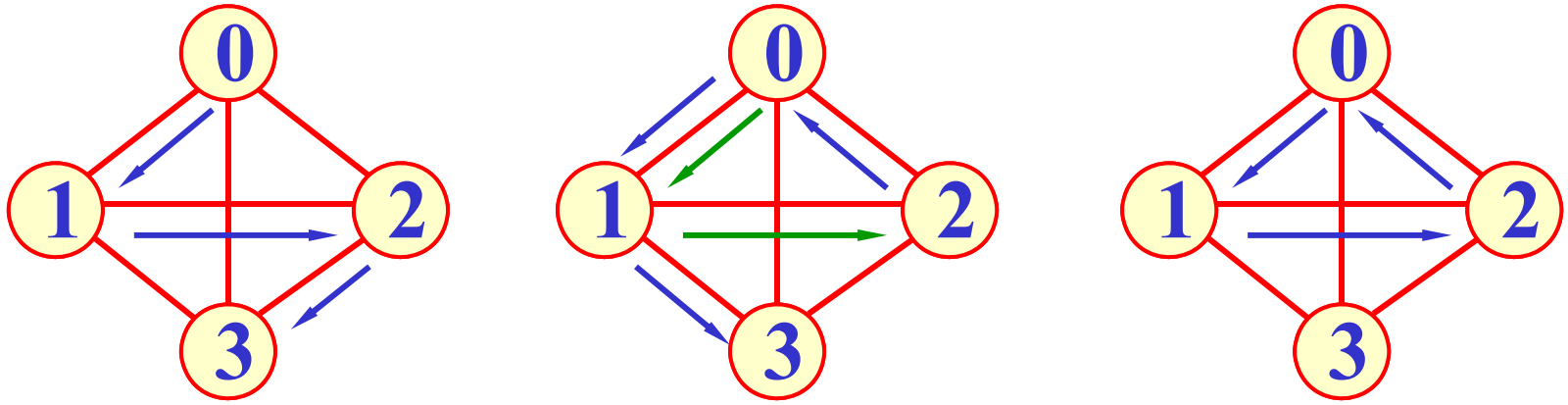
- **邻接顶点** 如果 (u, v) 是 E(G) 中的一条边，则称 u 与 v 互为邻接顶点。
- **子图** 设有两个图 G = (V, E) 和 G' = (V', E')。若 V'⊆V 且 E'⊆E，则称图 G' 是图 G 的子图。



- **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。

- **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 $v_1, v_2, \cdots, v_m$ 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。
- **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其极小连通子图，在 n 个顶点的情形下，有 n-1 条边。

图的抽象数据类型

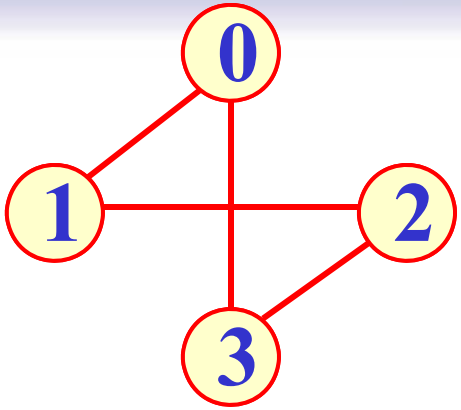
```
template <class Type> class Graph {
public:
    Graph ( );
    void InsertVertex ( Type &vertex );
    void InsertEdge ( int v1, int v2, int weight );
    void RemoveVertex ( int v );
    void RemoveEdge ( int v1, int v2 );
    bool IsEmpty ( );
    Type GetWeight ( int v1, int v2 );
    int GetFirstNeighbor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

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



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

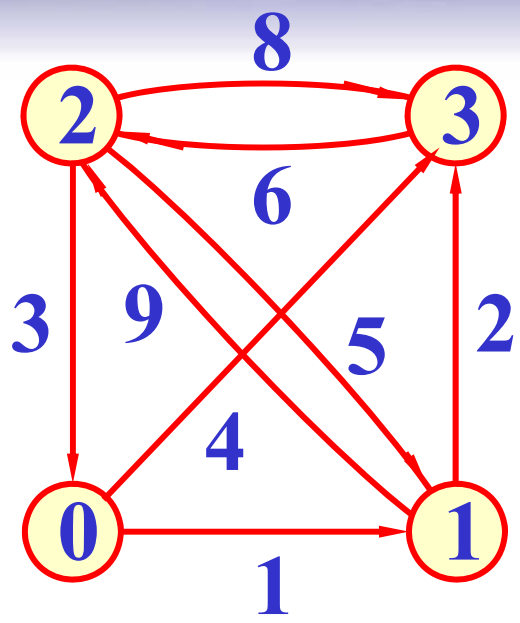
$$A.edge = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}$$

- 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

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



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

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

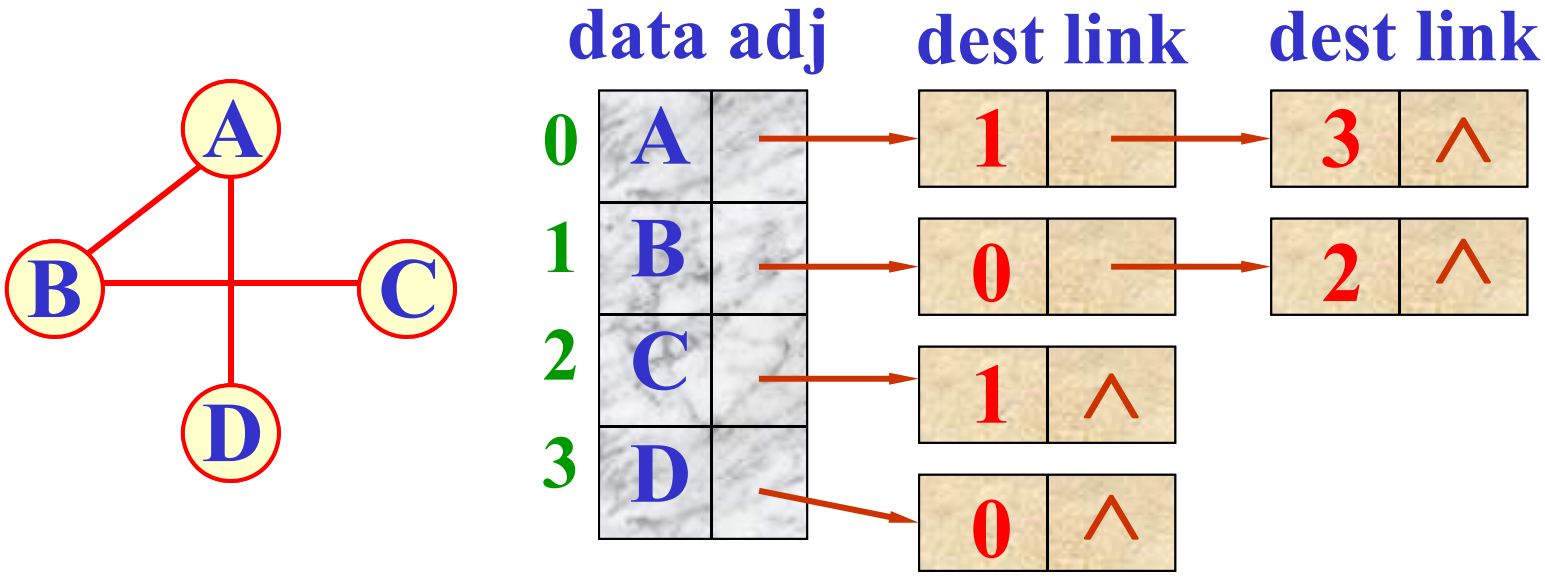
```
template <class Type> Graph <Type> ::
Graph ( int sz ) {  //构造函数
    for ( int i = 0; i < sz; i++ )
        for ( int j = 0; j < sz; j++ )
            Edge[i][j] = 0;
    NumberEdges = 0;
}
```

```
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;
}
```

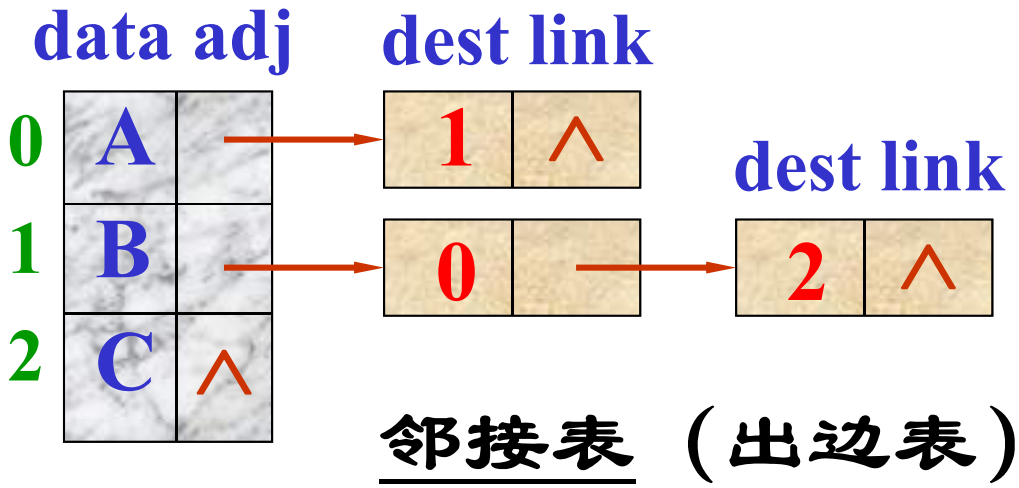
邻接表 (Adjacency List)

- **无向图的邻接表**

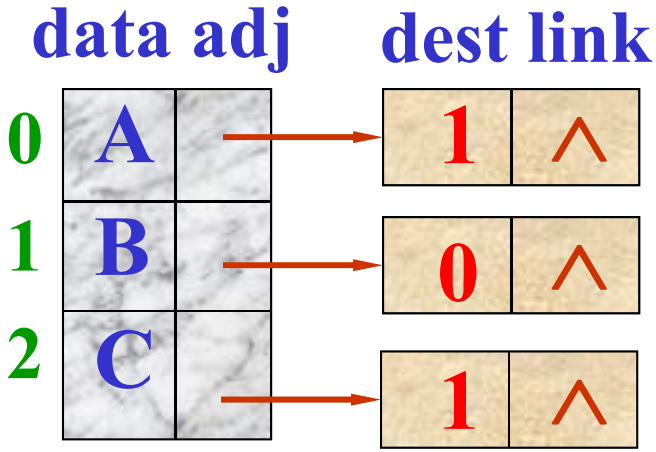


同一个顶点发出的边链接在同一个边链表中，每一个链结点代表一条边（边结点），结点中有另一顶点的下标 dest 和指针 link。

有向图的邻接表和逆邻接表

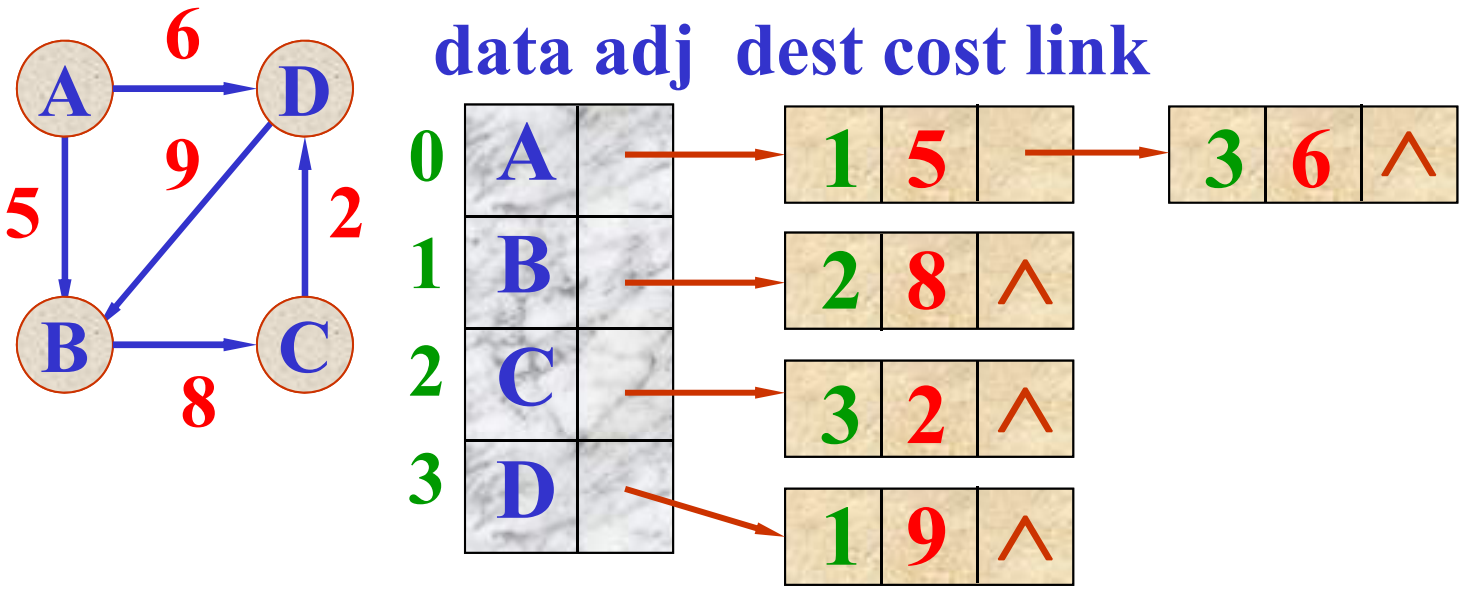


邻接表（出边表）



逆邻接表（入边表）

网络（带权图）的邻接表



（顶点表）（出边表）

- 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

```
#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

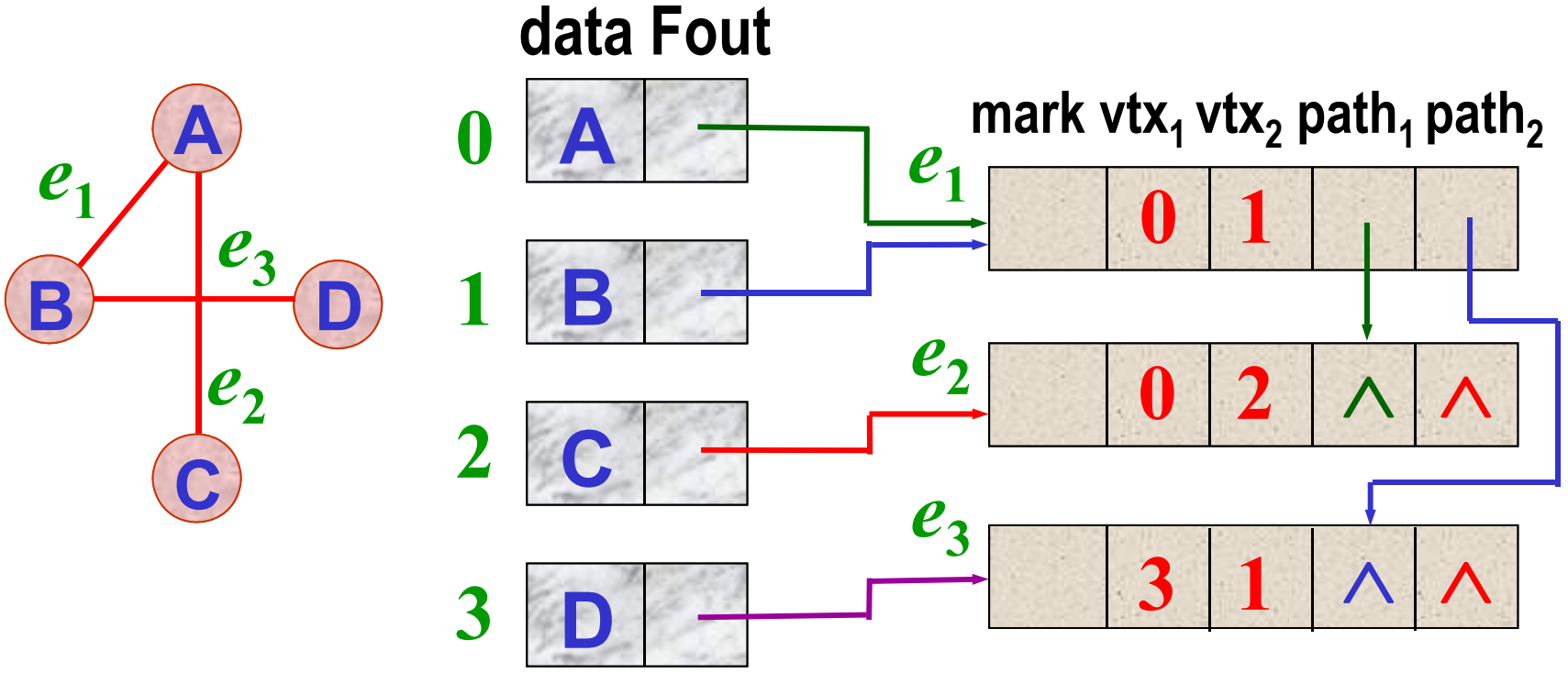
mark	vertex1	vertex2	path1	path2

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**

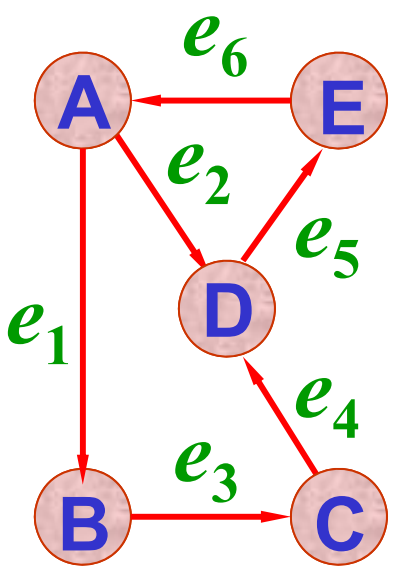
mark	vertex1	vertex2	path1	path2

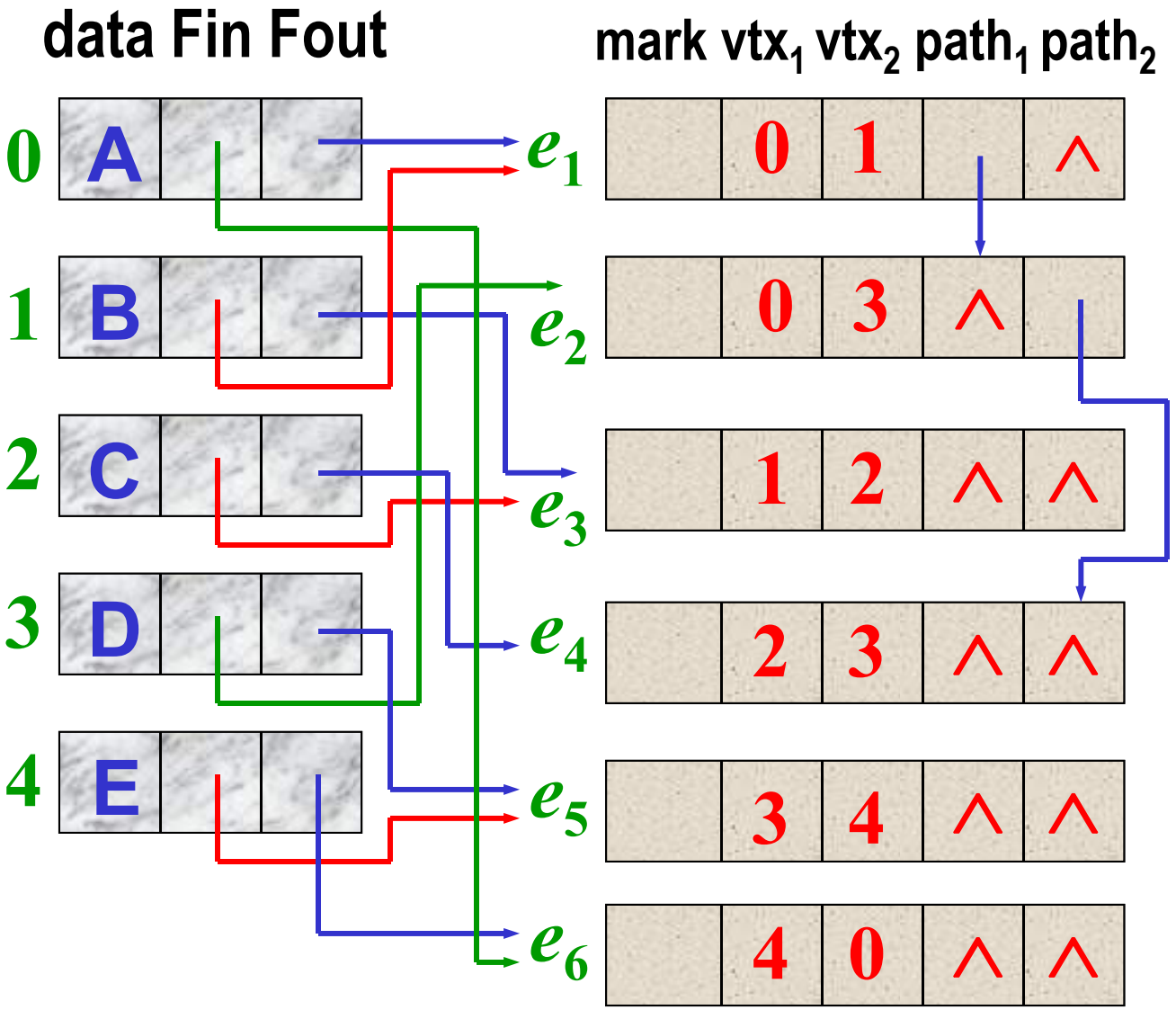
其中，mark 是处理标记；vertex1 和 vertex2 指明该有向边始顶点和终顶点的位置；path1 是指向始顶点与该边相同的下一条边的指针；path2 是指向终顶点与该边相同的下一条边的指针；需要时还可有权值域 cost 。

顶点结点的结构

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 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。


A
E
D
B
C
e_1
e_2
e_3
e_4
e_5
e_6


data Fin Fout
mark vtx1 vtx2 path1 path2
0 A
1 B
2 C
3 D
4 E
e_1
e_2
e_3
e_4
e_5
e_6
0 1 ∧
0 3 ∧
1 2 ∧ ∧
2 3 ∧ ∧
3 4 ∧ ∧
4 0 ∧ ∧

8.3 图的遍历

- 从已给的连通图中某一顶点出发，沿着一些边访遍图中所有的顶点，且使每个顶点仅被访问一次，就叫做图的遍历 (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[ ] 。

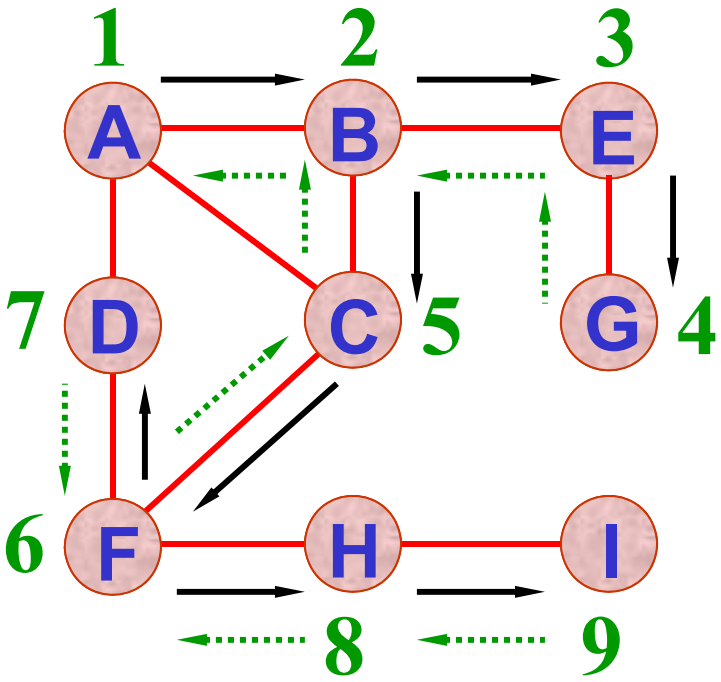
- 辅助数组 visited[ ] 的初始状态为 0，在图的遍历过程中，一旦某一个顶点 i 被访问，就立即让 visited[i] 为1，防止它被多次访问。
- 图的遍历的分类
  - 深度优先搜索

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

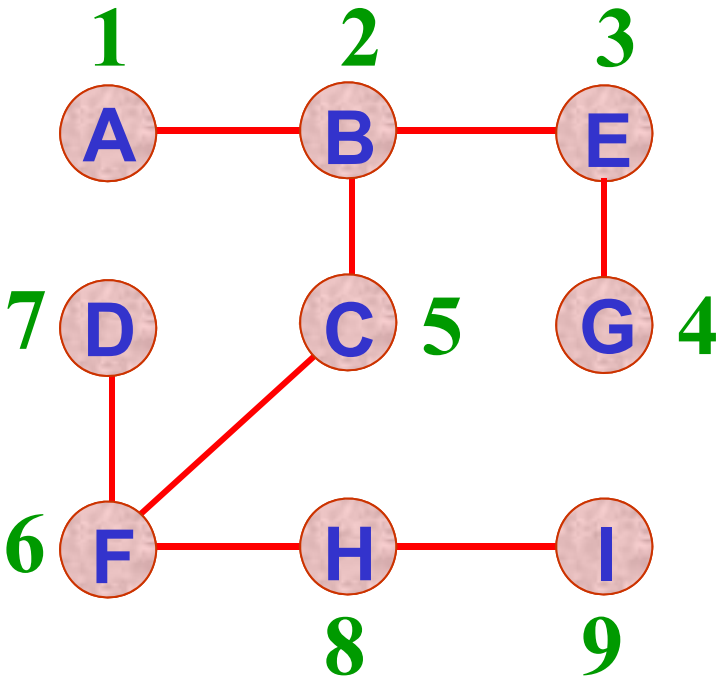
    BFS (Breadth First Search)

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。

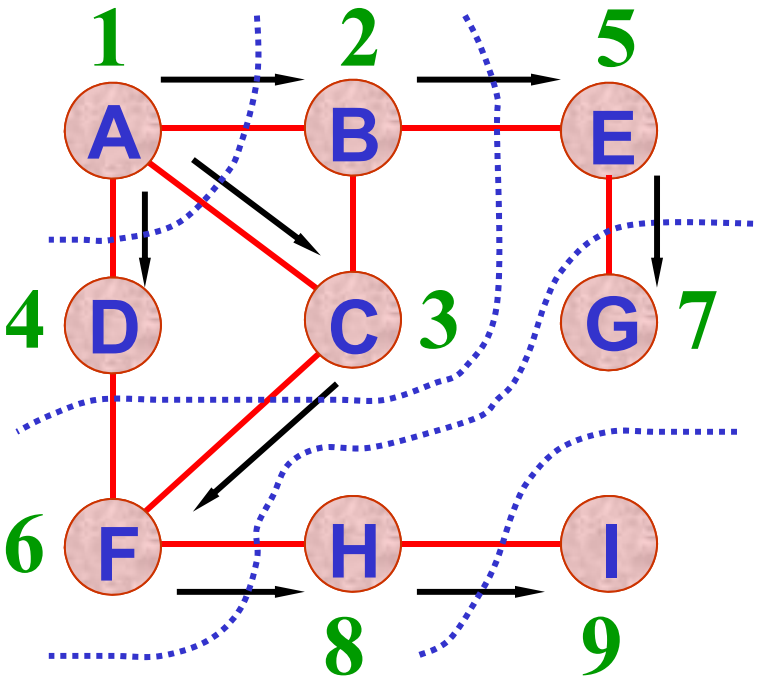
图的深度优先搜索算法

```
template <class Type> void Graph <Type> ::
DFS ( Graph <Type> &G ) {
    int n = G.NumberOfVertices ( );
    static int *visited = new int [n];
    for ( int i = 0; i < n; i++ )  //访问数组
        visited [i] = 0;  //visited 初始化
    DFS ( 0, visited );  //从顶点 0 开始深度优先搜索
    delete [ ] visited;  //释放 visited
}
```

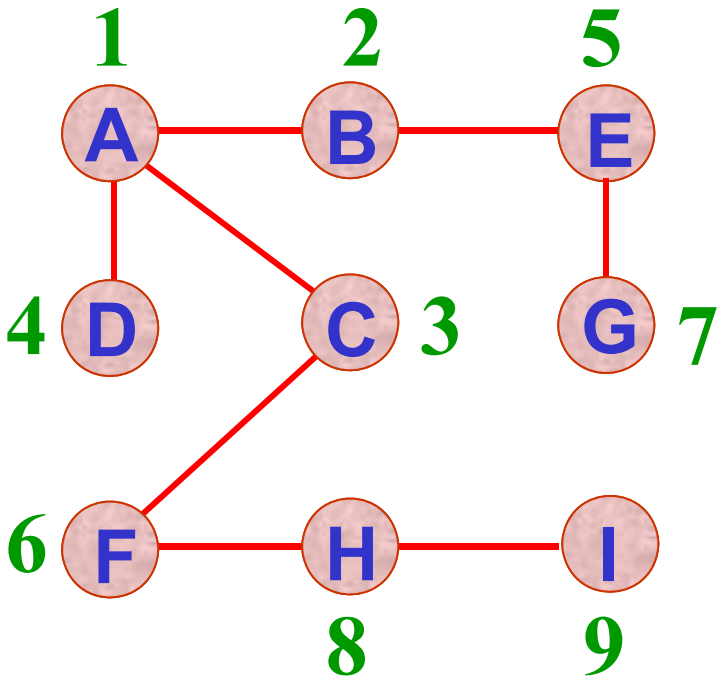
```
template <class Type> void Graph <Type> ::
DFS ( int v, int visited [ ] ) {
    cout << G.GetValue ( v ) << ' '; //访问顶点 v
    visited[v] = 1; //顶点 v 作访问标记
    int w = G.GetFirstNeighbor ( v );
    while ( w != -1 ) { //若邻接顶点 w 存在
        if ( !visited[w] ) DFS ( w, visited );
        //若顶点 w 未访问过，递归访问顶点 w
        w = G.GetNextNeighbor ( v, w );
        //取顶点 v 排在 w 后的下一个邻接顶点
    }
}
```

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。

图的广度优先搜索算法

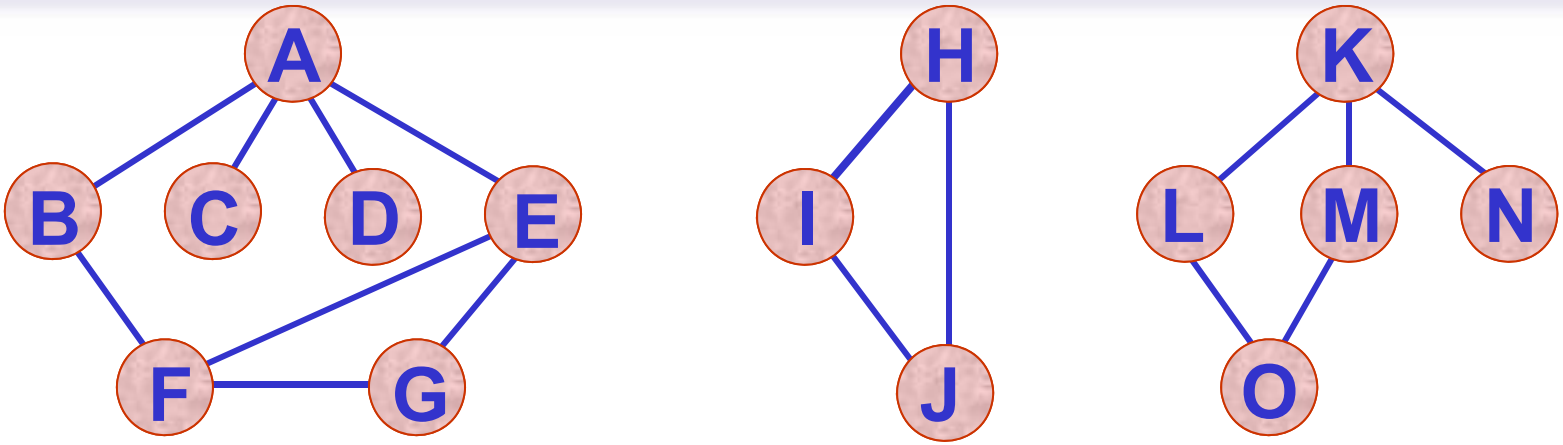
```
template <class Type> void Graph <Type> ::
BFS ( int v ) {
    int n = G.NumberOfVertices ( );
    int *visited = new int[n];
    for ( int i = 0; i < n; i++ ) visited[i] = 0;
    cout << G.GetValue ( v ) << ' ';   visited[v] = 1;
    Queue <int> q;    q.EnQueue ( v );  //进队列
```

```
while ( !q.IsEmpty ( ) ) {  //队空搜索结束
    q.GetFront (v);  q.DeQueue ( );
    int w = G.GetFirstNeighbor ( v );
    while ( w != -1 ) {  //若邻接顶点 w 存在
        if ( !visited[w] ) {  //未访问过
            cout << G.GetValue ( w ) << ' ';
            visited[w] = 1;   q.EnQueue ( w );    }
        w = G.GetNextNeighbor ( v, w );
    } //重复检测 v 的所有邻接顶点
} //外层循环，判队列空否
delete [ ] visited;
}
```

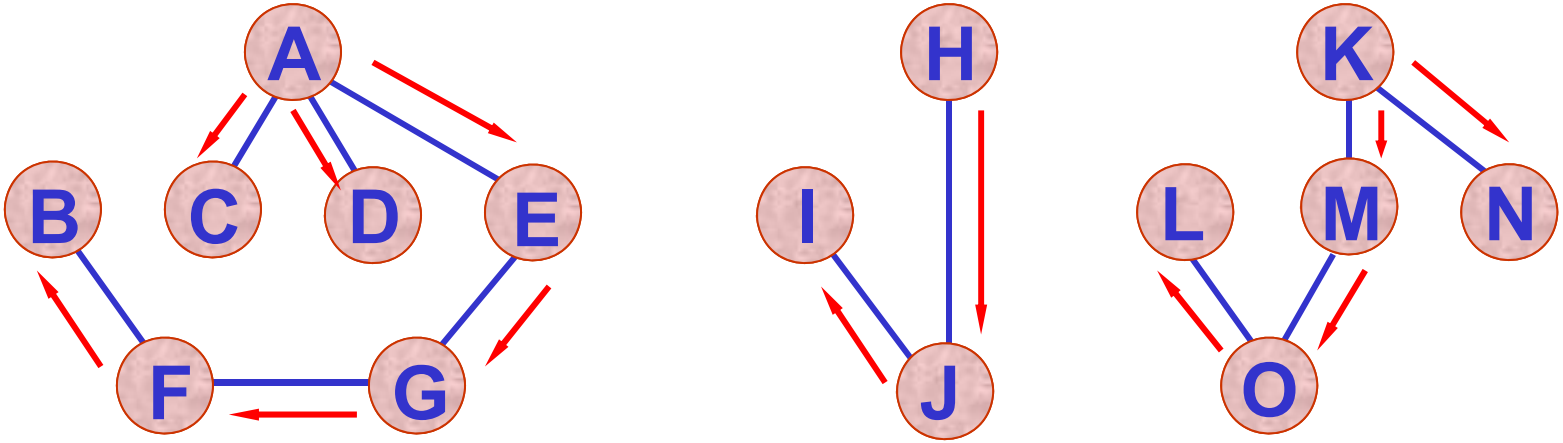
连通分量 (Connected component)

- 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

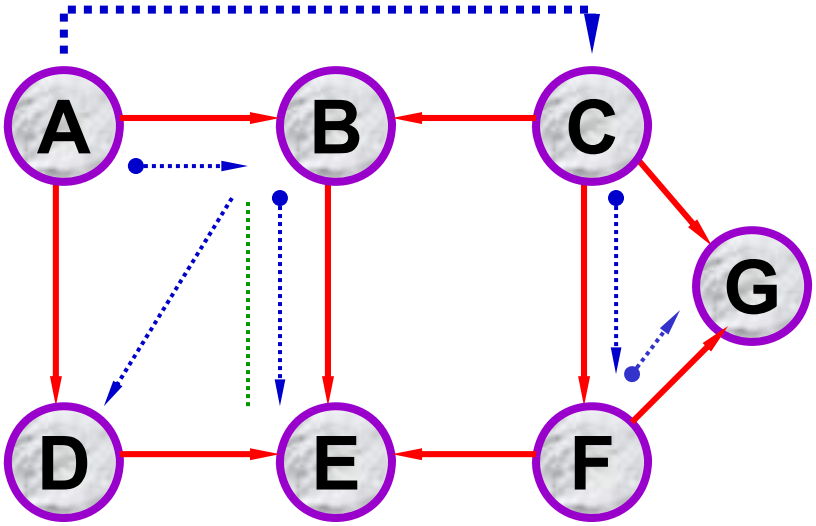


非连通图的连通分量

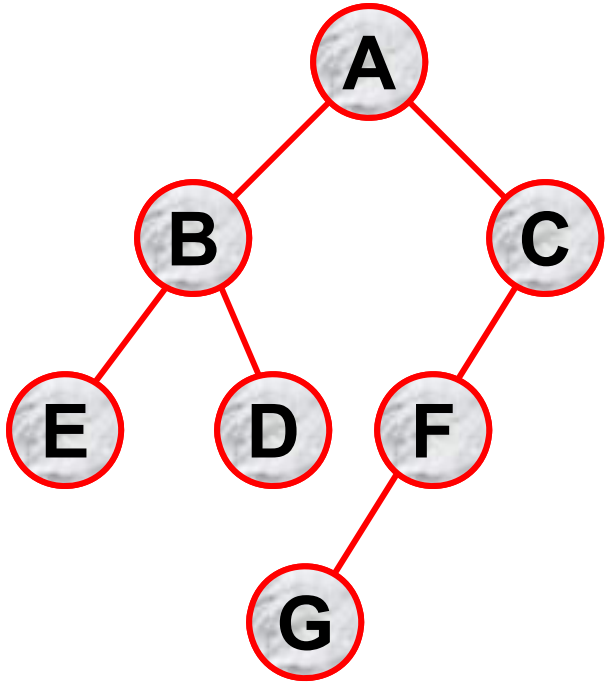
确定连通分量的算法

```
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。

克鲁斯卡尔 (Kruskal) 算法

- **克鲁斯卡尔算法的基本思想**

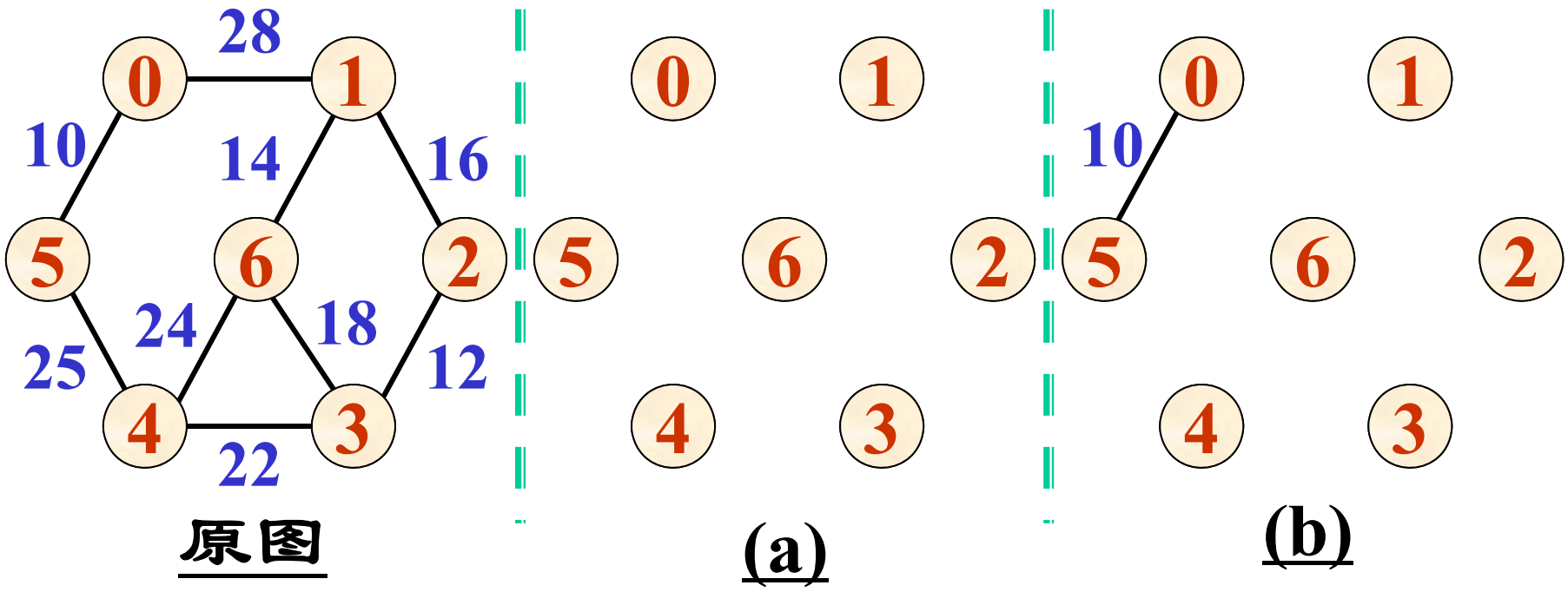
  设一个有 n 个顶点的连通网络 N = { V, E }，最初先构造一个只有 n 个顶点，没有边的非连通图 T = { V, ∅ }，图中每个顶点自成一个连通分量。当在 E 中选到一条具有最小权值的边时，**若该边的两个顶点落在不同的连通分量上，则将此边加入到 T 中**；否则将此边舍去，重新选择一条权值最小的边。如此重复下去，直到所有顶点在同一个连通分量上为止。

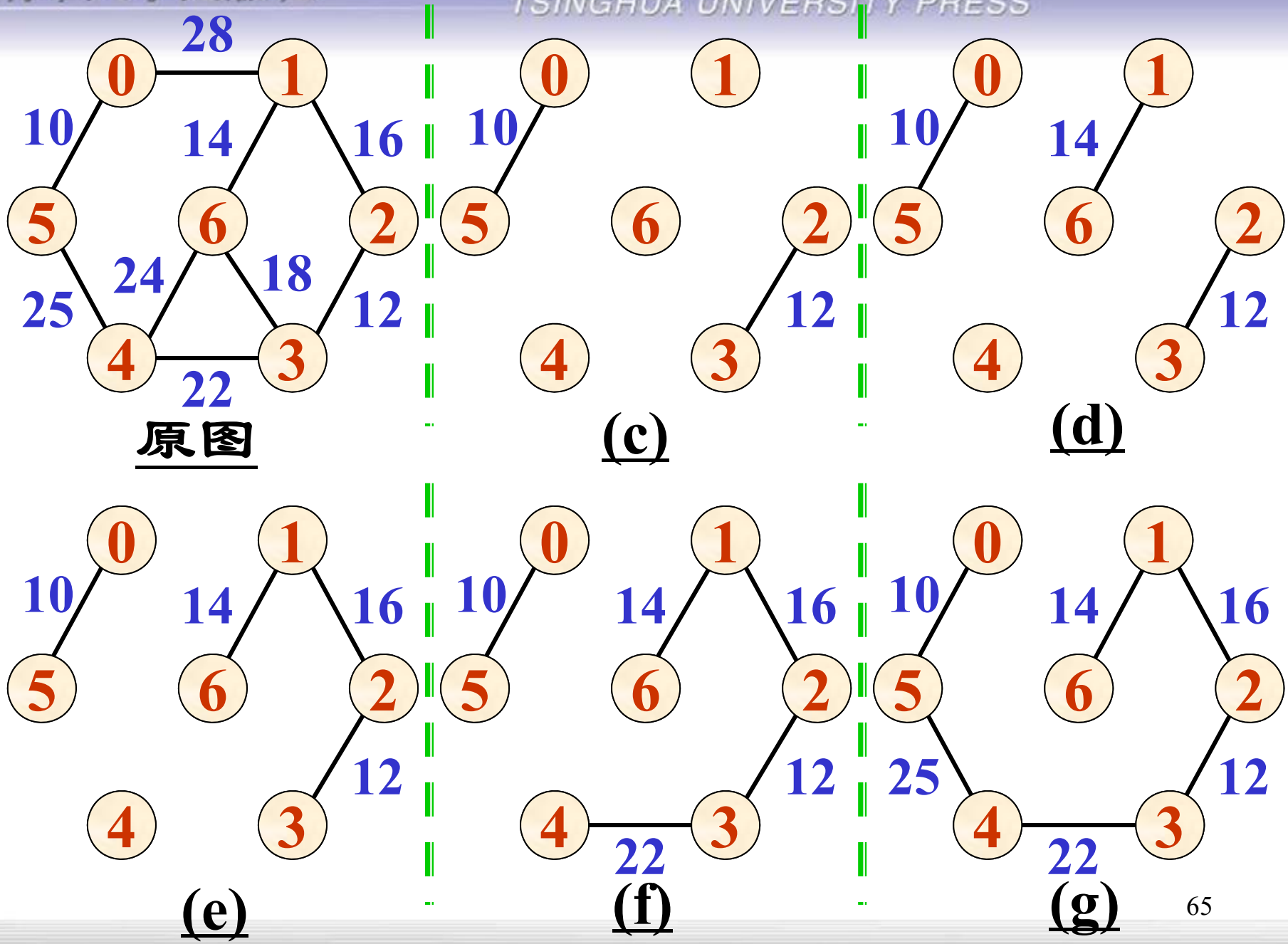
- **算法的框架**　　利用**最小堆**(MinHeap)和**并查集**(DisjointSets)来实现**克鲁斯卡尔算法**。
- 首先，利用**最小堆**来存放 E 中所有的边，堆中每个结点的格式为

- 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




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

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

H	0	0, 1, 28							
建堆	1	0, 5, 10	0, 1, 28						
	2	0, 5, 10	0, 1, 28	1, 2, 16					
	3	0, 5, 10	1, 6, 14	1, 2, 16	0, 1, 28				
	4	0, 5, 10	2, 3, 12	1, 2, 16	0, 1, 28	1, 6, 14			
	5	0, 5, 10	2, 3, 12	1, 2, 16	0, 1, 28	1, 6, 14	3, 4, 22		
	6	0, 5, 10	2, 3, 12	1, 2, 16	0, 1, 28	1, 6, 14	3, 4, 22	3, 6, 18	
	7	0, 5, 10	2, 3, 12	1, 2, 16	4, 5, 25	1, 6, 14	3, 4, 22	3, 6, 18	0, 1, 28
	8	0, 5, 10	2, 3, 12	1, 2, 16	4, 6, 24	1, 6, 14	3, 4, 22	3, 6, 18	0, 1, 28	4, 5, 25
取边		0, 5, 10	2, 3, 12	1, 2, 16	4, 6, 24	1, 6, 14	3, 4, 22	3, 6, 18	0, 1, 28	4, 5, 25
			2, 3, 12	1, 6, 14	1, 2, 16	4, 6, 24	4, 5, 25	3, 4, 22	3, 6, 18	0, 1, 28
				1, 6, 14	4, 6, 24	1, 2, 16	0, 1, 28	4, 5, 25	3, 4, 22	3, 6, 18
					1, 2, 16	4, 6, 24	3, 6, 18	0, 1, 28	4, 5, 25	3, 4, 22
						3, 6, 18	4, 6, 24	3, 4, 22	0, 1, 28	4, 5, 25
							3, 4, 22	4, 6, 24	4, 5, 25	0, 1, 28
								4, 6, 24	0, 1, 28	4, 5, 25
									4, 5, 25	0, 1, 28
										0, 1, 28

（b）最小堆

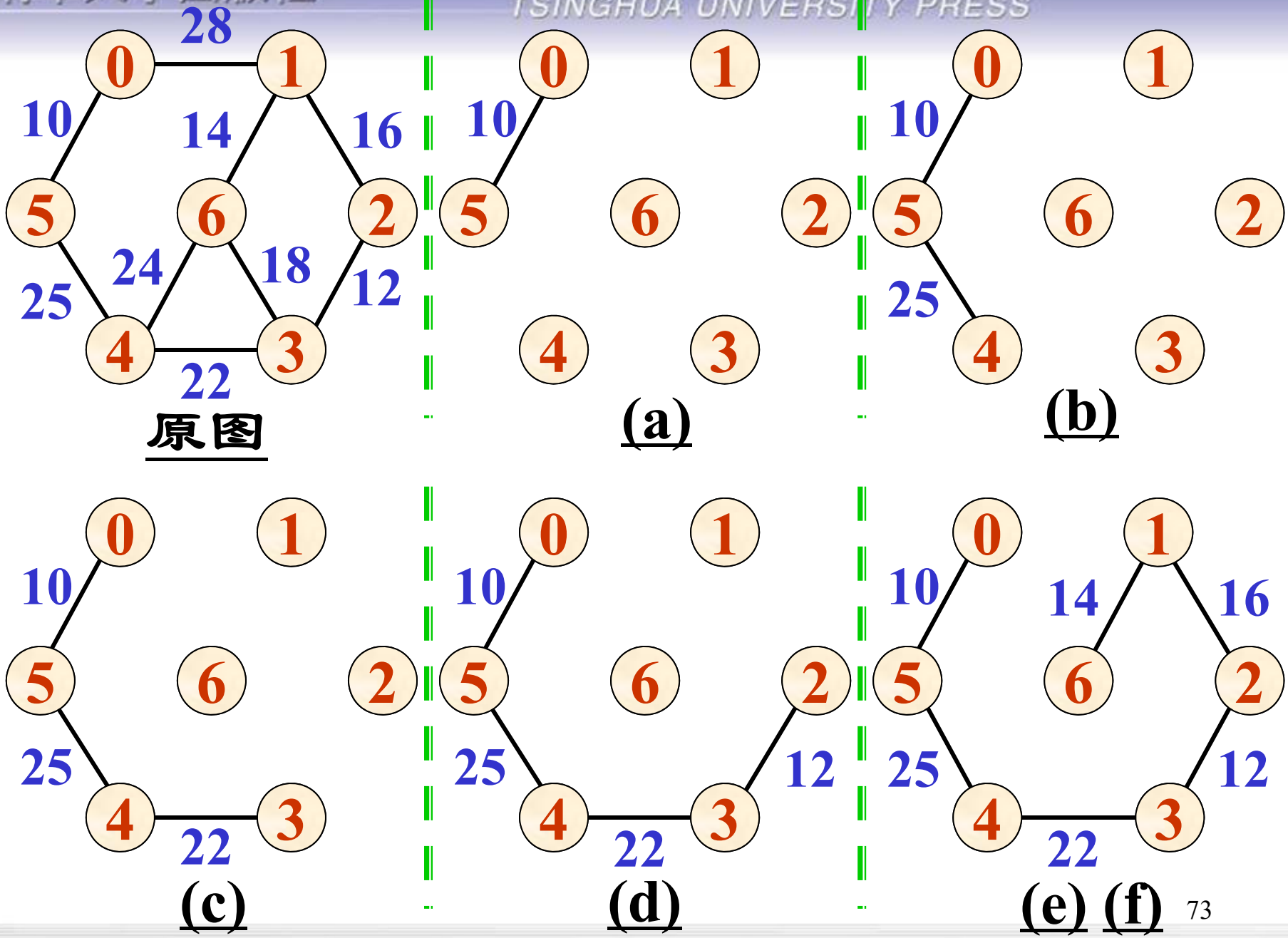
普里姆(Prim)算法

- **普里姆算法的基本思想**

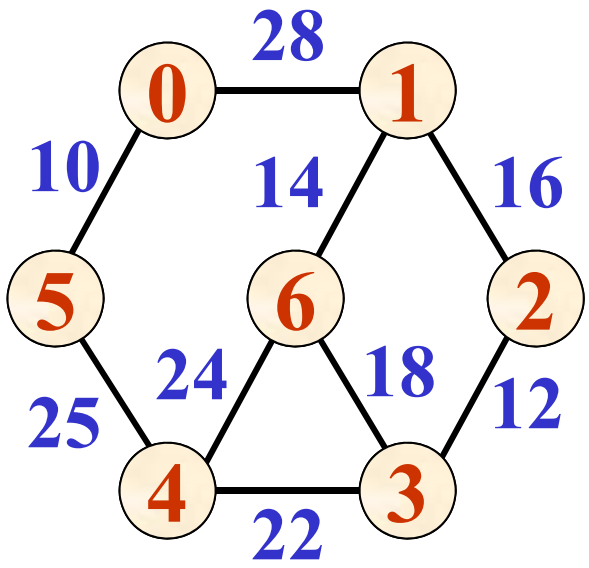
  从连通网络 N = { V, E } 中的某一顶点 u_0 出发，选择与它关联的具有最小权值的边 (u_0, v)，将其顶点加入到生成树顶点集合U中。

  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。**


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

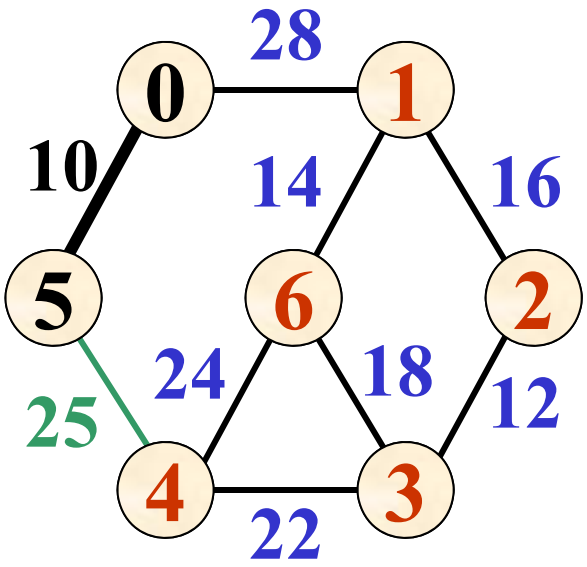
	0	1	2	3	4	5	6
lowcost	0	28	∞	∞	∞	10	∞
nearvex	-1	0	0	0	0	0	0

选 v=5 ↑ 选边 (0, 5)

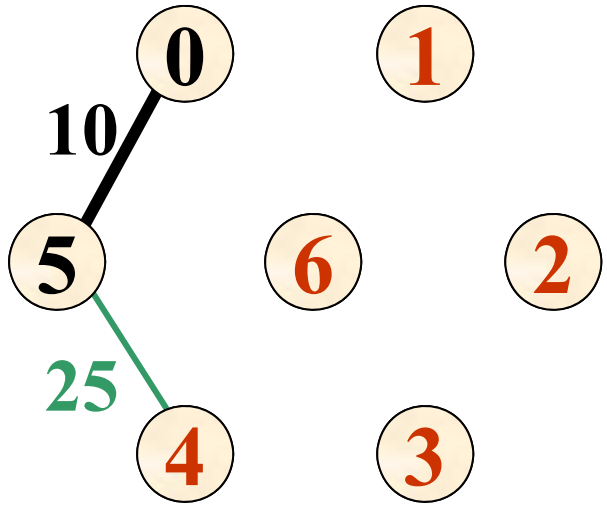
顶点 v=5 加入生成树顶点集合：

	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

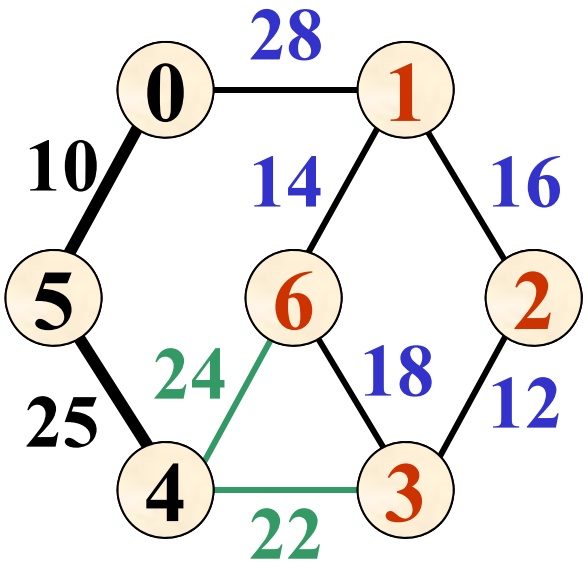


边 (0, 5, 10) 加入生成树

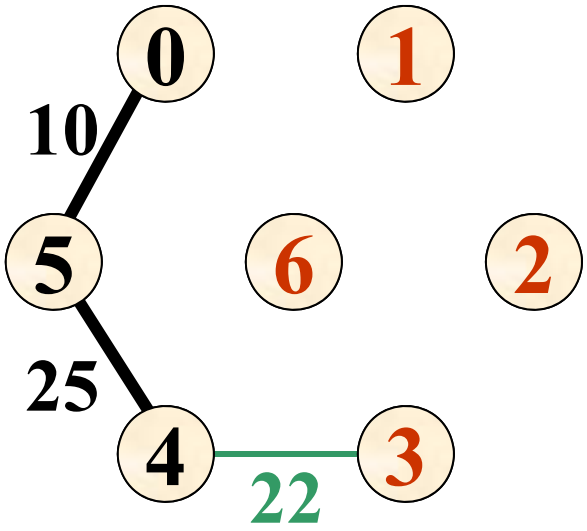
顶点 v=4 加入生成树顶点集合：

	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

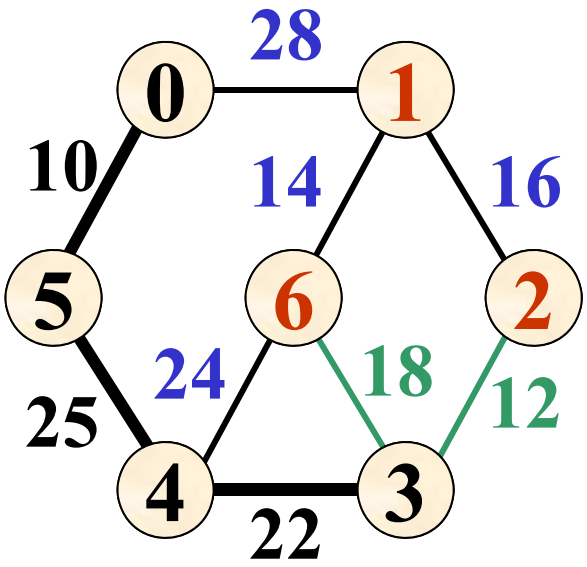


边 (5, 4, 25) 加入生成树

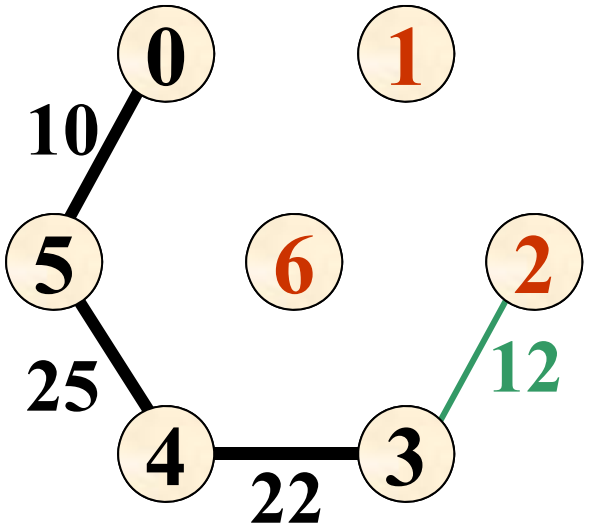
顶点 v=3 加入生成树顶点集合：

	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

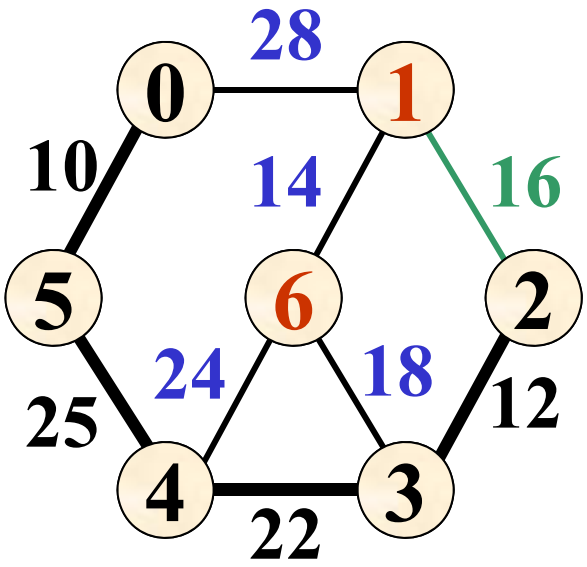


边 (4, 3, 22) 加入生成树

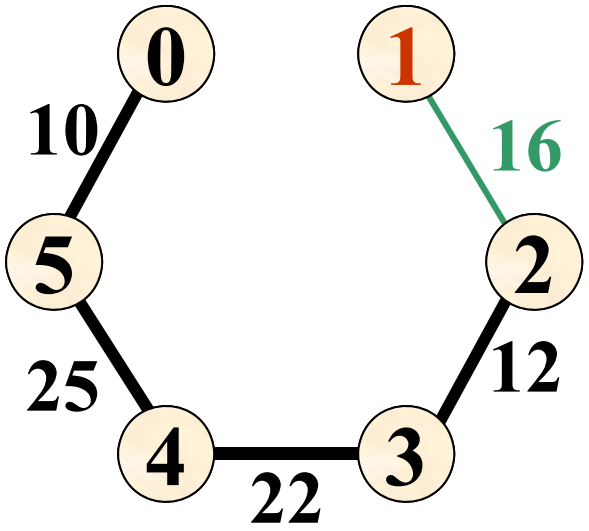
顶点 v=2 加入生成树顶点集合：

	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

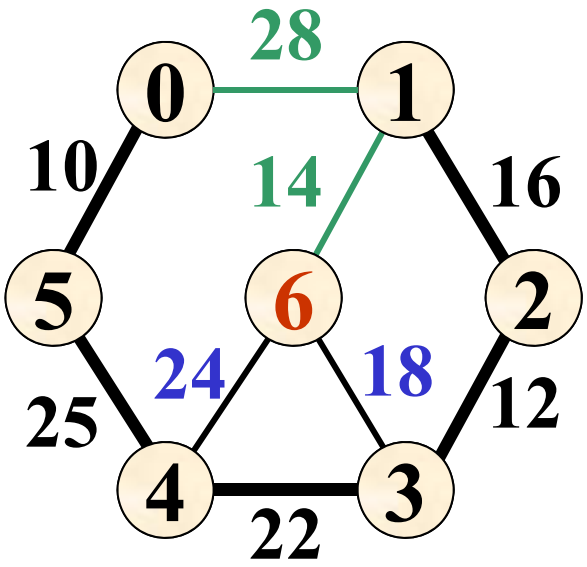


边 (3, 2, 12) 加入生成树

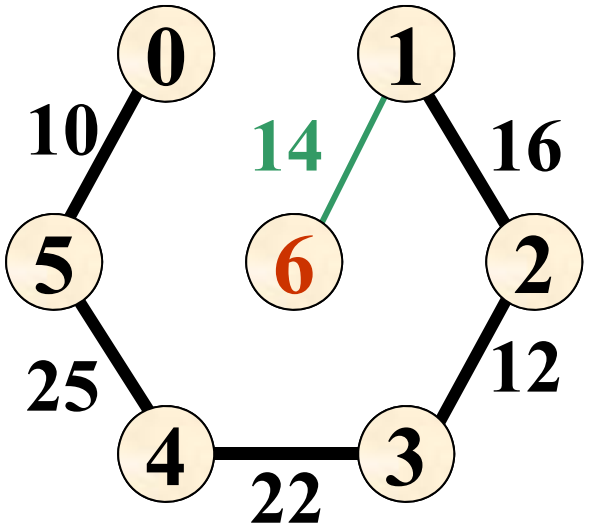
顶点 v=1 加入生成树顶点集合：

	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)



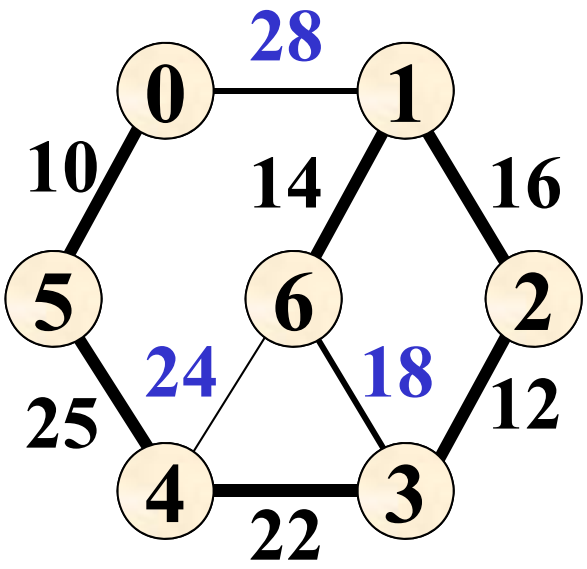
原图



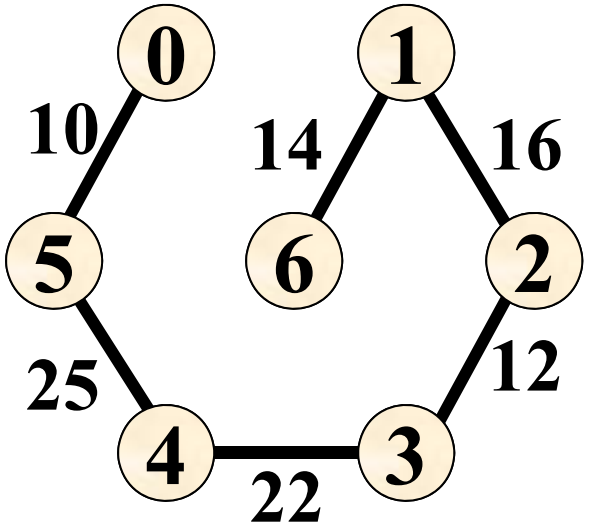
边 (2, 1, 16) 加入生成树

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
}
```

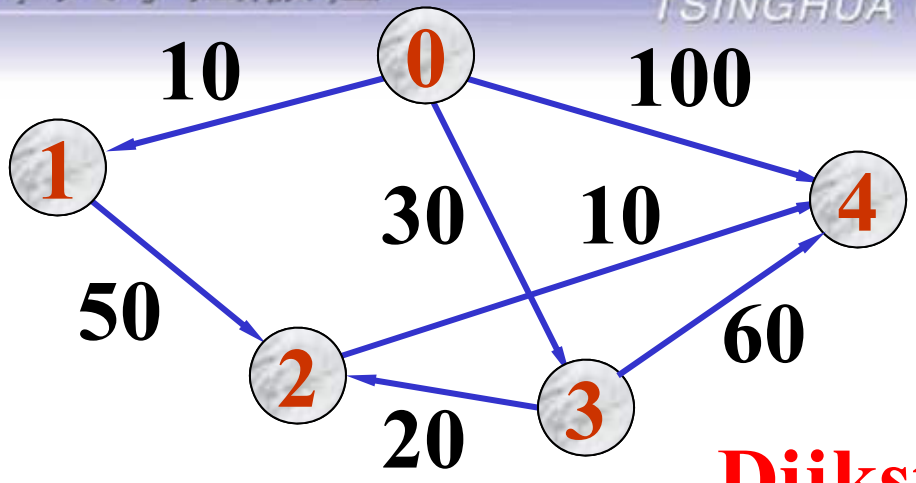
- 分析以上算法，设连通网络有 n 个顶点，则该算法的时间复杂度为 $O(n^2)$，它适用于边稠密的网络。
- 克鲁斯卡尔算法不仅适合于边稠密的情形，也适合于边稀疏的情形。
- 注意：当各边有相同权值时，由于选择的随意性，产生的生成树可能不惟一。

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。



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

- 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： $S \leftarrow \{ v_0 \}$;

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 ← S U { 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。

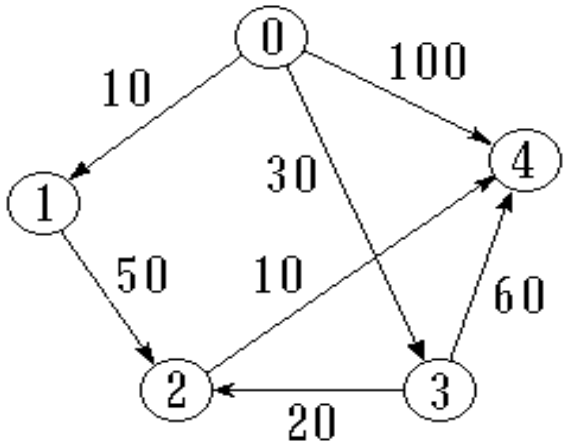
计算从单个顶点到其它各顶点最短路径

```
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



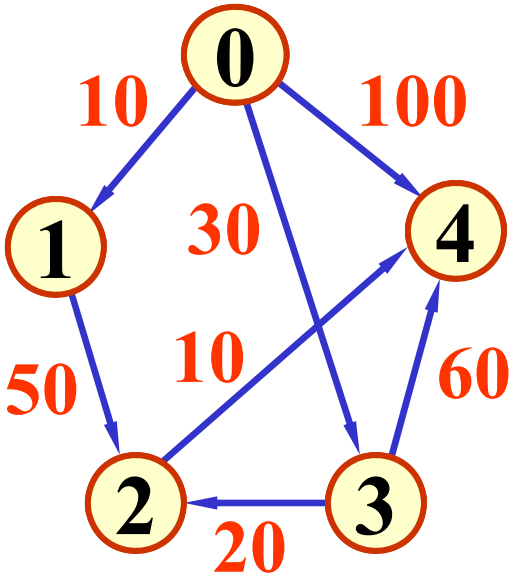
（a）带权有向图

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

（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

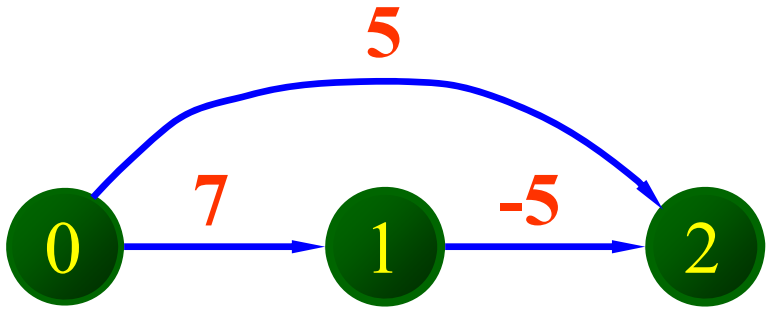
path[4] = 2 → path[2] = 3 →

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

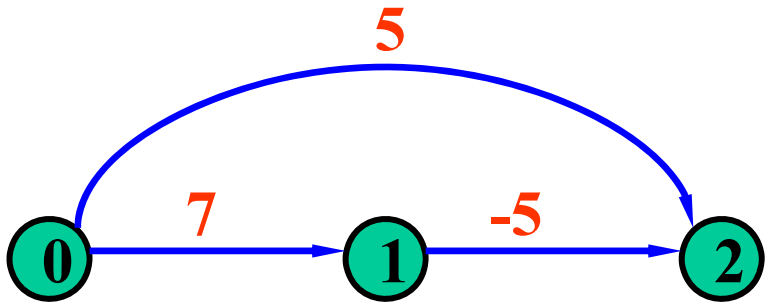
- 带权有向图 *D* 的某几条边或所有边的长度可能为负值，利用 Dijkstra 算法，不一定能得到正确的结果。

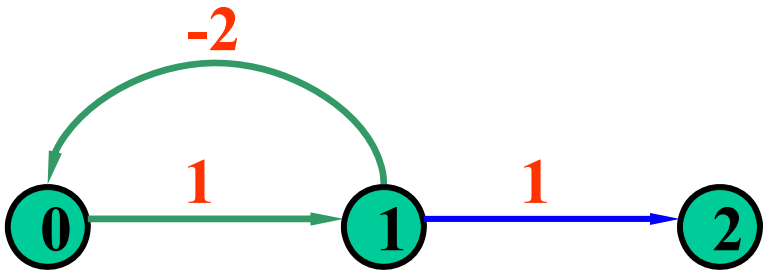
若设源点 v = 0 ，使用 Dijkstra 算法所得结果。



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$dist^1[u], dist^2[u], \ldots, dist^{n-1}[u]$。

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 $dist^{n-1}[u]$。

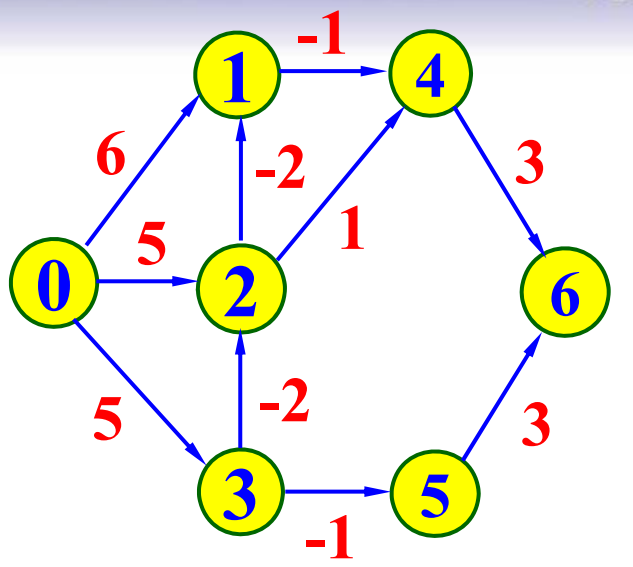
- 可以用递推方式计算 $dist^k[u]$ 。

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

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

  - 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。

所有各对顶点之间的最短路径

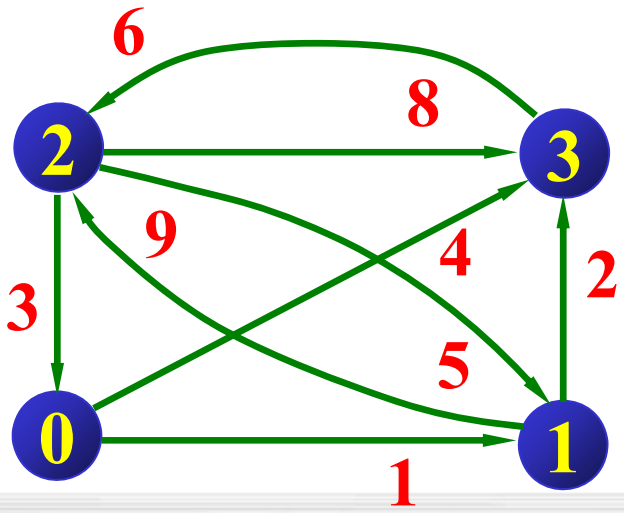
```
void AllLengths ( Graph <float> &G,
                  float a[ ][ ], int path[ ][ ] ) {
// a[i][j] 是顶点 i 和 j 之间的最短路径长度
// path[i][j] 是相应路径上顶点 j 的前一顶点的顶点号
   int i, j, k, n = G.NumberOfVertices ( );
   for ( i = 0; i < n; i++ )  //矩阵 a 与 path 初始化
      for ( j = 0; j < n; j++ ) {
         a[i][j]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

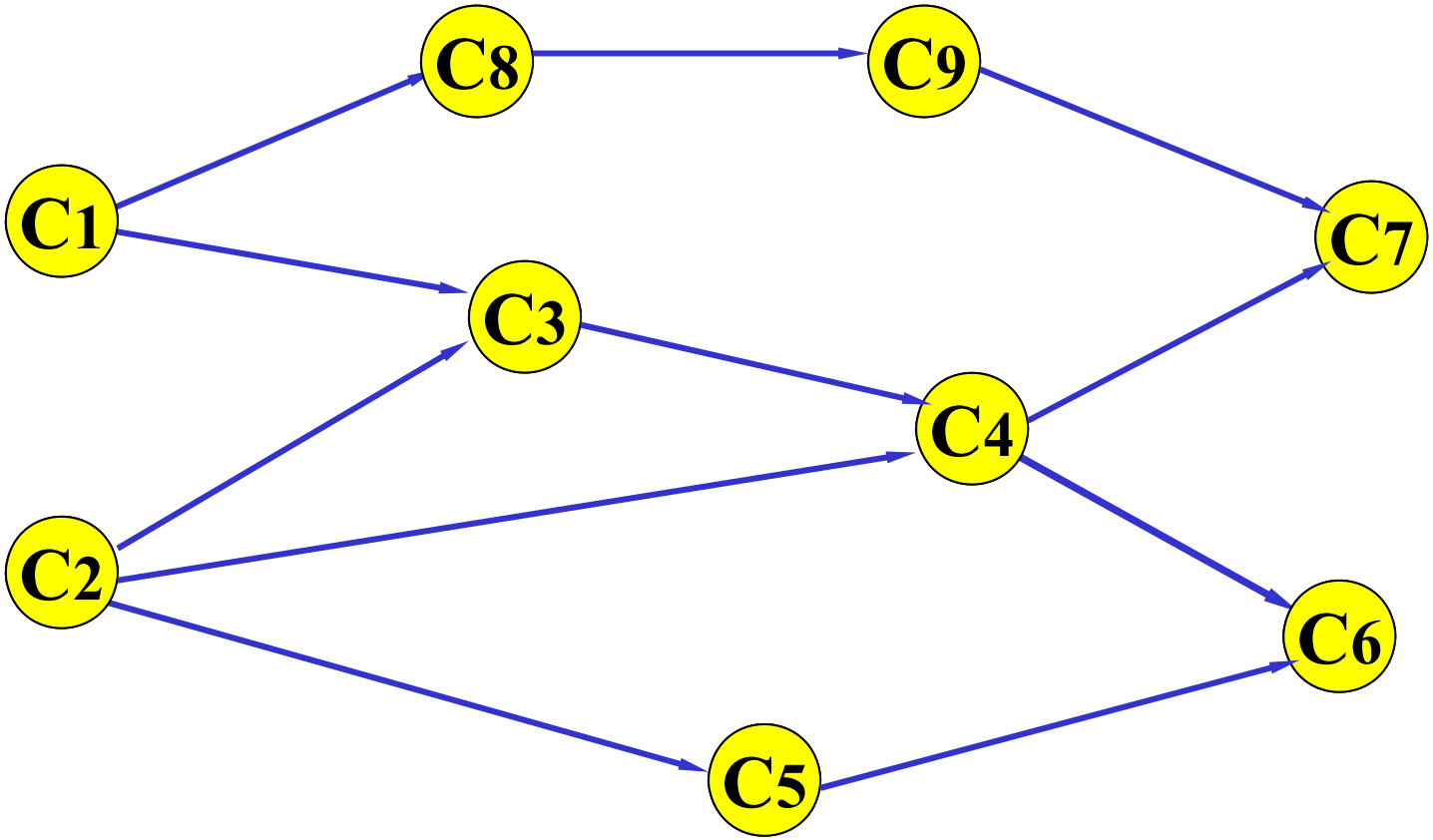


- 以 $Path^{(3)}$ 为例，对最短路径读法加以说明。从 $A^{(3)}$ 知，顶点 1 到 0 的最短路径长度为 $a[1][0] = 11$，其最短路径看 $path[1][0] = 2$, $path[1][2] = 3$, $path[1][3] = 1$，表示顶点 0 ← 顶点2 ← 顶点3 ← 顶点1；从顶点 1 到顶点 0 最短路径为<1, 3>, <3, 2>, <2, 0>。
- 本章给出的求解最短路径的算法不仅适用于带权有向图，对带权无向图也可以适用。因为带权无向图可以看作是有往返二重边的有向图。

8.6 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。

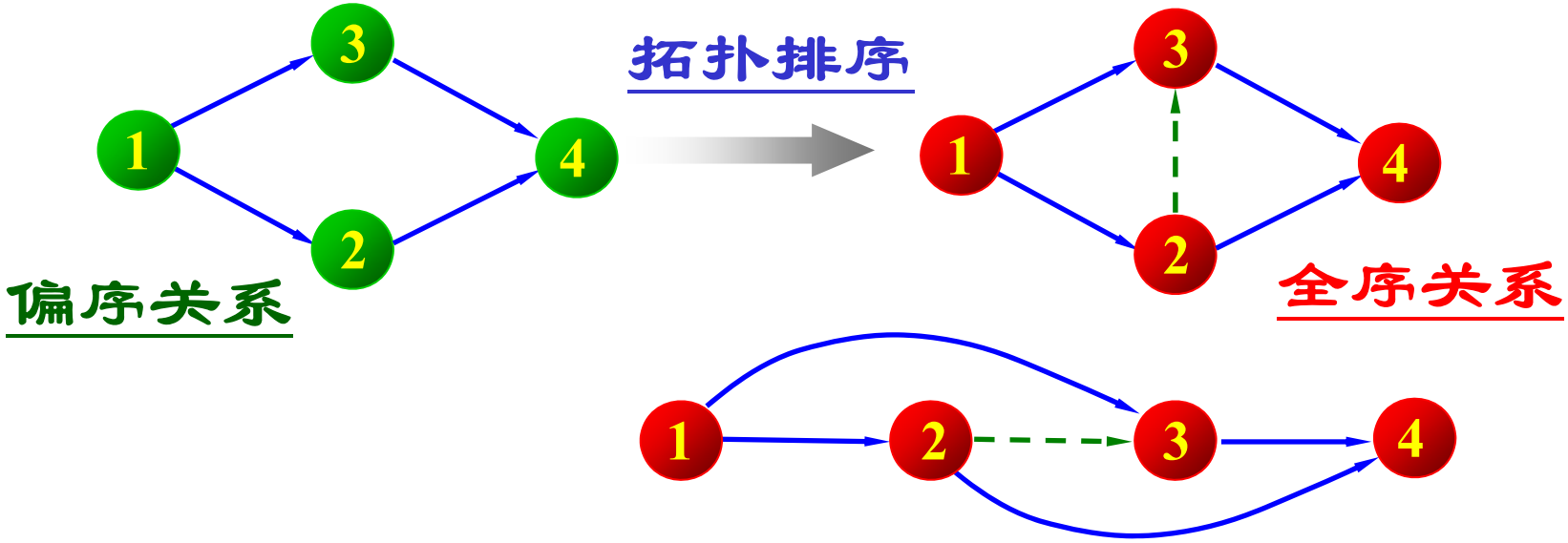
课程代号	课程名称	先修课程
C_1	高等数学	
C_2	程序设计基础	
C_3	离散数学	C_1, C_2
C_4	数据结构	C_3, C_2
C_5	高级语言程序设计	C_2
C_6	编译方法	C_5, C_4
C_7	操作系统	C_4, C_9
C_8	普通物理	C_1
C_9	计算机原理	C_8



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。

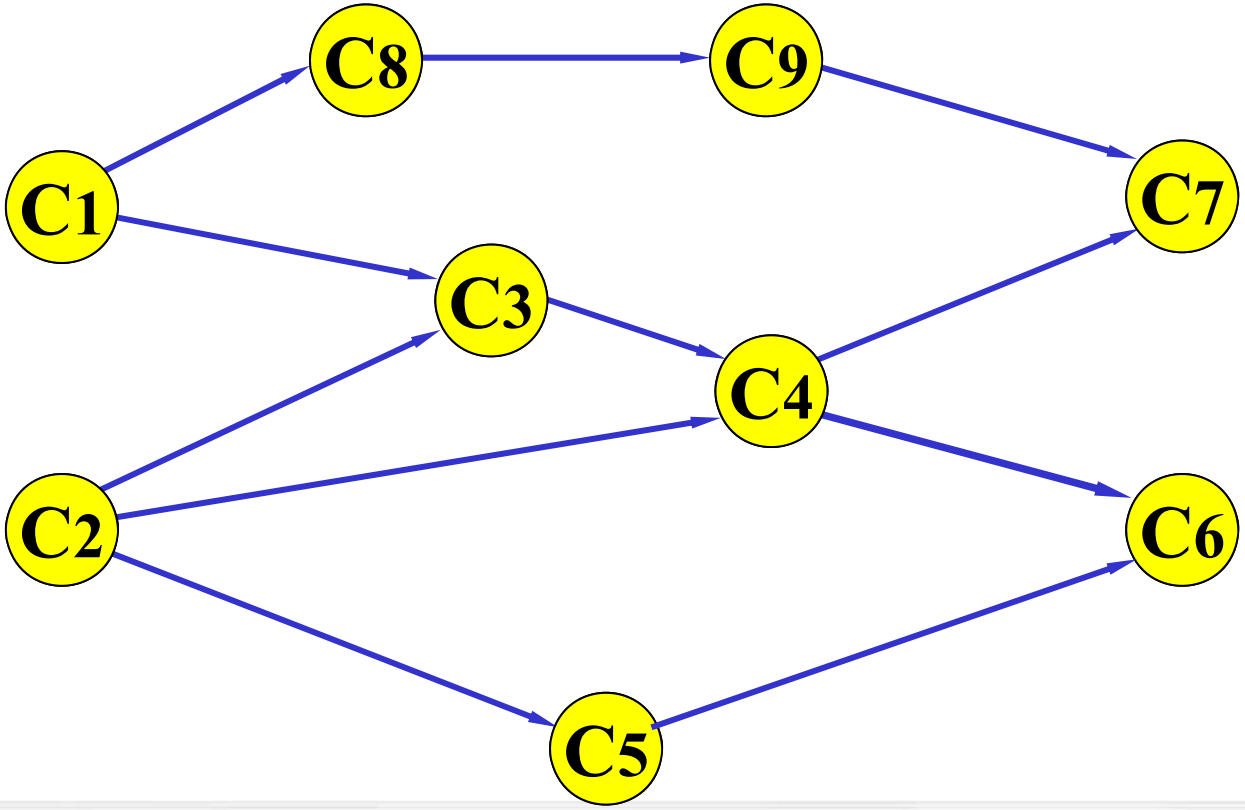


- 这种构造AOV网络全部顶点的拓扑有序序列的运算就叫做拓扑排序。
- 如果通过拓扑排序能将AOV网络的所有顶点都排入一个拓扑有序的序列中，则该网络中必定不会出现有向环。
- 如果AOV网络中存在有向环，此AOV网络所代表的工程是不可行的。

- 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。

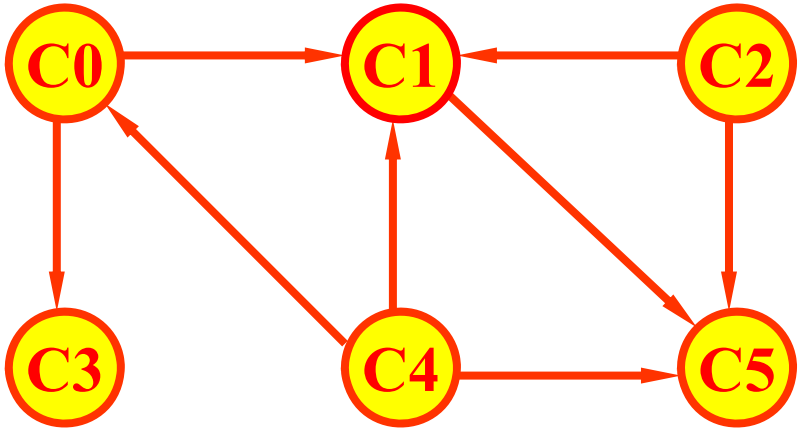
②在AOV网络中选一个没有直接前驱的顶点，并输出之；

③从图中删去该顶点，同时删去所有它发出的有向边；

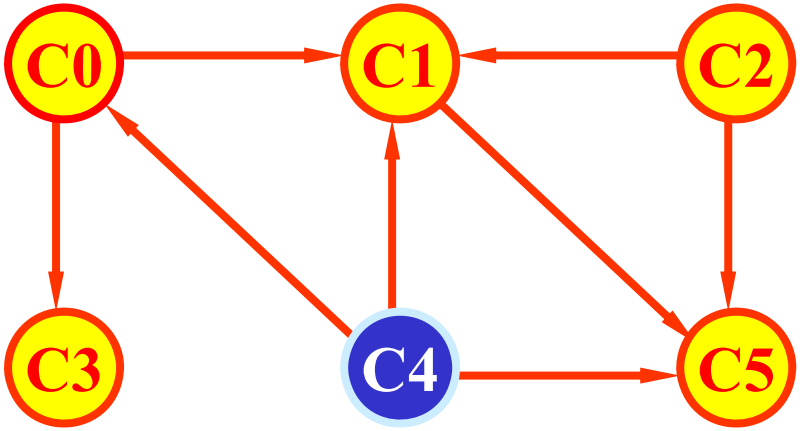
④重复以上 ②、③步，直到

- ☞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。

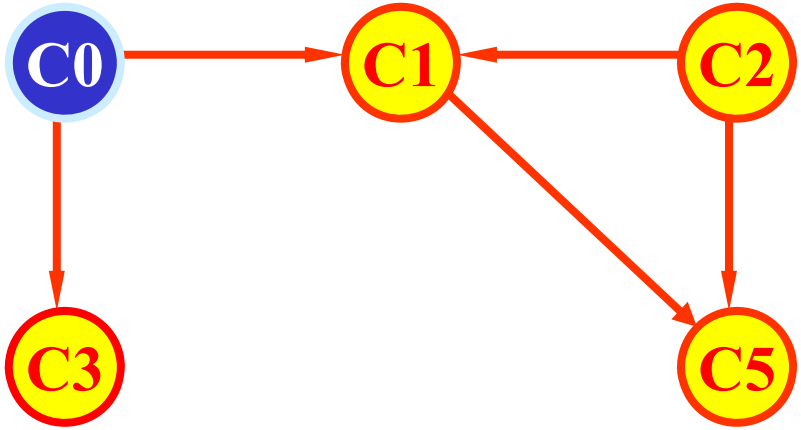
拓扑排序的过程



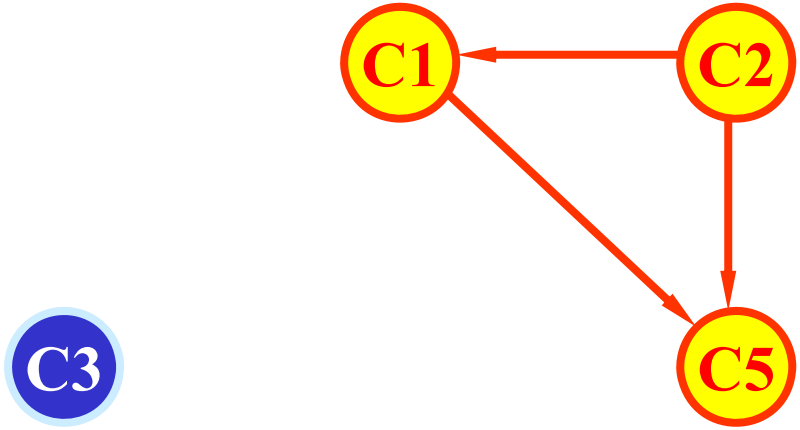
(a) 有向无环图



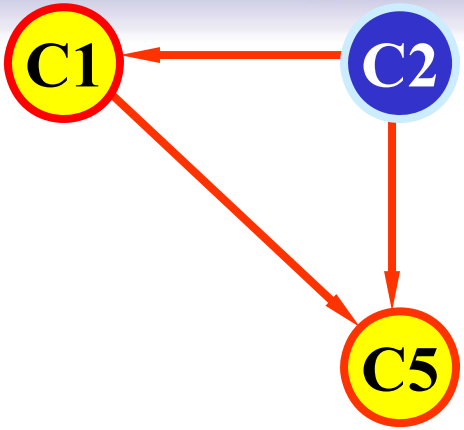
(b) 输出顶点C4



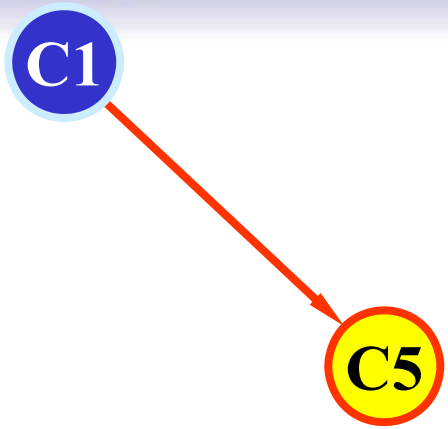
(c) 输出顶点C0



(d) 输出顶点C3



(e) 输出顶点C2



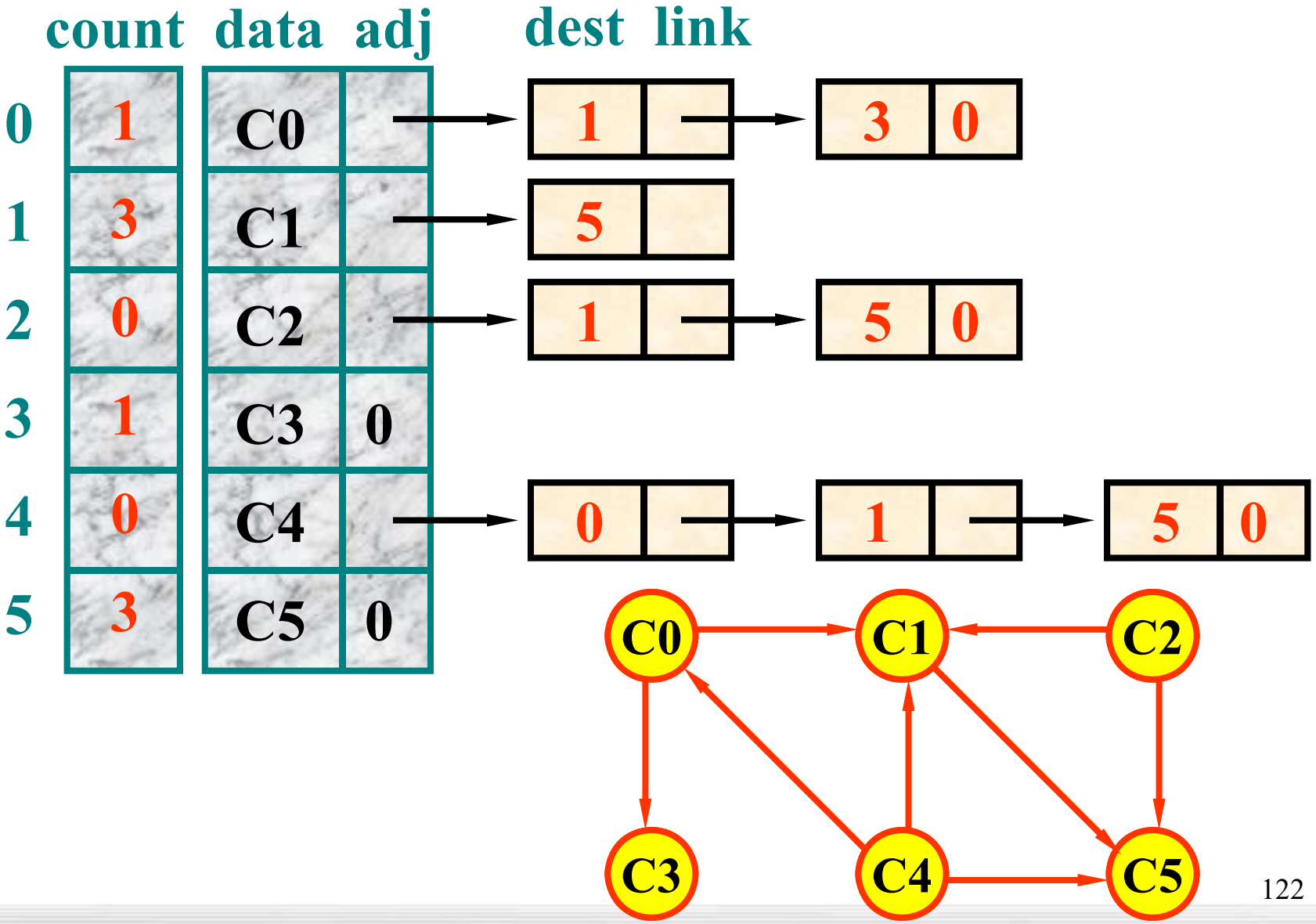
(f) 输出顶点C1

(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。

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
```

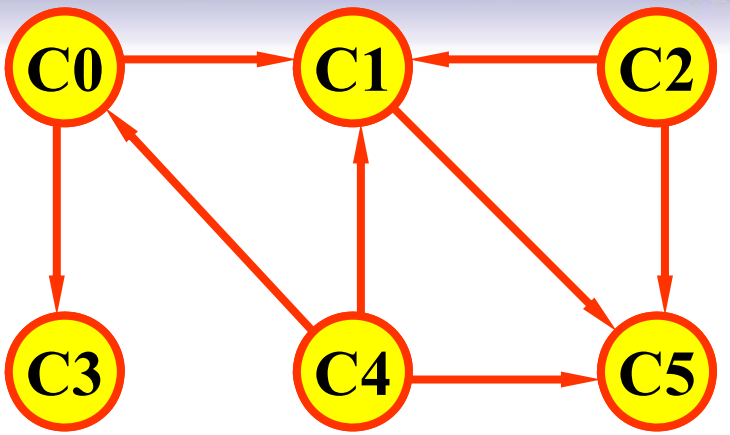
- 在算法中，使用一个存放入度为零的顶点的链式栈，供选择和输出无前驱的顶点。
- 拓扑排序算法可描述如下：
  - 建立入度为零的顶点栈；
  - 当入度为零的顶点栈不空时，重复执行：
    - 从顶点栈中退出一个顶点，并输出之；
    - 从AOV网络中删去这个顶点和它发出的边，边的终顶点入度减一；
    - 如果边的终顶点入度减至 0 ，则该顶点进入度为零的顶点栈。
  - 如果输出顶点个数少于AOV网络的顶点个数，则报告网络中存在有向环。

- 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：

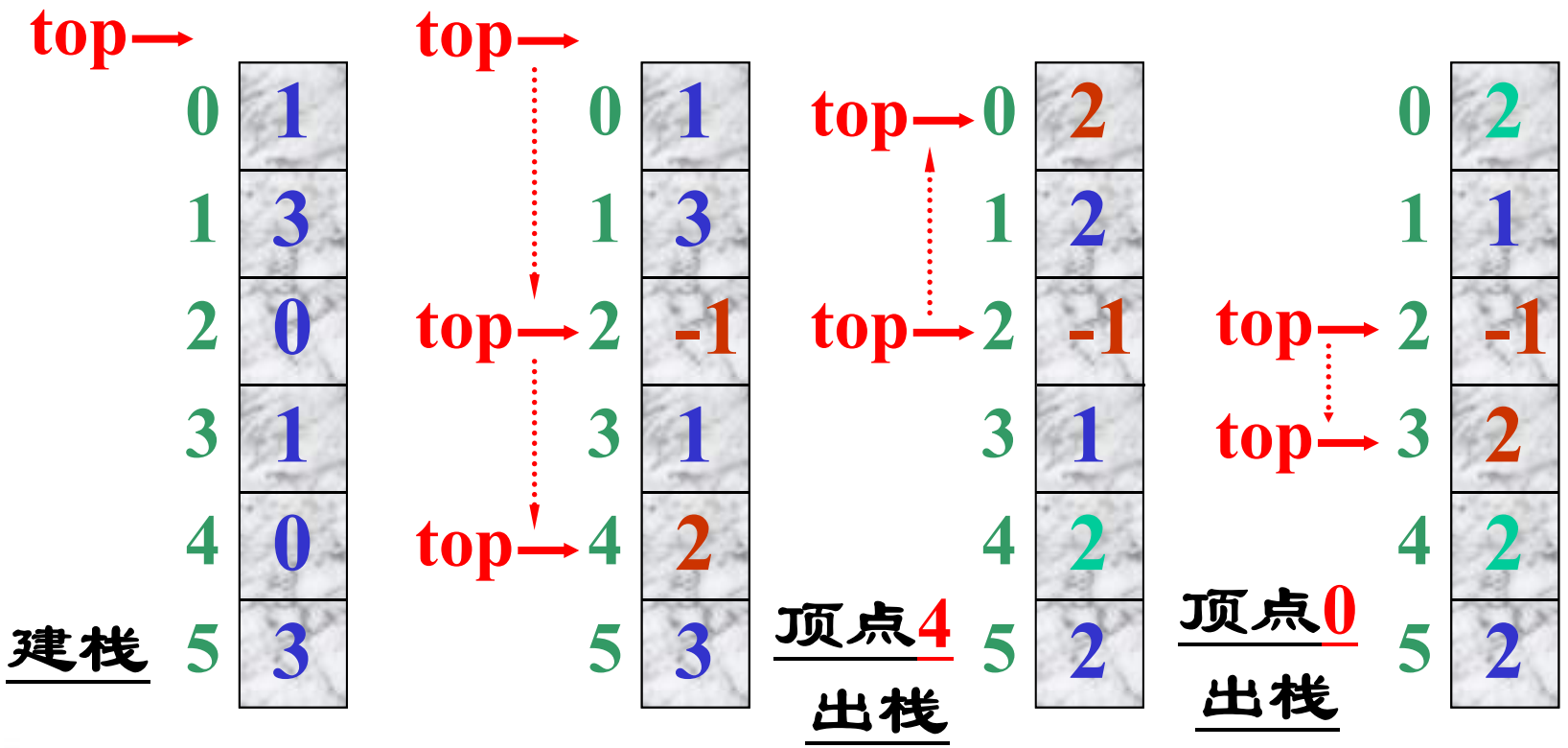
```
count[i] = top;  top = i;
// top 指向新栈顶 i，原栈顶元素在 count[i] 中
```

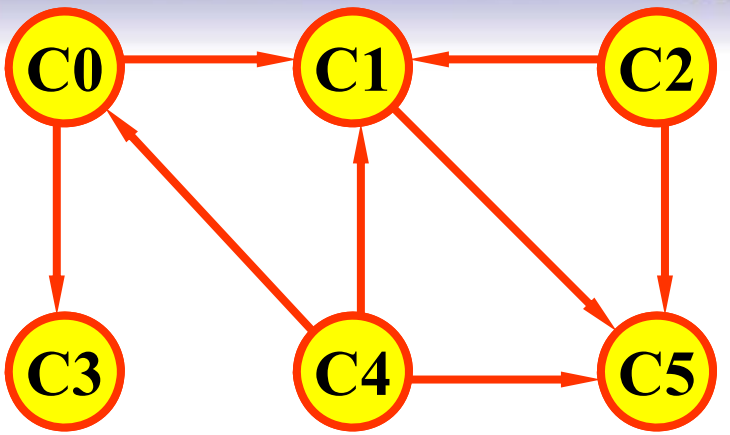
- 退栈操作可以写成：

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

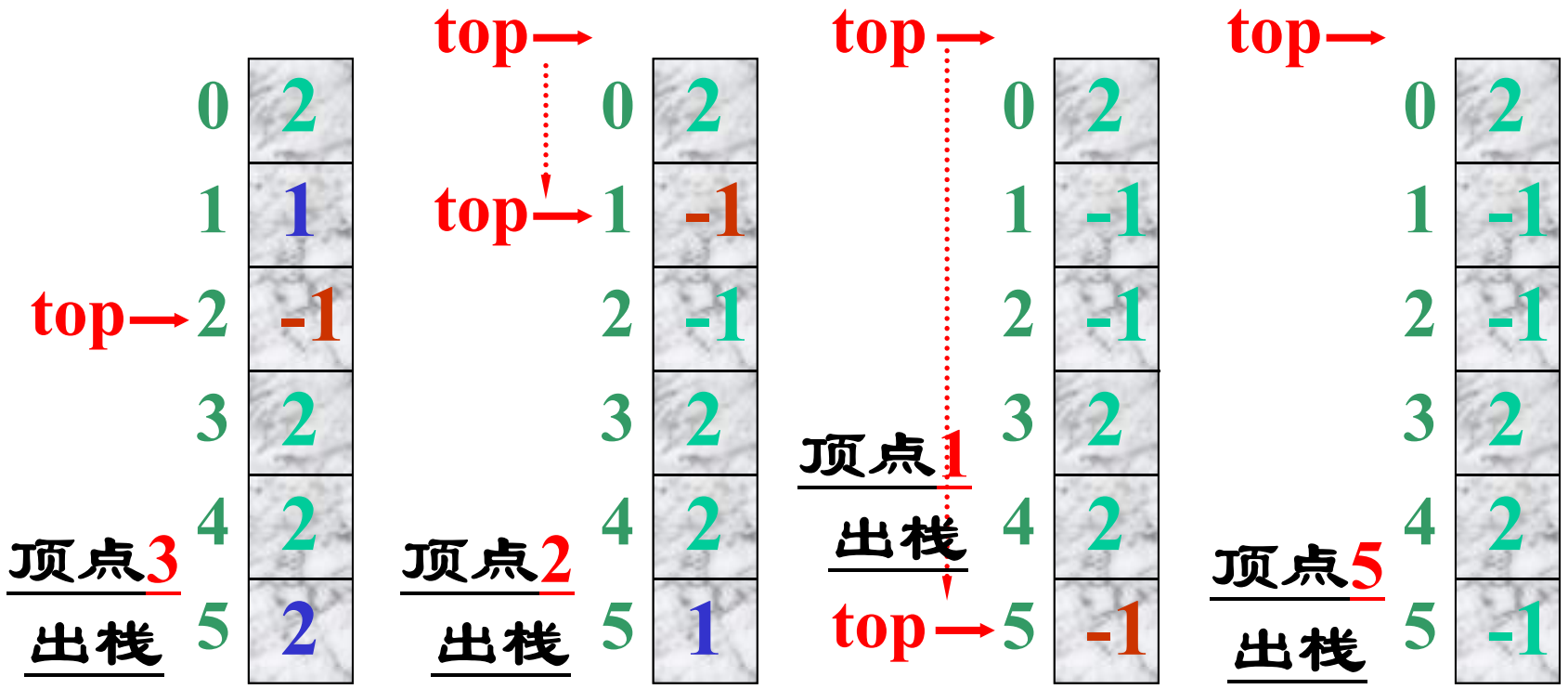


拓扑排序时入度为零的顶点栈在count[ ]中的变化





拓扑排序时入度为零的顶点栈在count[ ]中的变化



拓扑排序的算法

```
void TopologicalSort ( Graph <Type> &G ) {
    int i, j, k;
    int top = -1;  //入度为零的顶点栈初始化
    int n = G.NumberOfVertices ( );  //顶点个数
    int *count = new int[n];  //入度为零顶点栈
    for ( i = 0; i < n; i++ ) count[i] = 0;
    cin >> i >> j;   cout << endl;
    while ( i > -1 && i < n && j > -1 && j < n ) {
        G.InsertEdge ( i, j );  count[j]++;
        cin >> i >> j;  cout << endl;
    }
```

```
for ( i = 0; i < n; i++ ) //入度为零顶点
    if ( count[i] == 0 ) //进栈
       { 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！”<< endl;
         return;
    }
    else { //继续拓扑排序
         int j = top;  top = count[top]; //退栈
         cout << j << endl; //输出
         k = G.GetFirstNeighbor ( j );
```

```
            while ( k != -1 ) {  //扫描出边表
                if ( --count[k] == 0 )  //顶点入度减一
                  { count[k] = top;  top = k; }
                //顶点的入度减至零，进栈
                k = G.GetNextNeighbor ( j, k );
            }
        }
    }
```

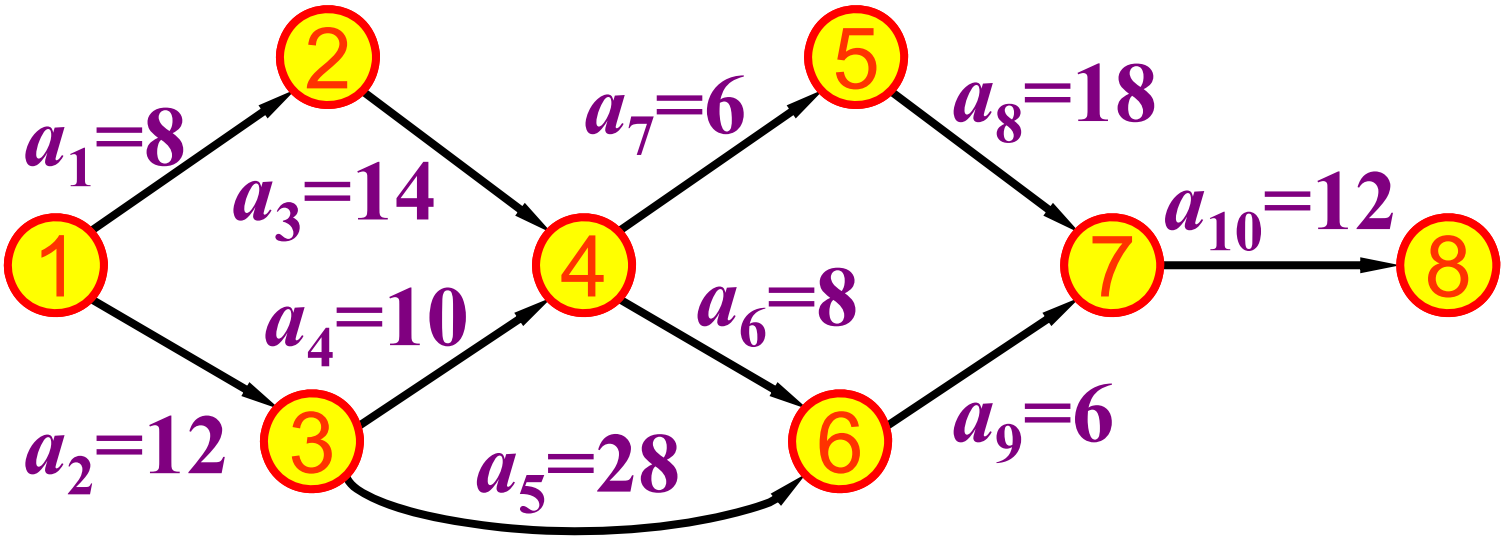
- 分析此拓扑排序算法可知，如果AOV网络有 n 个顶点，e 条边，在拓扑排序的过程中，搜索入度为零的顶点，建立链式栈所需要的时间是 O(n) 。在正常的情况下，有向图有 n 个顶点，每个顶点进一次栈，出一次栈，共输出 n 次。顶点入度减一的运算共执行了 e 次。所以总的时间复杂度为 O(n+e) 。

8.7 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：
  - 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。



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 Ve(i)

是从源点 V_0 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 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。

③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，

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

④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。

⑤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。

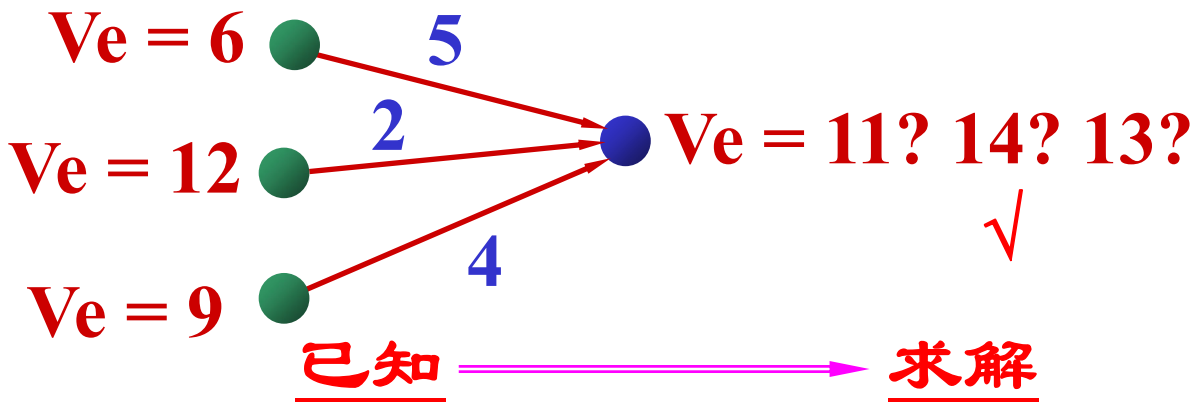
- 为找出关键活动，需要求各个活动的 e[k] 与 l[k]，以判别是否 l[k] == e[k] 。

- 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

$$Ve[j] = \max_i \{Ve[i] + dur(<V_i, V_j>)\},$$

$<V_i, V_j> \in S2$，　$j = 1, 2, \ldots, n-1$

S2 是所有指向 V_j 的有向边 $<V_i, V_j>$ 的集合。

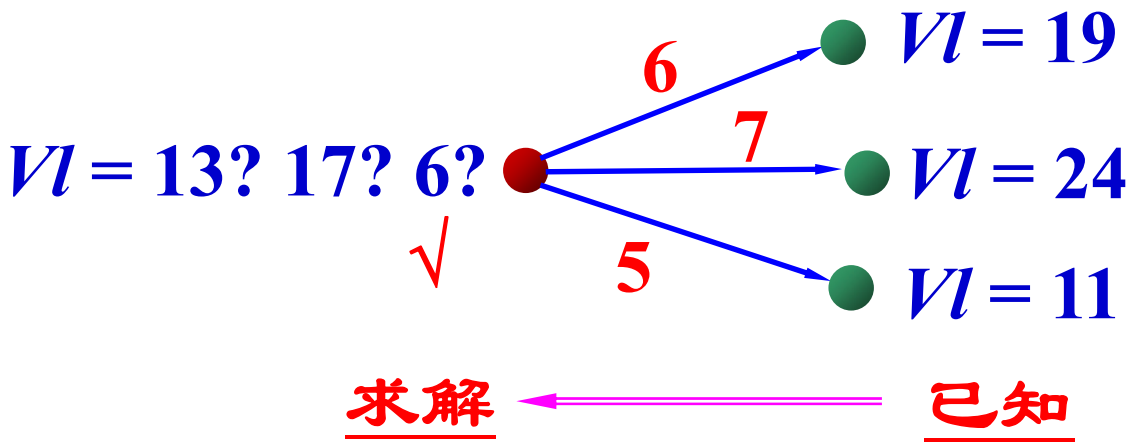


- 从Vl[n-1] = Ve[n-1]开始，反向递推

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

$< V_j, V_k > \in S1$, j = n-2, n-3, …, 0

S1 是所有源自 V_j 的有向边 $< V_j , V_k >$ 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。

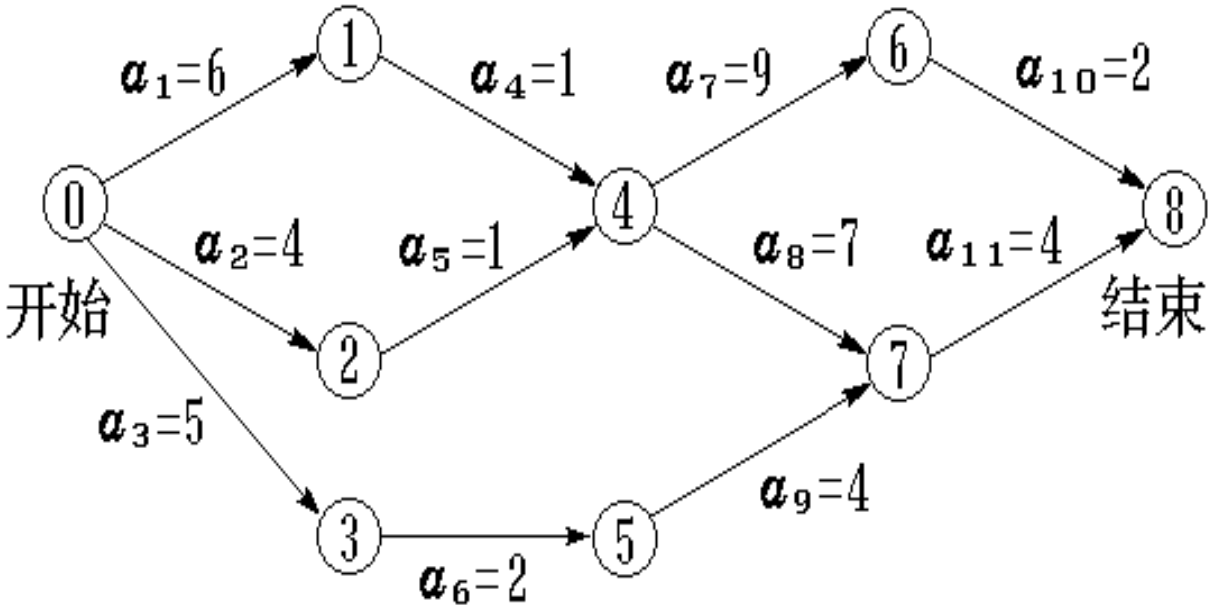
- 设活动 a_k (k=1, 2, …, e) 在带权有向边 $<V_i, V_j>$ 上，其持续时间用 dur ($<V_i, V_j>$) 表示，则有

  e[k] = 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。

- 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	Ve[i]	Vl[i]
V_0	0	0
V_1	6	6
V_2	4	6
V_3	5	8
V_4	7	7
V_5	7	10
V_6	16	16
V_7	14	14
V_8	18	18

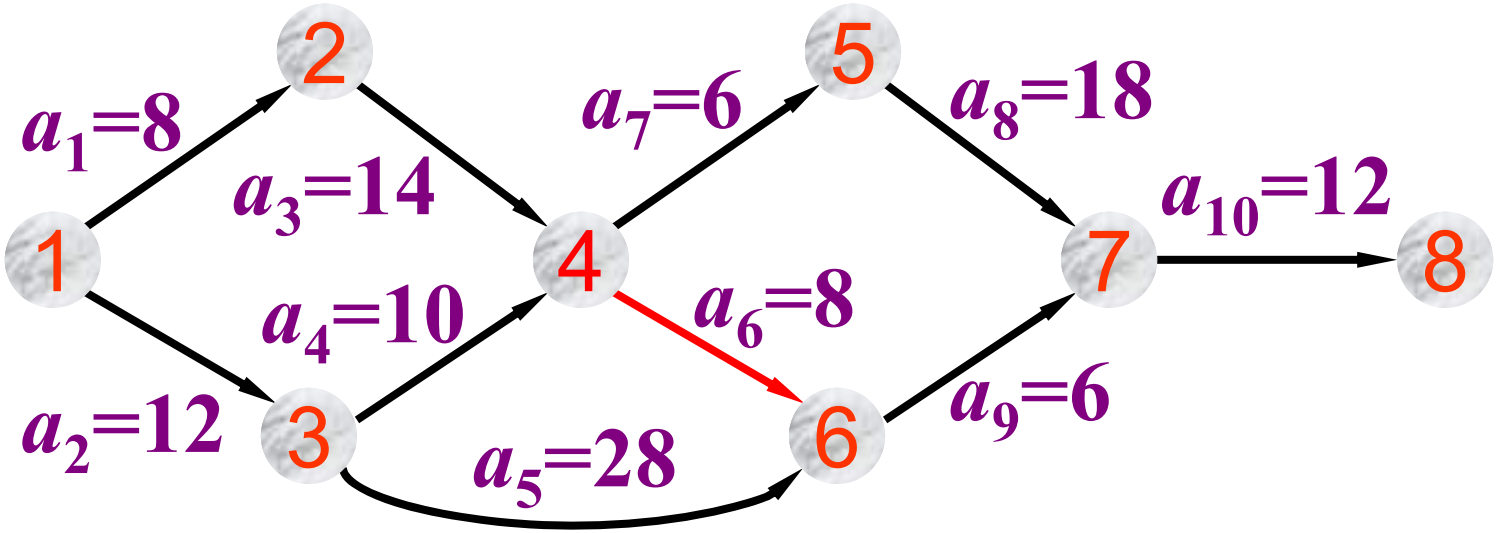


(a)

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
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 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 *Ve = new float[n];
    float *Vl = new float[n];
    for ( i = 0; i < n; i++ ) Ve[i] = 0;
    for ( i = 0; i < n; i++ ) {  //顺向计算 Ve[ ]
        j = G.GetFirstNeighbor ( i );
        while ( j != -1 ) {
```

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

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 $e \leq n(n-1)/2$，解之得 $n \geq 8$；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

- **课程习题**

  - **笔做题——8.17, 8.23, 8.25, 8.29**
    (以作业形式提交)

  - **上机题——8.19, 8.21**

  - **思考题——剩余的其它习题**