第八章 图

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

图的基本概念

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

$$G = (V, E)$$

其中　$V = \{x \mid x \in$ 某个数据对象$\}$

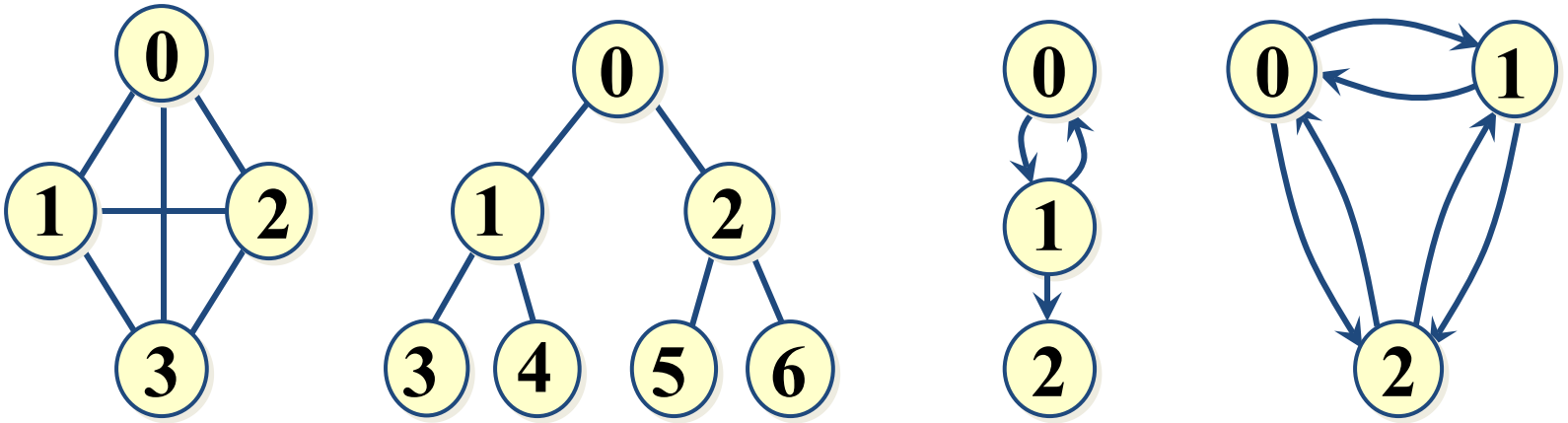
是顶点的有穷非空集合;

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

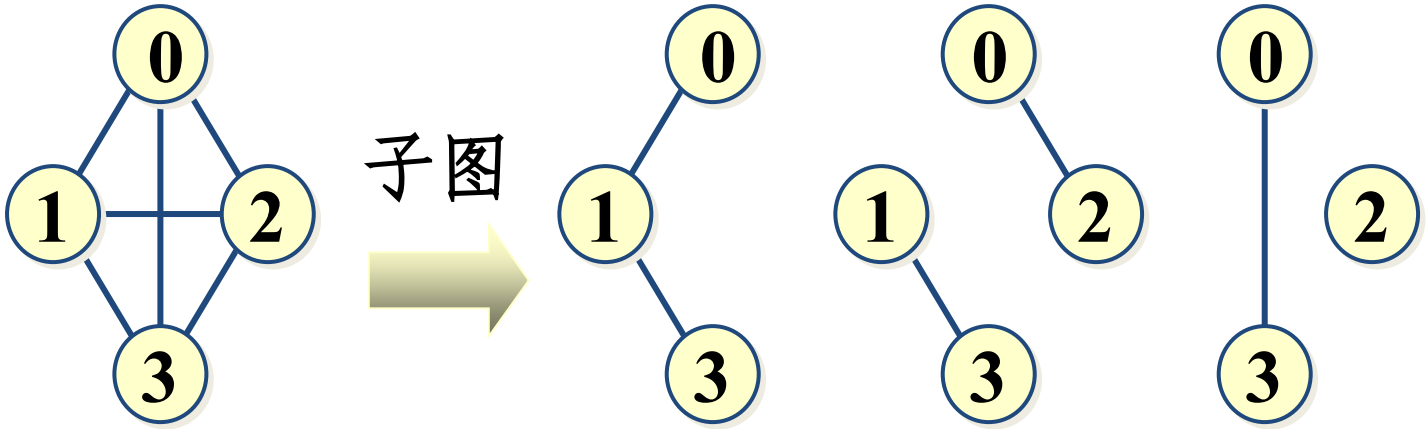
或　$E = \{<x, y> \mid x, y \in V \;\&\&\; Path\ (x, y)\}$

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，则此图为无向完全图。若有 *n* 个顶点的有向图有*n*(*n*-1) 条边，则此图为有向完全图。



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



- 权或权重(weight)　在某些图中，边具有与它相关的数值，称为权重。带权图也叫作网络(network)。

- 顶点的度 (degree)　一个顶点v的度是与它相关联的边的条数, 记作$\deg(v)$。

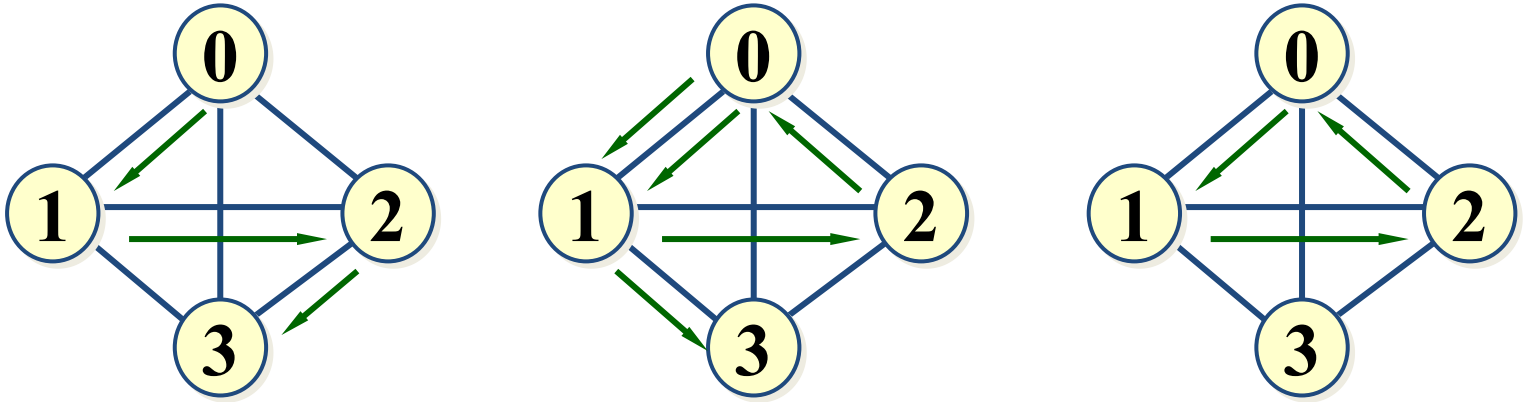
- 在有向图中, 顶点 v 的入度是以 v 为终点的有向边的条数, 记作 $\text{indeg}(v)$; 顶点 v 的出度是以 v 为始点的有向边的条数, 记作 $\text{outdeg}(v)$。在有向图中, 顶点的度等于该顶点的入度与出度之和。

- 路径　在图 $G = (V, E)$ 中, 若从顶点 v_i 出发, 沿一些边经过若干顶点 $v_{p1}, v_{p2}, \cdots, v_{pm}$, 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 的一条路径。它经过的边(v_i, v_{p1})、(v_{p1}, v_{p2})、…、(v_{pm}, v_j) 都是来自于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值之和。

- **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(spanning tree)　一个无向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。若是有向图，则可能得到它的由若干有向树组成的生成森林。

图的抽象数据类型

```
class Graph {
//对象: 由一个非空顶点集合和一个边集合构成
//每条边由一个顶点对来表示。
public:
   Graph();                              //建立一个空的图
   void insertVertex (const T& vertex);
     //插入一个顶点vertex, 该顶点暂时没有入边
   void insertEdge (int v1, int v2, int weight);
     //在图中插入一条边(v1, v2, w)
   void removeVertex (int v);
     //在图中删除顶点v和所有关联到它的边
```

```
void removeEdge (int v1, int v2);
  //在图中删去边(v1,v2)
bool IsEmpty();
  //若图中没有顶点, 则返回true, 否则返回
false
T getWeight (int v1, int v2);
  //函数返回边 (v1,v2) 的权值
int getFirstNeighbor (int v);
  //给出顶点 v 第一个邻接顶点的位置
int 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
};
```

图的存储表示

- 图的模板基类　在模板类定义中的数据类型参数表 <class T, class E> 中，T是顶点数据的类型，E是边上所附数据的类型。

- 这个模板基类是按照最复杂的情况（即带权无向图）来定义的，如果需要使用非带权图，可将数据类型参数表 <class T, class E> 改为 <class T>。

- 如果使用的是有向图，也可以对程序做相应的改动。

图的模板基类

```
const int maxWeight = ……;          //无穷大的值(=∞)
const int DefaultVertices = 30;     //最大顶点数(=n)
template <class T, class E>
class Graph {                       //图的类定义
protected:
    int maxVertices;                //图中最大顶点数
    int numEdges;                   //当前边数
    int numVertices;                //当前顶点数
    virtual int getVertexPos (T vertex);
     //给出顶点vertex在图中位置
public:
```

```
Graph (int sz = DefaultVertices);          //构造函数
~Graph();                                  //析构函数
bool GraphEmpty () const                   //判图空否
   { return numEdges == 0; }
int NumberOfVertices () { return numVertices; }
 //返回当前顶点数
int NumberOfEdges () { return numEdges; }
 //返回当前边数
virtual T getValue (int i);                //取顶点 i 的值
virtual E getWeight (int v1, int v2);      //取边上权值
virtual int getFirstNeighbor (int v);
 //取顶点 v 的第一个邻接顶点
```

```
    virtual int getNextNeighbor (int v, int w);
     //取邻接顶点 w 的下一邻接顶点
    virtual bool insertVertex (const T vertex);
     //插入一个顶点vertex
    virtual bool insertEdge (int v1, int v2, E cost);
     //插入边(v1,v2), 权为cost
    virtual bool removeVertex (int v);
     //删去顶点 v 和所有与相关联边
    virtual bool removeEdge (int v1, int v2);
     //在图中删去边(v1,v2)
};
```

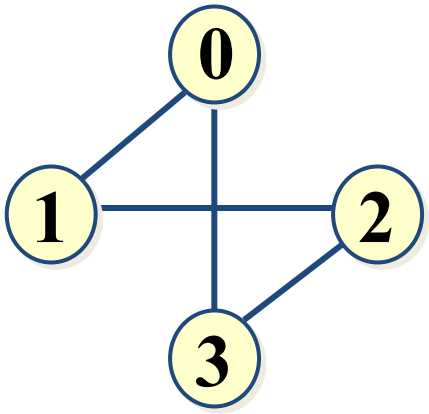
所有虚函数都与具体存储表示相关。

邻接矩阵 (Adjacency Matrix) 表示

- 在图的邻接矩阵表示中，有一个记录各个顶点信息的顶点表，还有一个表示各个顶点之间关系的邻接矩阵。

- 设图 **A = (V, E)** 是一个有 ***n*** 个顶点的图, 图的邻接矩阵是一个二维数组 **A.Edge[*n*][*n*]**，定义:

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



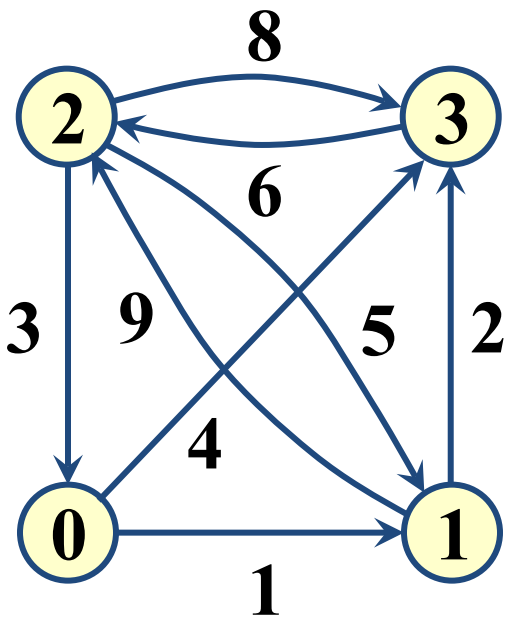
$$A.Edge = \begin{bmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{bmatrix}$$

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

- 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（带权图）的邻接矩阵

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



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

用邻接矩阵表示的图的类定义

```
template <class T, class E>
class Graphmtx : public Graph<T, E> {
friend istream& operator >> ( istream& in,   Graphmtx<T,
    E>& G);
friend ostream& operator << (ostream& out, Graphmtx<T,
    E>& G);                    //输出
private:
    T *VerticesList;                          //顶点表
    E **Edge;                                 //邻接矩阵
    int getVertexPos (T vertex) {
     //给出顶点vertex在图中的位置
       for (int i = 0; i < numVertices; i++)
           if (VerticesList[i] == vertex) return i;
       return -1;
    };
```

```
public:
  Graphmtx (int sz = DefaultVertices);    //构造函数
  ~Graphmtx ()                            //析构函数
    { delete [ ]VerticesList;
      for (i = 0; i < maxVertices; i++)
          delete [ ] Edge [i];
       delete [ ] Edge; }
  T getValue (int i) {
   //取顶点 i 的值, i 不合理返回0
    return i >= 0 && i < numVertices ?
              VerticesList[i] : NULL;
  }
  E getWeight (int v1, int v2) {  //取边(v1,v2)上权值
    return v1 != -1 && v2 != -1 ? Edge[v1][v2] : 0;
  }
```

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

```
template <class T, class E>
Graphmtx<T, E>::Graphmtx (int sz) {      //构造函数
    maxVertices = sz;
    numVertices = 0;  numEdges = 0;
    int i, j;
    VerticesList = new T[maxVertices];  //创建顶点表
    Edge = (E **) new E*[maxVertices];
    for (i = 0; i < maxVertices; i++)
        Edge[i] = new E[maxVertices];   //邻接矩阵
    for (i = 0; i < maxVertices; i++)       //矩阵初始化
        for (j = 0; j < maxVertices; j++)
            Edge[i][j] = (i == j)  ?  0 : maxWeight;
};
```

```
template <class T, class E>
int Graphmtx<T, E>::getFirstNeighbor (int v) {
//给出顶点位置为v的第一个邻接顶点的位置,
//如果找不到,则函数返回-1
    if (v != -1) {
        for (int col = 0; col < numVertices; col++)
            if (Edge[v][col] > 0 && Edge[v][col] < maxWeight)
                return col;
    }
    return -1;
};
```

```
template <class T, class E>
int Graphmtx<T, E>::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 col = w+1; col < numVertices; col++)
          if (Edge[v][col] >0 && Edge[v][col] < maxWeight)
             return col;
    }
    return -1;
};
```

邻接表 (Adjacency List)

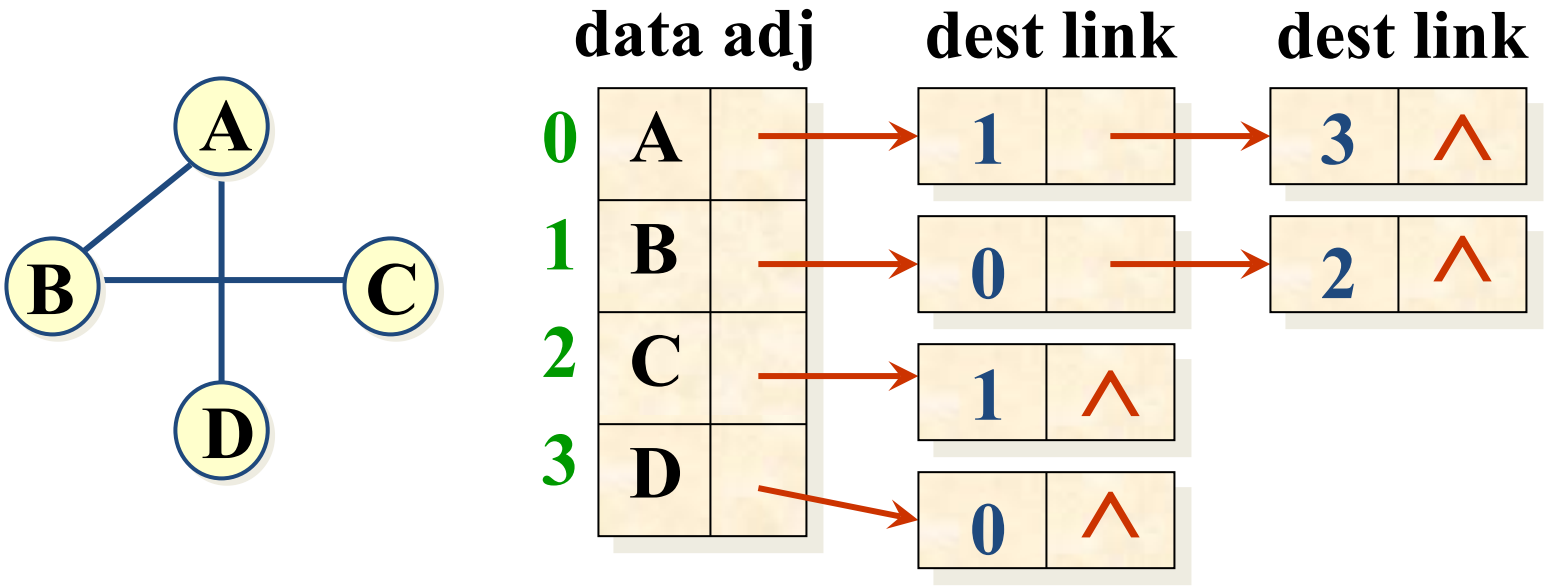
- 以邻接矩阵表示图，当边数远远小于n^2 的时候，大量元素是**maxWeight**，会造成空间浪费。

- 邻接表是邻接矩阵的改进形式。为此需要把邻接矩阵的各行分别组织为一个单链表。

- 在邻接表中，同一个顶点发出的边链接在同一个边链表中，每一个链表结点代表一条边（边结点），结点中有另一顶点的标识 **dest** 和指针 **link**。对于带权图，边结点中还要保存该边的权值**cost**。

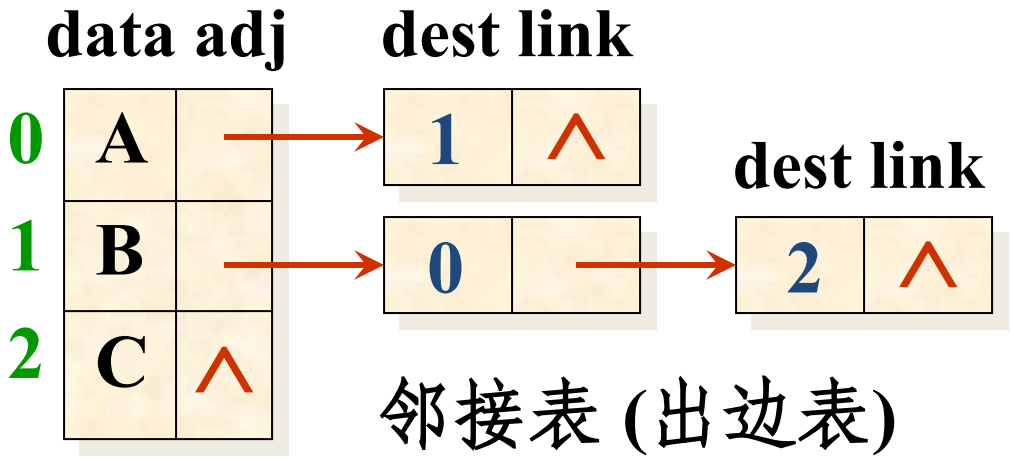
- 顶点表的第 **i** 个顶点中保存该顶点的数据，以及它对应边链表的头指针**adj**。

无向图的邻接表

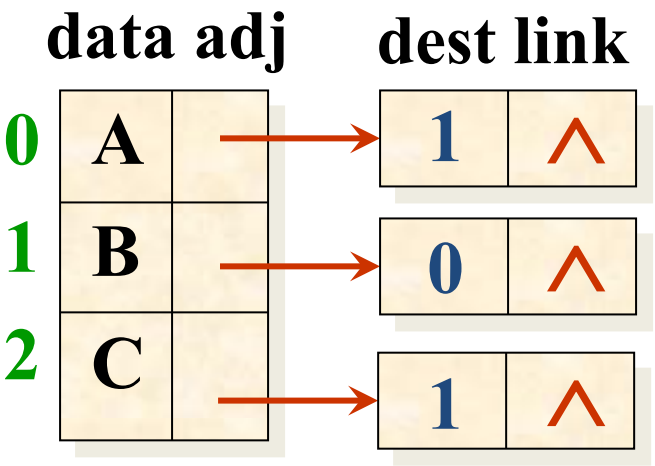


- 统计某顶点对应边链表中结点个数，可得该顶点的度。
- 某条边(v_i, v_j)在邻接表中有两个边结点，分别在第 i 个顶点和第 j 个顶点对应的边链表中。

有向图的邻接表和逆邻接表

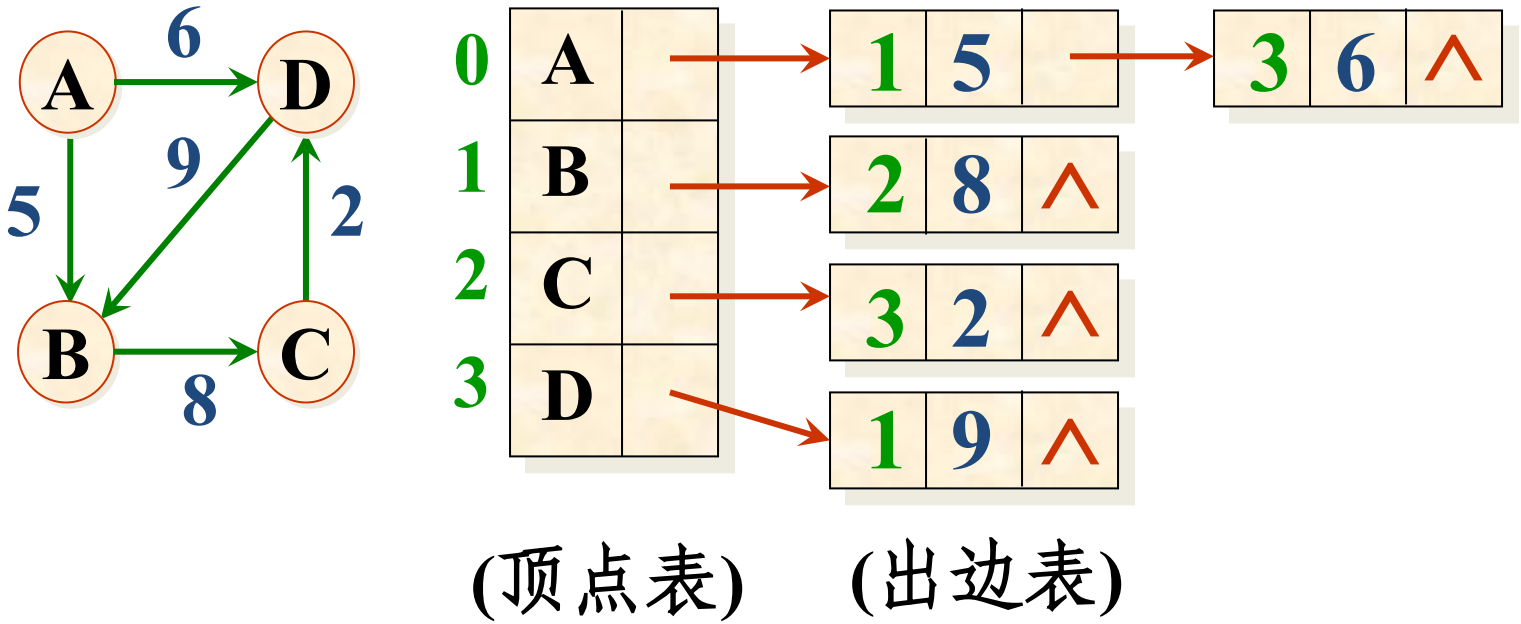


邻接表 (出边表)



逆邻接表 (入边表)

网络(带权图)的邻接表



- 统计出边表中结点个数，得到该顶点的出度;
- 统计入边表中结点个数，得到该顶点的入度。

用邻接表表示的图的类定义

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

```
template <class T, class E>
struct Vertex {                          //顶点的定义
    T data;                              //顶点的名字
    Edge<T, E> *adj;                     //边链表的头指针
};

template <class T, class E>
class Graphlnk : public Graph<T, E> {    //图的
  类定义
friend istream& operator >> (istream& in,
        Graphlnk<T, E>& G);                    //输入
friend ostream& operator << (ostream& out,
        Graphlnk<T, E>& G);                    //输出
```

```
private:
    Vertex<T, E> *NodeTable;
        //顶点表 (各边链表的头结点)
    int getVertexPos (const T vertex) {
        //给出顶点vertex在图中的位置
        for (int i = 0; i < numVertices; i++)
            if (NodeTable[i].data == vertex) return i;
        return -1;
    }
public:
    GraphInk (int sz = DefaultVertices);  //构造函数
    ~GraphInk();                          //析构函数
```

```
    T getValue (int i) {                    //取顶点 i 的值
        return (i >= 0 && i < NumVertices) ?
            NodeTable[i].data : 0;
    }
    E getWeight (int v1, int v2);           //取边(v1,v2)权值
    bool insertVertex (const T& vertex);
    bool removeVertex (int v);
    bool insertEdge (int v1, int v2, E cost);
    bool removeEdge (int v1, int v2);
    int getFirstNeighbor (int v);
    int getNextNeighbor (int v, int w);
};
```

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

```
template <class T, class E>
Graphlnk<T, E>::~Graphlnk() {
//析构函数：删除一个邻接表
    for (int i = 0; i < maxVertices; i++ ) {
        Edge<T, E> *p = NodeTable[i].adj;
        while (p != NULL) {
            NodeTable[i].adj = p->link;
            delete p;  p = NodeTable[i].adj;
        }
    }
    delete [ ]NodeTable;                    //删除顶点
    表数组
};
```

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

```
template <class T, class E>
int Graphlnk<T, E>::getNextNeighbor (int v, int w) {
//给出顶点v的邻接顶点w的下一个邻接顶点的位置,
//若没有下一个邻接顶点, 则函数返回-1
    if (v != -1) {                                  //顶点v存在
        Edge<T, E> *p = NodeTable[v].adj;
        while (p != NULL && p->dest != w)
            p = p->link;
        if (p != NULL && p->link != NULL)
            return p->link->dest;       //返回下一个邻接顶点
    }
    return -1;                                      //下一邻接顶点不存在
};
```

邻接矩阵与邻接表对比

- 空间
- 时间

邻接多重表 (Adjacency Multilist)

- 无向图的邻接表表示中，每条边被存储了两遍，既浪费了空间，又造成为边做标记等有关边的处理的不方便

- 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

- 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**。

无向图

- 顶点结点的结构

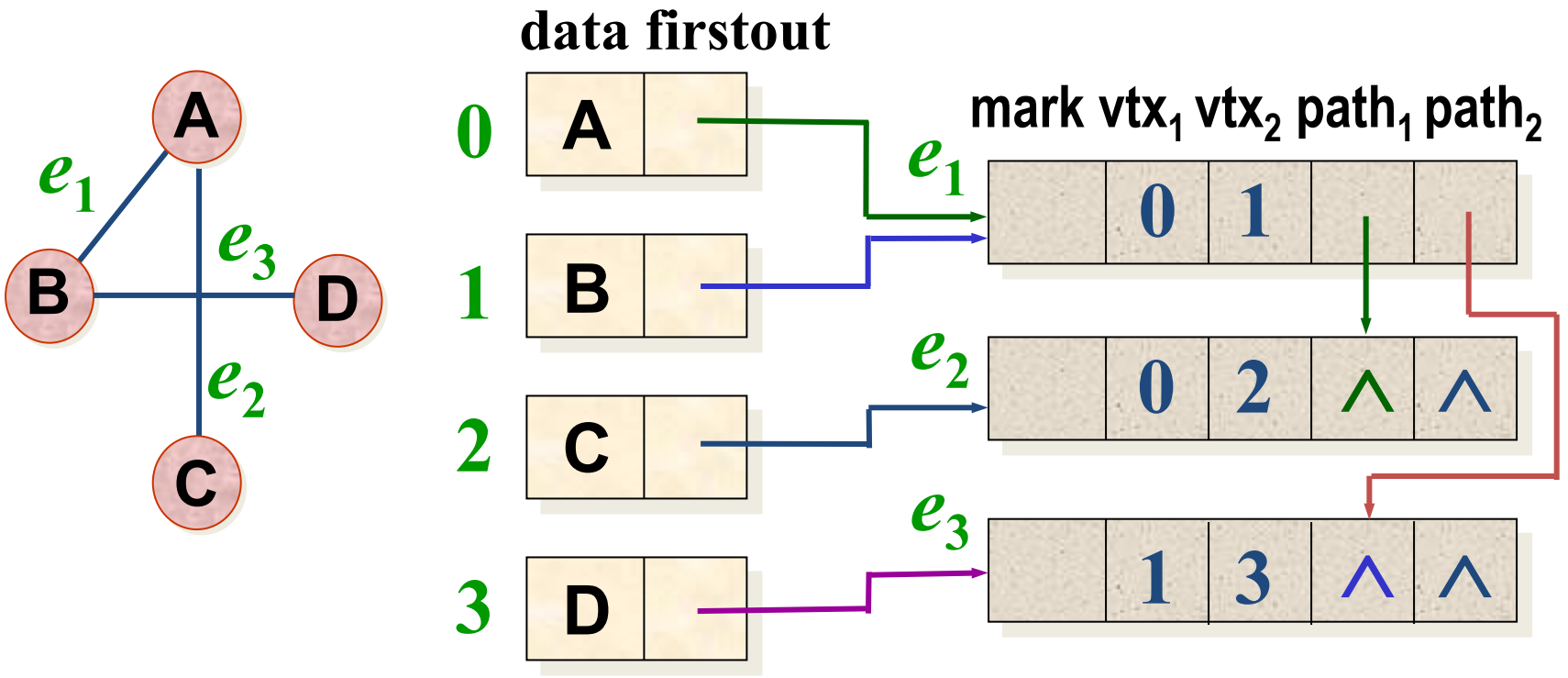
- 存储顶点信息的结点表以顺序表方式组织,每个顶点结点有两个数据成员：其中，**data** 存放与该顶点相关的信息，**firstout** 是指向依附于该顶点的第一条边的指针。

- 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。

无向图

- 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

- 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

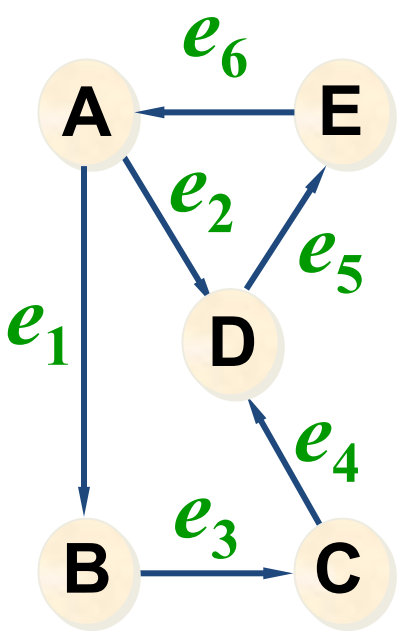
- 其中，**mark**是处理标记；**vertex1**和**vertex2**指明该有向边始顶点和终顶点的位置。**path1**是链接指针，指向与该边有同一始顶点的下一条边（出边表）；**path2**也是链接指针，指向与该边有同一终顶点的下一条边（入边表）。需要时还可有权值域**cost**。

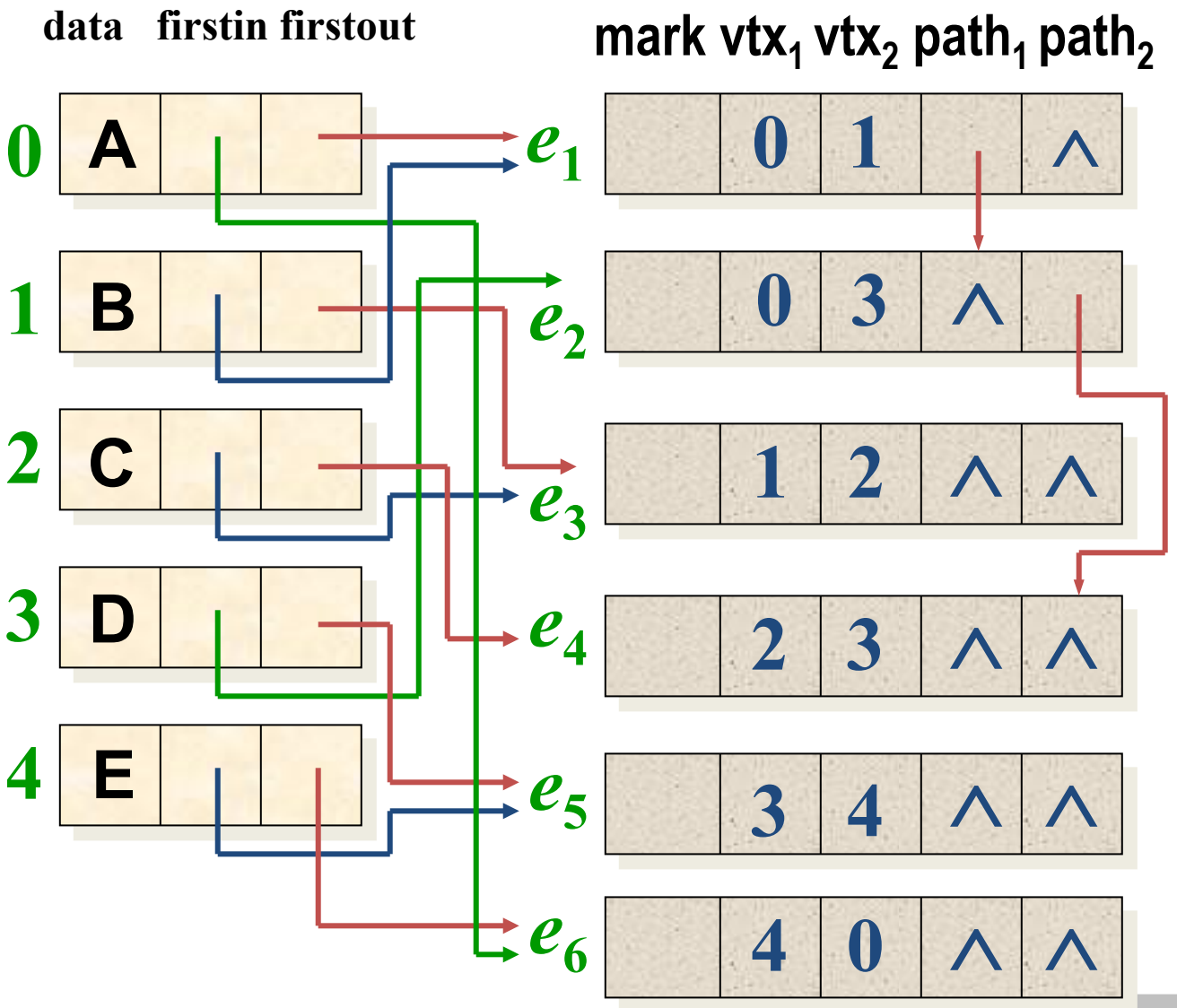
有向图

- 顶点结点的结构

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

有向图


A
E
D
B
C
e_1
e_2
e_3
e_4
e_5
e_6


data firstin firstout
mark vtx_1 vtx_2 $path_1$ $path_2$
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 ∧ ∧

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 [ ]**。

- 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。

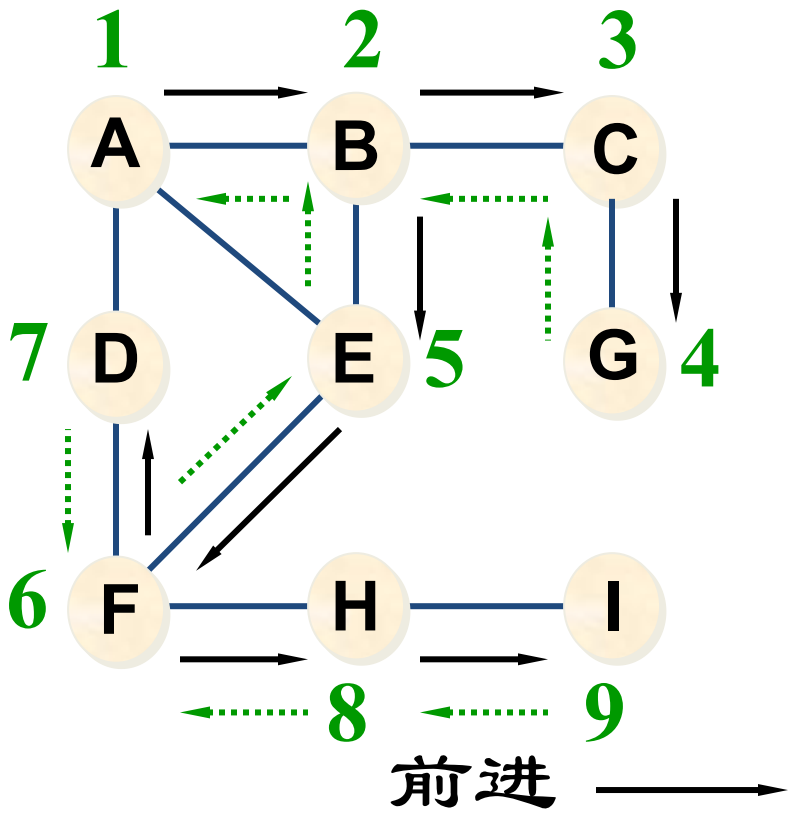
- 图的遍历的分类:
  - 深度优先搜索

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

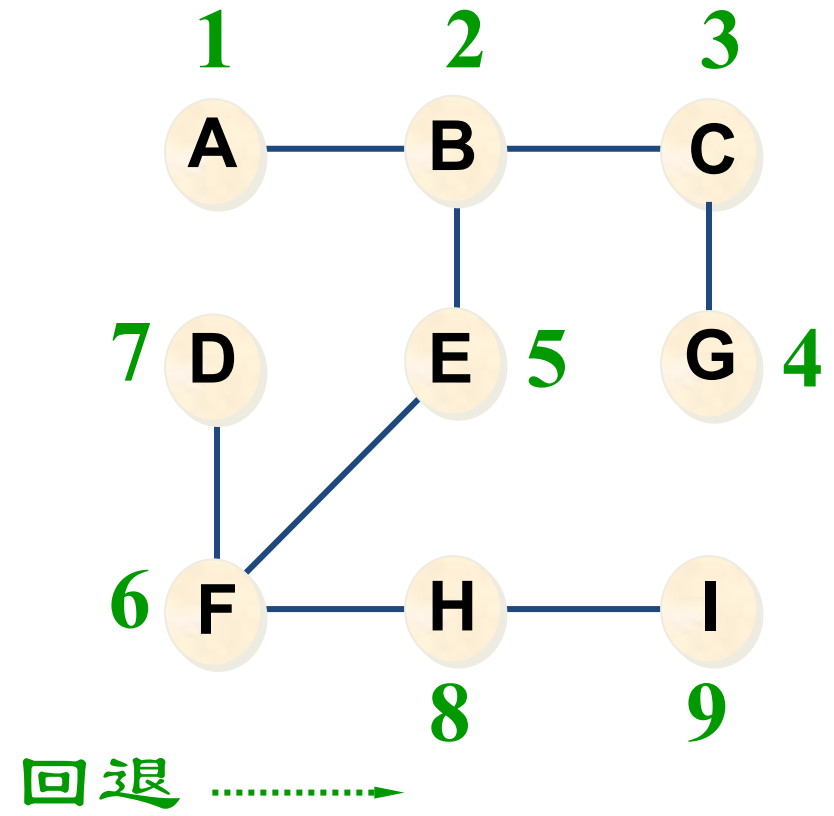
    BFS (Breadth First Search)

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

- **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。

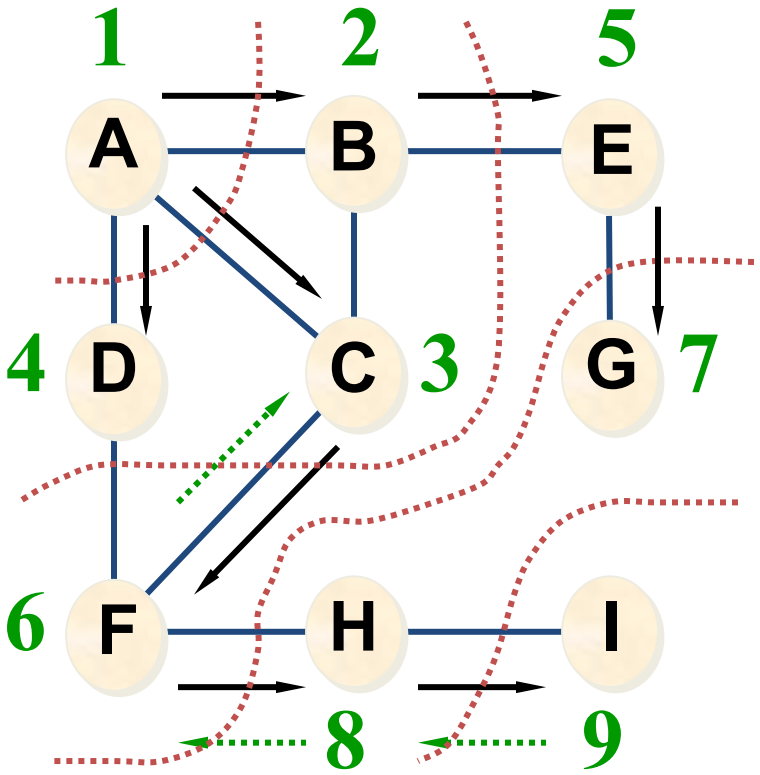
图的深度优先搜索算法

```
template<class T, class E>
void DFS (Graph<T, E>& G, const T& v) {
//从顶点v出发对图G进行深度优先搜索的主过程
    int i, loc, n = G.NumberOfVertices();   //顶点个数
    bool *visited = new bool[n];            //创建辅助数组
    for (i = 0; i < n; i++) visited [i] = false;
      //辅助数组visited初始化
    loc = G.getVertexPos(v);
    DFS (G, loc, visited); //从顶点loc开始深度优先搜索
    delete [] visited;                      //释放visited
};
```

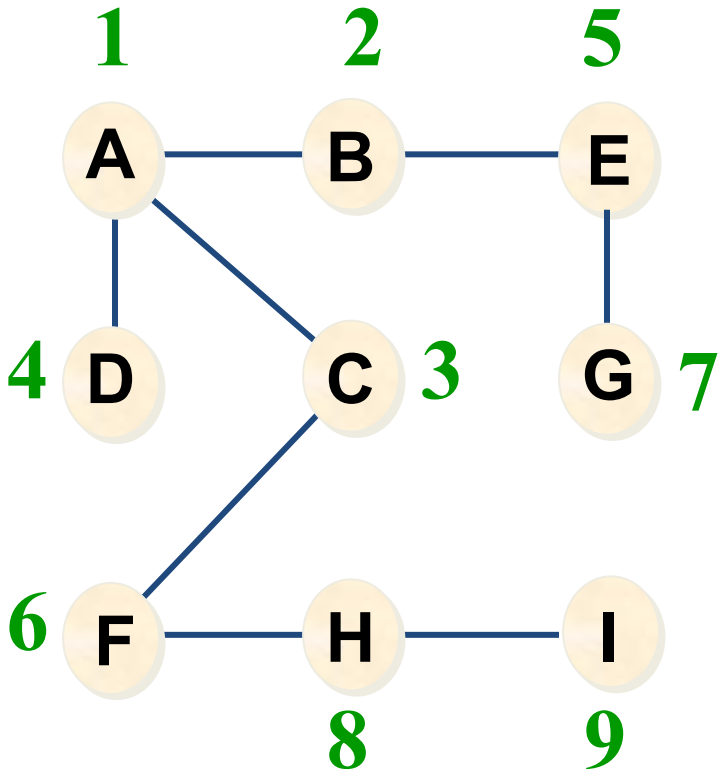
```
template<class T, class E>
void DFS (Graph<T, E>& G, int v, bool visited[]) {
    cout << G.getValue(v) << ' ';        //访问顶点v
    visited[v] = true;                   //作访问标记
    int w = G.getFirstNeighbor (v);      //第一个邻接顶点
    while (w != -1) { //若邻接顶点w存在
        if ( !visited[w] ) DFS(G, w, visited);
                                 //若w未访问过, 递归访问顶点w
        w = G.getNextNeighbor (v, w); //下一个邻接顶点
    }
};
```

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 $w_1, w_2, \ldots, w_t$，
  - 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。

- 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 T, class E>
void BFS (Graph<T, E>& G, const T& v) {
    int i, w, n = G.NumberOfVertices();
                    //图中顶点个数
```

```
bool *visited = new bool[n];
for (i = 0; i < n; i++) visited[i] = false;
int loc = G.getVertexPos (v);          //取顶点号
cout << G.getValue (loc) << ' ';       //访问顶点v
visited[loc] = true;                   //做已访问标记
Queue<int> Q;  Q.EnQueue (loc);
                //顶点进队列, 实现分层访问
while (!Q.IsEmpty() ) {                //循环, 访问所有结点
   Q.DeQueue (loc);
   w = G.getFirstNeighbor (loc);  //第一个邻接顶点
   while (w != -1) {                   //若邻接顶点w存在
      if (!visited[w]) {               //若未访问过
```

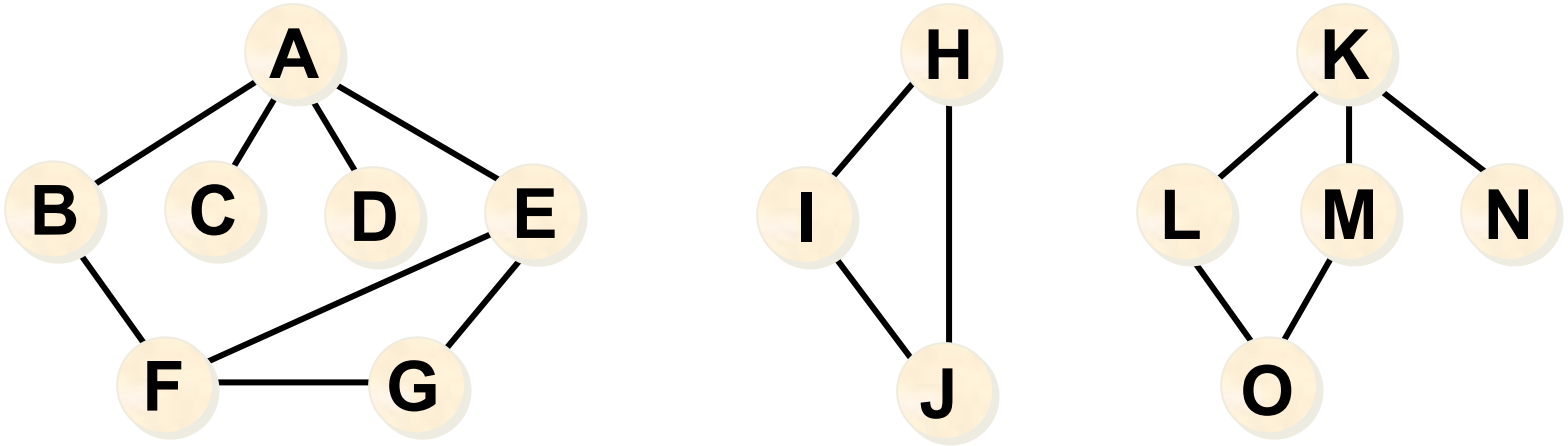
```
            cout << G.getValue (w) << ' ';      //访问
            visited[w] = true;
            Q.EnQueue (w);                      //顶点w进队列
        }
        w = G.getNextNeighbor (loc, w);
                //找顶点loc的下一个邻接顶点
    }
}               //外层循环，判队列空否
delete [] visited;
};
```

连通分量 (Connected component)

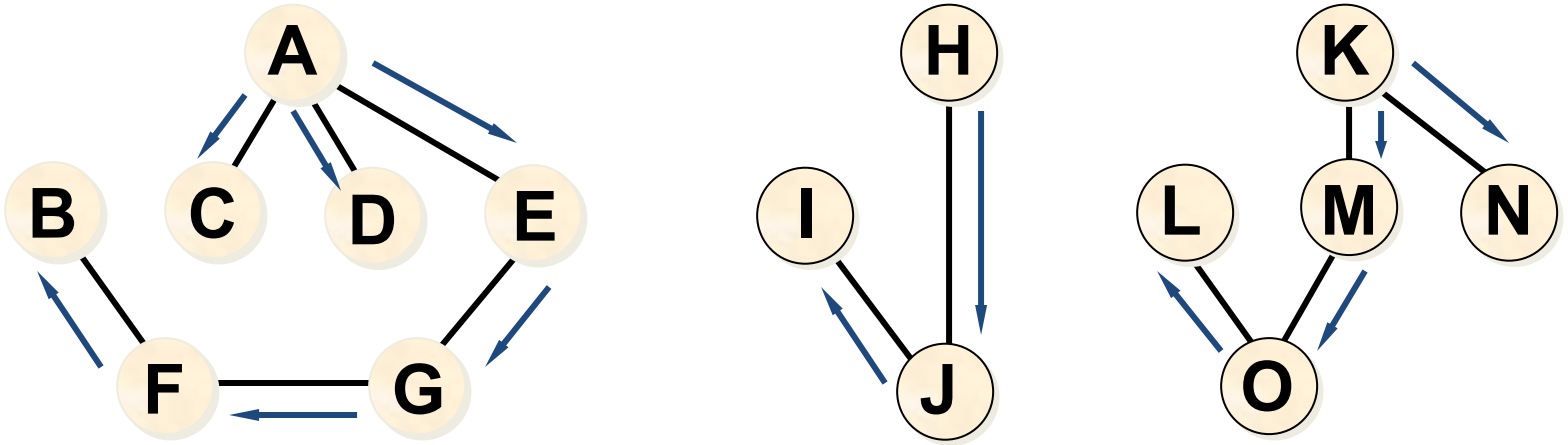
- 当无向图为非连通图时，从图中某一顶点出发，利用深度优先搜索算法或广度优先搜索算法不可能遍历到图中的所有顶点，只能访问到该顶点所在极大连通子图（连通分量）的所有顶点。

- 若从无向图每一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。

- 例如，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



非连通图的生成森林

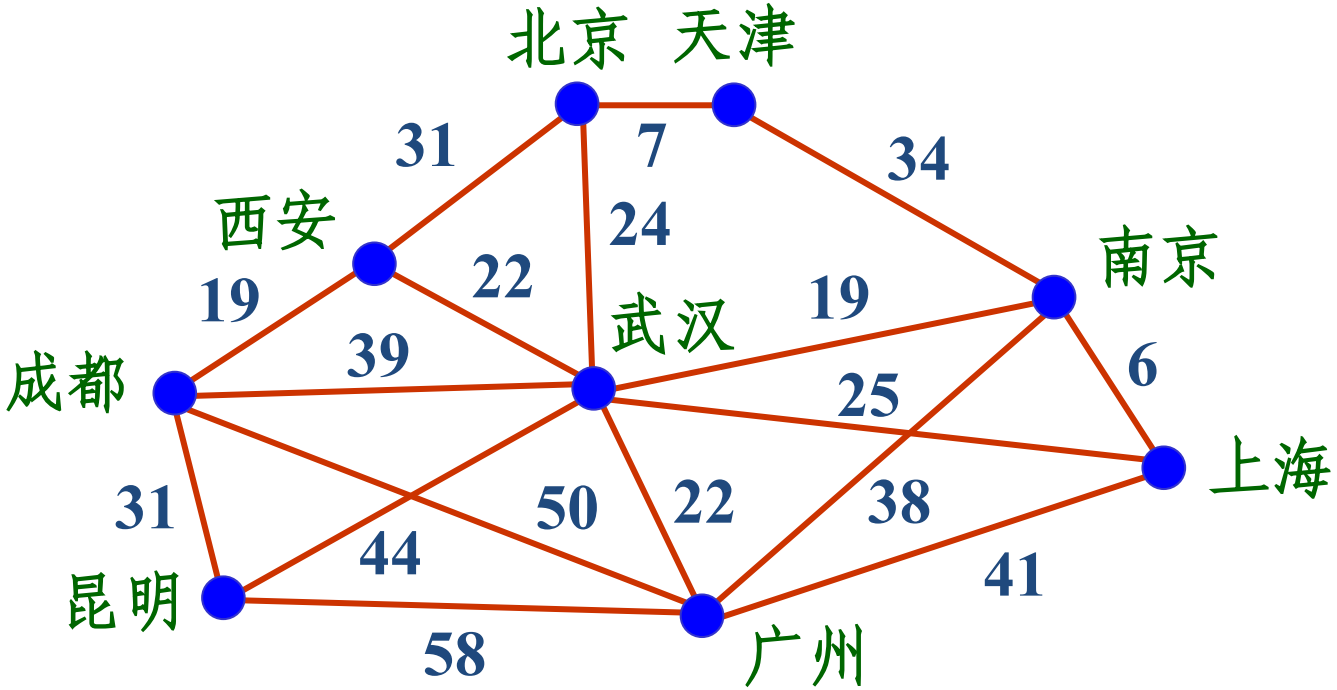
确定连通分量的算法

```
template <class T, class E>
void Components (Graph<T, E>& G) {
//通过DFS，找出无向图的所有连通分量
   int i, n = G.NumberOfVertices();          //图中顶点个数
   bool *visited = new bool[n];              //访问标记数组
   for (i = 0; i < n; i++) visited[i] = false;
   for (i = 0; i < n; i++)                   //扫描所有顶点
      if (!visited[i]) {                     //若没有访问过
         DFS (G, i, visited);                //访问
         OutputNewComponent();               //输出连通分量
      }
   delete [ ] visited;
}
```

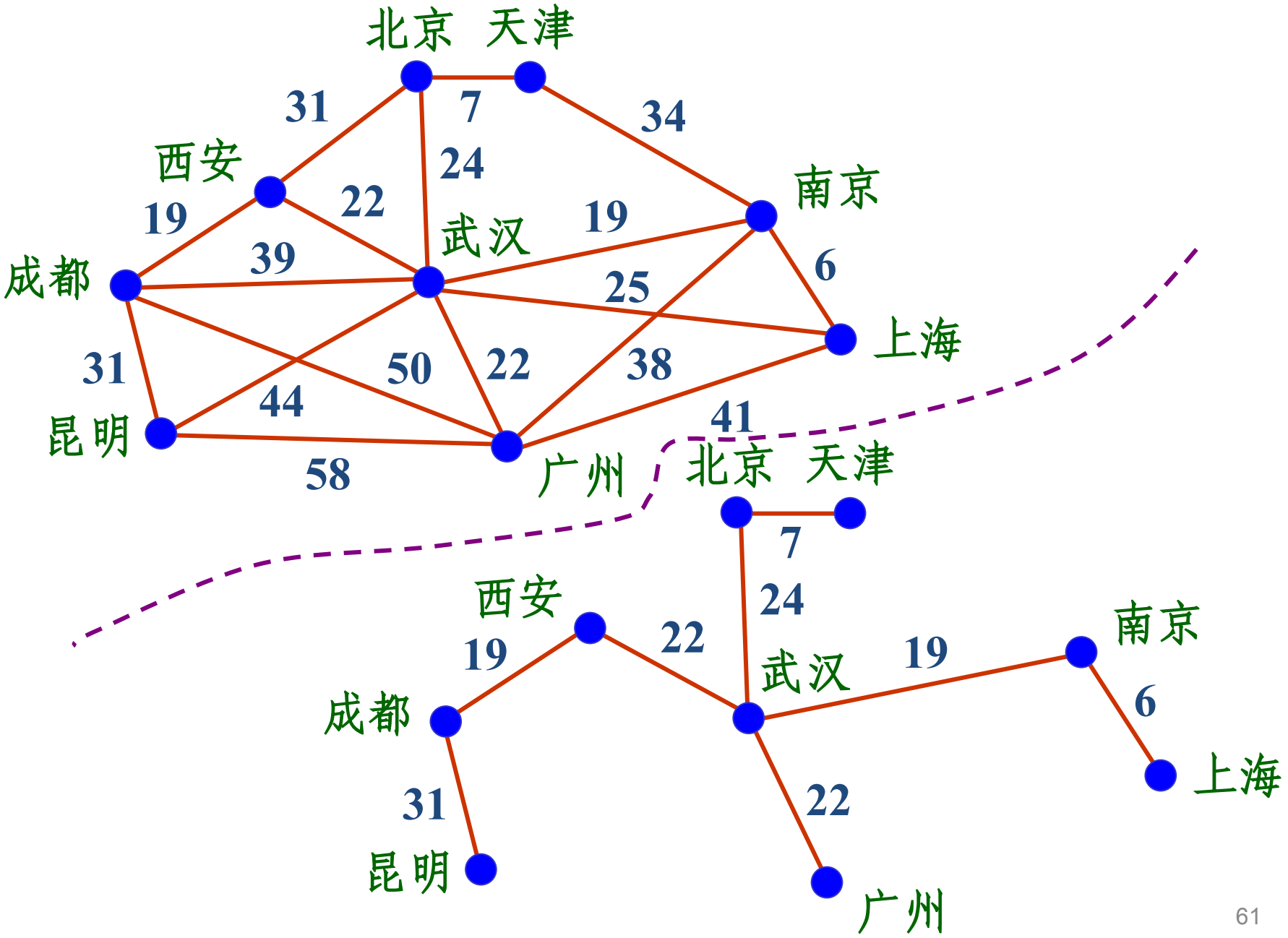
最小生成树
(minimum cost spanning tree)

- 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

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



- 构造最小生成树的准则
  - ❖ 必须使用且仅使用该网络中的 $n-1$ 条边来连接网络中的 n 个顶点；
  - ❖ 不能使用产生回路的边；
  - ❖ 各边上的权值的总和达到最小。


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

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。

Kruskal算法的伪代码描述

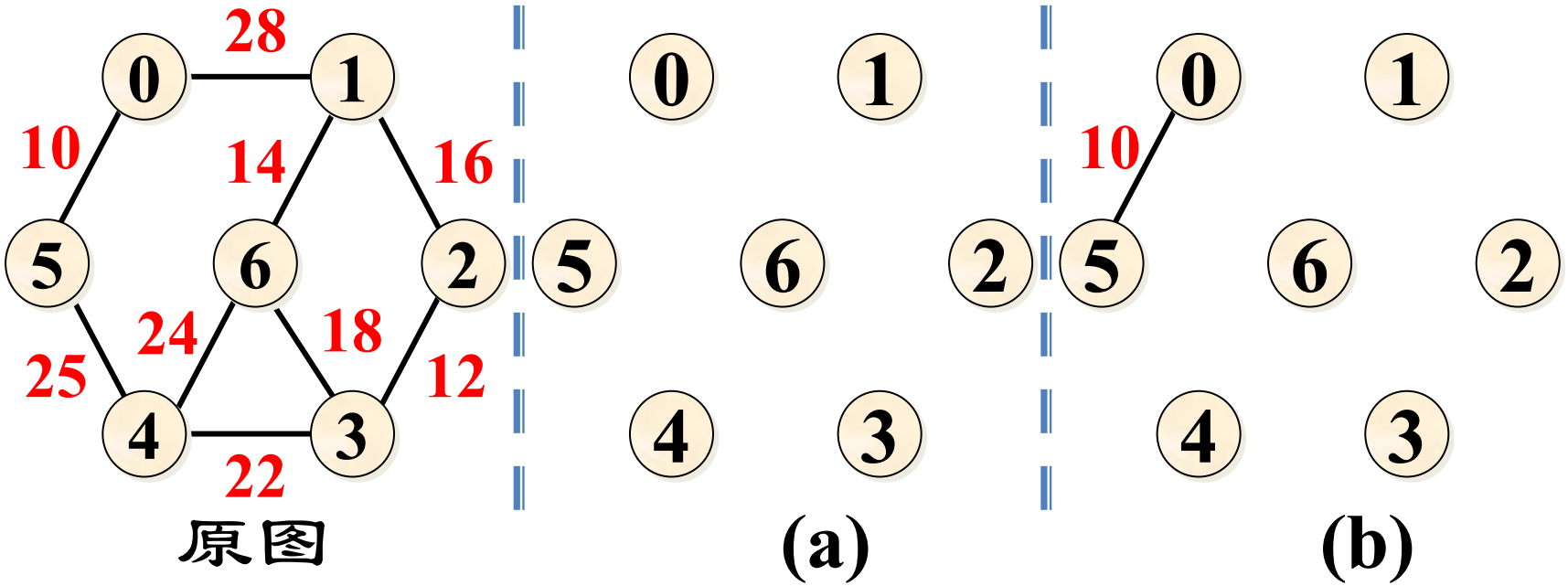
```
T = ∅;  //T是最小生成树的边集合
        //E是带权无向图的边集合
while ( T包含的边少于n-1 && E不空) {
    从 E 中选一条具有最小代价的边 (v, w);
    从 E 中删去(v, w);
    如果(v, w)加到 T 中后不会产生回路, 则将
            (v, w)加入T; 否则放弃(v, w);
}
if ( T 中包含的边少于n-1条)
    cout << "不是最小生成树" << endl;
```

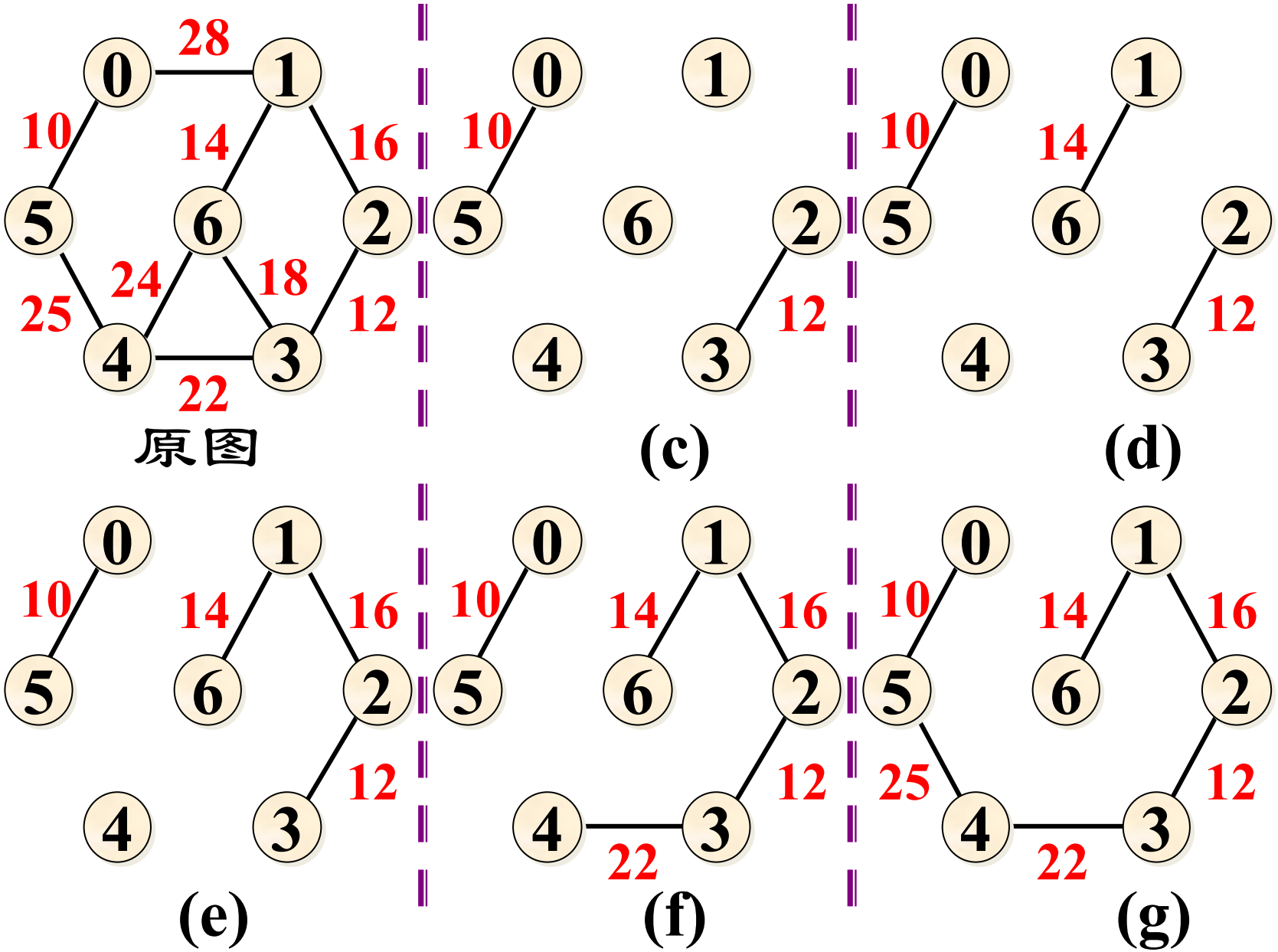
- 算法的框架　利用最小堆(MinHeap)和并查集(UFSets)来实现克鲁斯卡尔算法。
- 首先,利用最小堆来存放E中所有的边,堆中每个结点的格式为

- 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。

构造最小生成树的过程




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

最小生成树类定义

```
const float maxValue = FLOAT_MAX
  //机器可表示的、问题中不可能出现的大数
template <class T, class E>
struct MSTEdgeNode {                    //树边结点的类定义
    int tail, head;                     //两顶点位置
    E cost;                             //边上的权值
    MSTEdgeNode() : tail(-1), head(-1), cost(0) { }
                                        //构造函数
};
```

```
template <class T, class E>
class MinSpanTree {        //树的类定义
protected:
    MSTEdgeNode<T, E> *edgevalue;//边值数组
    int maxSize; //最大元素个数
    int currentSize; //当前元素个数
public:
    MinSpanTree (int sz = DefaultSize-1) :
            maxSize (sz), currentSize (0) {
        edgevalue = new MSTEdgeNode<T, E>[sz];
    }
    int Insert (MSTEdgeNode& item);
};
```

- 在求解最小生成树时，可以用邻接矩阵存储图，也可以用邻接表存储图。算法中使用图的抽象基类的操作，无需考虑图及其操作的具体实现。

Kruskal算法的实现

```
#include "heap.h"
#include "UFSets.h"
template <class T, class E>
void Kruskal (Graph<T, E>& G,
              MinSpanTree<T, E>& MST) {
```

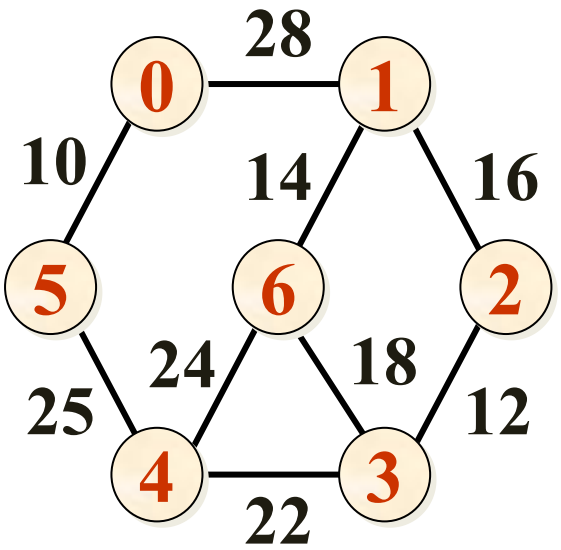
```
MSTEdgeNode<T, E> edge;          //边结点辅助单元
int u, v, count;
int n = G.NumberOfVertices();   //顶点数
int m = G.NumberOfEdges();      //边数
MinHeap <MSTEdgeNode<T, E>> H(m);   //最小堆
UFSets F(n);                          //并查集
for (u = 0; u < n; u++)
    for (v = u+1; v < n; v++)
        if (G.getWeight(u,v) != maxValue) {//插入堆
            edge.tail = u;  edge.head = v;
            edge.cost = G.getWeight (u, v);
            H.Insert(edge);
        }
```

```
    count = 1;                          //最小生成树边数计数
    while (count < n) {                 //反复执行, 取n-1条边
        H.Remove(edge);                 //退出具最小权值的边
        u = F.Find(edge.tail);  v = F.Find(edge.head);
            //取两顶点所在集合的根u与v
        if (u != v) {
         //不是同一集合, 不连通
            F.Union(u, v);              //合并, 连通它们
            MST.Insert(edge);           //该边存入MST
            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	(0,5,10)
	-2	-1	-2	2	-1	0	-1	(2,3,12)
	-2	-2	-2	2	-1	0	1	(1,6,14)
	-2	-4	1	2	-1	0	1	(1,2,16)
	-2	-5	1	2	1	0	1	(3,4,22)
	1	-7	1	2	1	0	1	(4,5,25)

并查集



原图

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集合U中，而另一个顶点不在集合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。

普里姆(Prim)的伪代码描述

选定构造最小生成树的出发顶点u_0;

$V_{mst} = \{u_0\}$, $E_{mst} = \varnothing$;

while (V_{mst}包含的顶点少于n && E不空) {

从E中选一条边(u, v),

$u \in V_{mst} \cap v \in V - V_{mst}$, 且具有最小代价(cost);

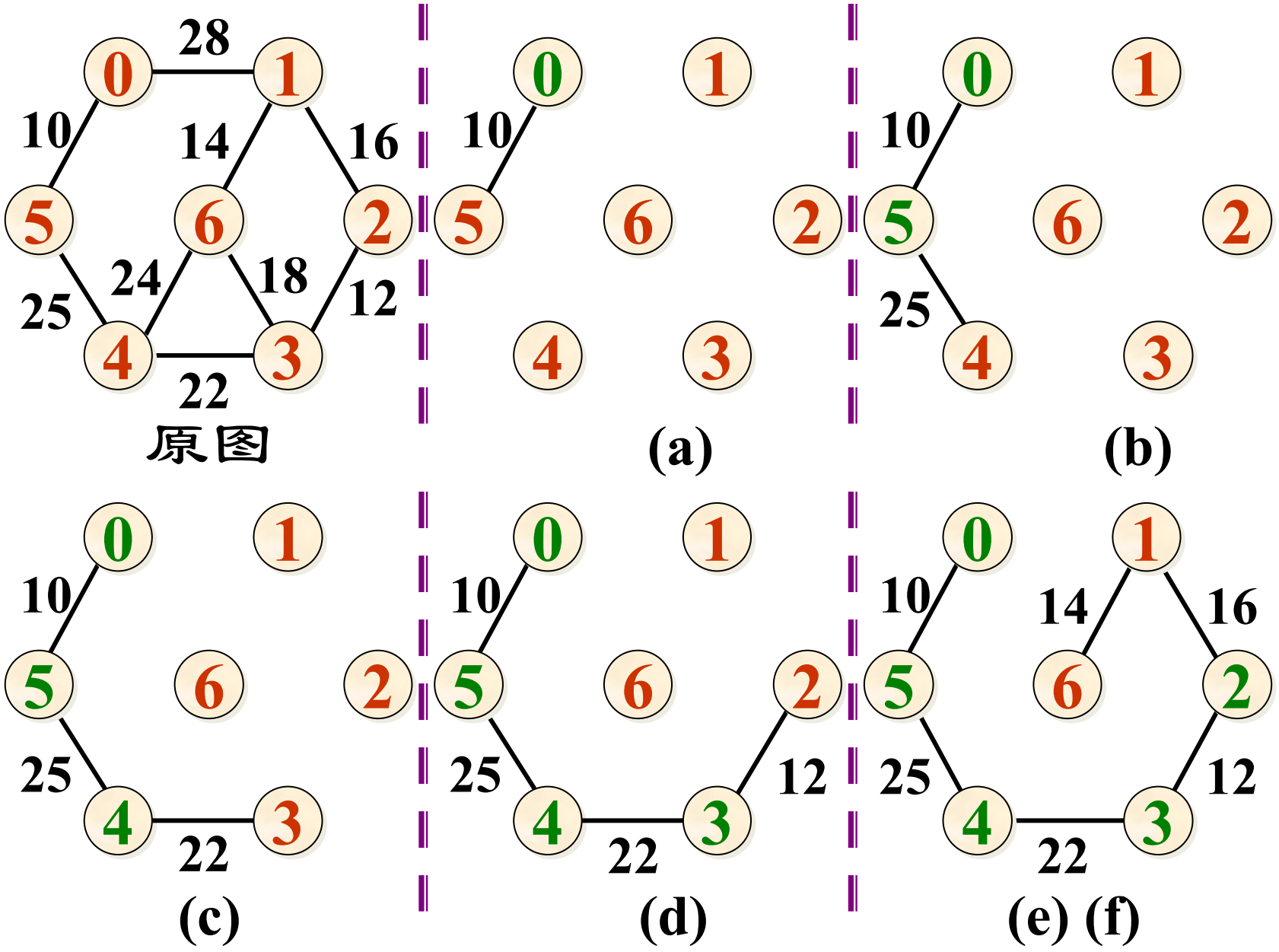
令$V_{mst} = V_{mst} \cup \{v\}$, $E_{mst} = E_{mst} \cup \{(u, v)\}$;

将新选出的边从E中剔除: $E = E - \{(u, v)\}$;

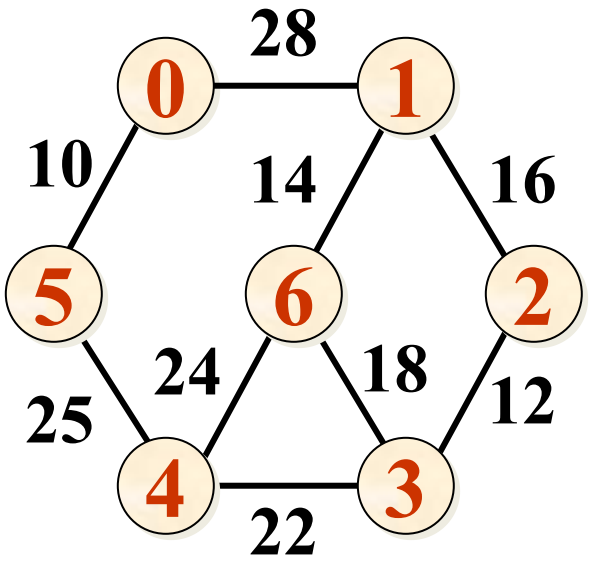
}

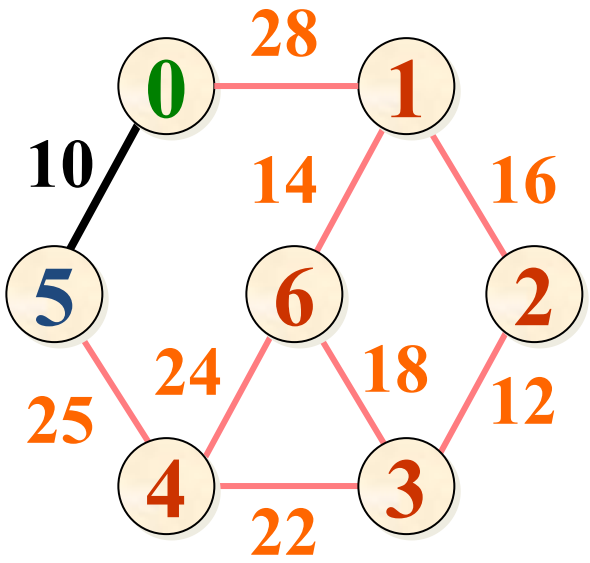
if (V_{mst}包含的顶点少于n)

cout << "不是最小生成树" << endl;


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

- 例子





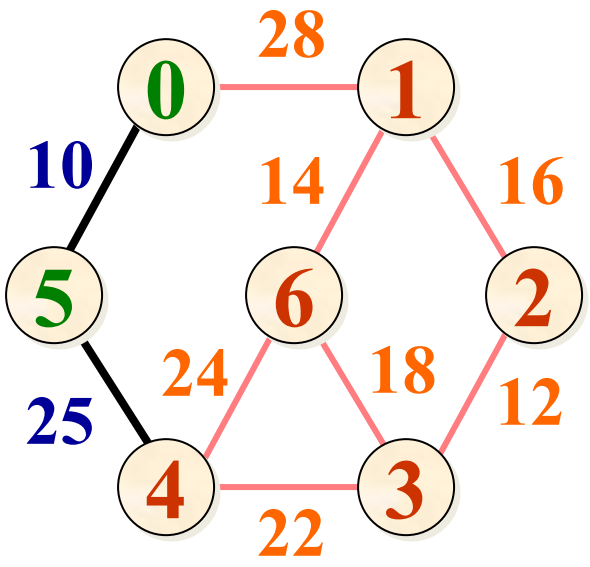
H = {(0,5,10), (0,1,28)}

edge = (0, 5, 10)

V_{mst} = {t, f, f, f, f, f, f}

⇩

V_{mst} = {t, f, f, f, f, t, f}

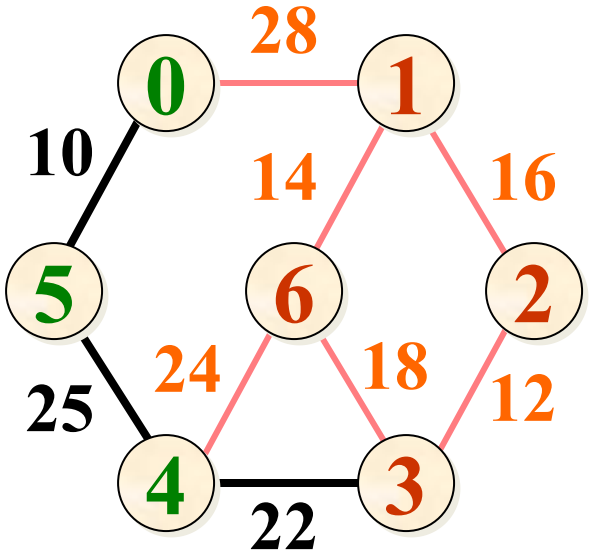


H = {(5,4,25), (0,1,28)}

edge = (5, 4, 25)

V_{mst} = {t, f, f, f, f, t, f}

V_{mst} = {t, f, f, f, t, t, f}

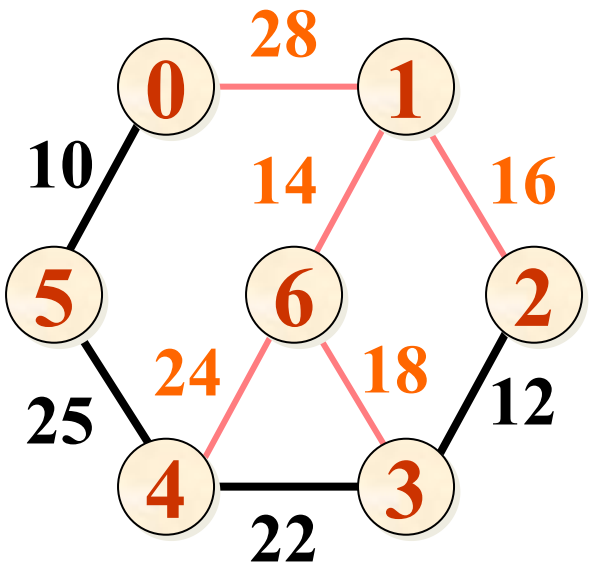


H = {(4,3,22), (4,6,24), (0,1,28)}

edge = (4, 3, 22)

V_{mst} = {t, f, f, f, t, t, f}

V_{mst} = {t, f, f, t, t, t, f}

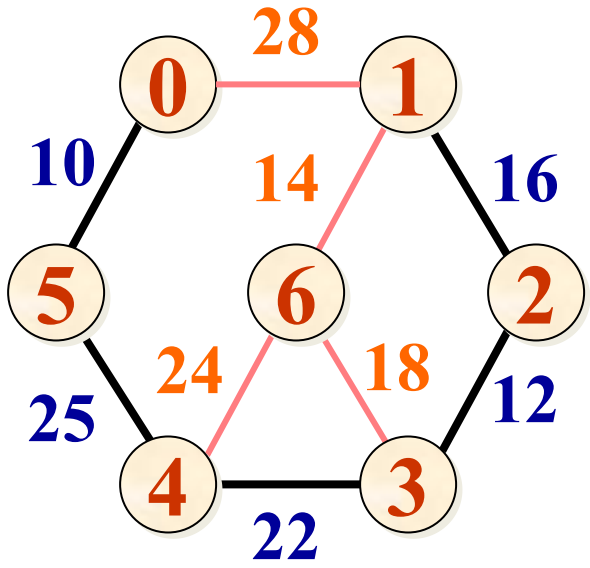


H = {(3,2,12), (3,6,18), (4,6,24), (0,1,28)}

edge = (3, 2, 12)

V_{mst} = {t, f, f, t, t, t, f}

V_{mst} = {t, f, t, t, t, t, f}

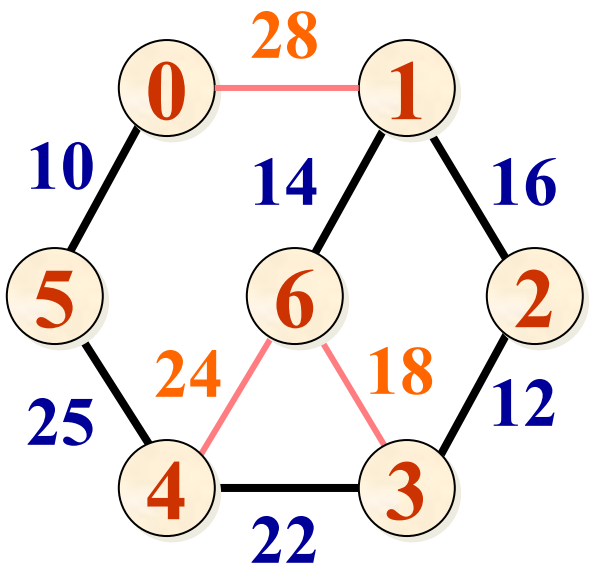


H = {(2,1,16), (3,6,18), (4,6,24), (0,1,28)}

edge = (2, 1, 16)

V_{mst} = {t, f, t, t, t, t, f}

V_{mst} = {t, t, t, t, t, t, f}



H = {(1,6,14), (3,6,18), (4,6,24),
(0,1,28)}

edge = (1, 6, 14)

V_{mst} = {t, t, t, t, t, t, f}

V_{mst} = {t, t, t, t, t, t, t}

- 最小生成树中边集合里存入的各条边为:

  (0, 5, 10), (5, 4, 25), (4, 3, 22),

  (3, 2, 12), (2, 1, 16), (1, 6, 14)

Prim算法的实现

```
#include "heap.h"
template <class T, class E>
void Prim (Graph<T, E>& G, const T u0,
        MinSpanTree<T, E>& MST) {
    MSTEdgeNode<T, E> edge; //边结点辅助单元
    int i, u, v, count;
    int n = G.NumberOfVertices();   //顶点数
    int m = G.NumberOfEdges();      //边数
    int u = G.getVertexPos(u0);                  //起始顶点号
    MinHeap <MSTEdgeNode<T, E>> H(m);  //最小堆
```

```
bool Vmst = new bool[n];   //最小生成树顶点集合
for (i = 0; i < n; i++) Vmst[i] = false;
Vmst[u] = true;                    //u 加入生成树
count = 1;
do {                               //迭代
    v = G.getFirstNeighbor(u);
    while (v != -1) {              //检测u所有邻接顶点
       if (!Vmst[v]) {             //v不在mst中
          edge.tail = u;  edge.head = v;
          edge.cost = G.getWeight(u, v);
          H.Insert(edge);          //(u,v)加入堆
       }    //堆中存所有u在mst中, v不在mst中的边
       v = G.getNextNeighbor(u, v);
    }
```

```
        while (!H.IsEmpty() && count < n) {
            H.Remove(edge);              //选堆中具最小权的边
            if (!Vmst[edge.head]) {
                MST.Insert(edge);        //加入最小生成树
                u = edge.head;  Vmst[u] = true;
                                         //u加入生成树顶点集合
                count++;
                break;
            }
        }
    } while (count < n);
};
```

- **Prim**算法适用于边稠密的网络。
- **Kruskal**算法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唯一。

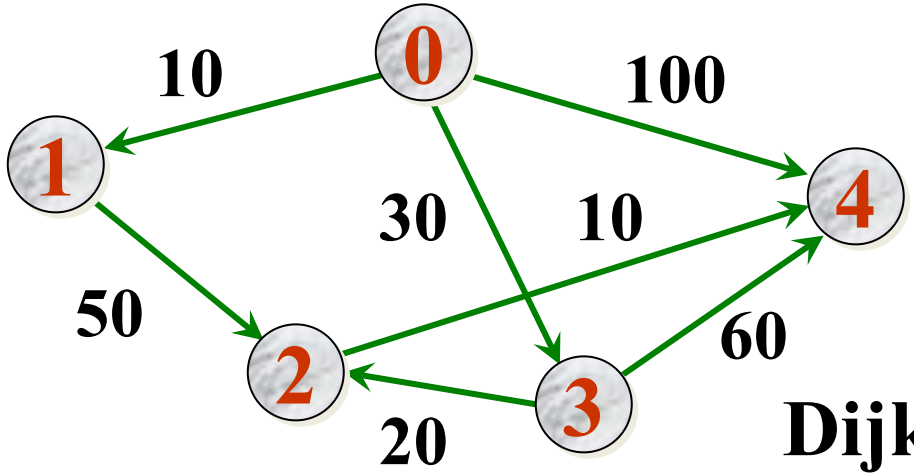
最短路径 (Shortest Path)

- 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** ×(不讲)
  - 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*中其他顶点的最短路径。 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。



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v_0到顶点v_i有边，则dist[i]为该边的权值；
  - 若从v_0到顶点v_i无边，则dist[i]为∞。
- 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 ($v_x \in V-S$)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，然后对所有的$v_i \in V-S$，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[u]←min {dist[i]}, $i \in V-S$;

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

③ 修改:

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

对于每一个 $k \in V-S$;

④ 判断：若 $S = V$, 则算法结束，否则转②。

计算从单个顶点到其他各顶点最短路径的算法

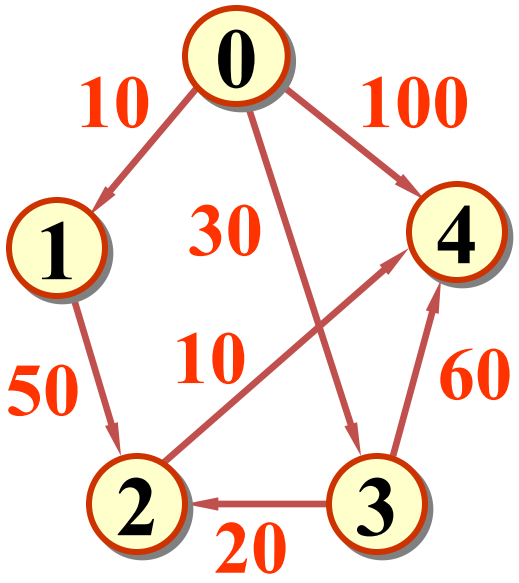
```
void ShortestPath (Graph<T, E>& G, T v,
        E dist[], int path[]) {
//Graph是一个带权有向图。dist[j], 0≤j<n, 是当前
//求到的从顶点v到顶点j的最短路径长度, path[j],
//0≤j<n, 存放求到的最短路径。
    int n = G.NumberOfVertices();
    bool *S = new bool[n];     //最短路径顶点集
    int i, j, k;  E w, min;
    for (i = 0; i < n; i++) {
        dist[i] = G.getWeight(v, i);
```

```
    S[i] = false;
    if (i != v && dist[i] < maxValue) path[i] = v;
    else path[i] = -1;
}
S[v] = true;  dist[v] = 0;          //顶点v加入顶点集合
for (i = 0; i < n-1; i++) {         //求解各顶点最短路径
    min = maxValue;  int u = v;
//选不在S中具有最短路径的顶点u
    for (j = 0; j < n; j++)
        if (!S[j] && dist[j] < min)
            { u = j; min = dist[j];}
    S[u] = true;                    //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
                dist[k] = dist[u]+w;
                path[k] = u;                    //修改到k的最短路径
            }
        }
    }
};
```

Dijkstra算法中各辅助数组的最终结果

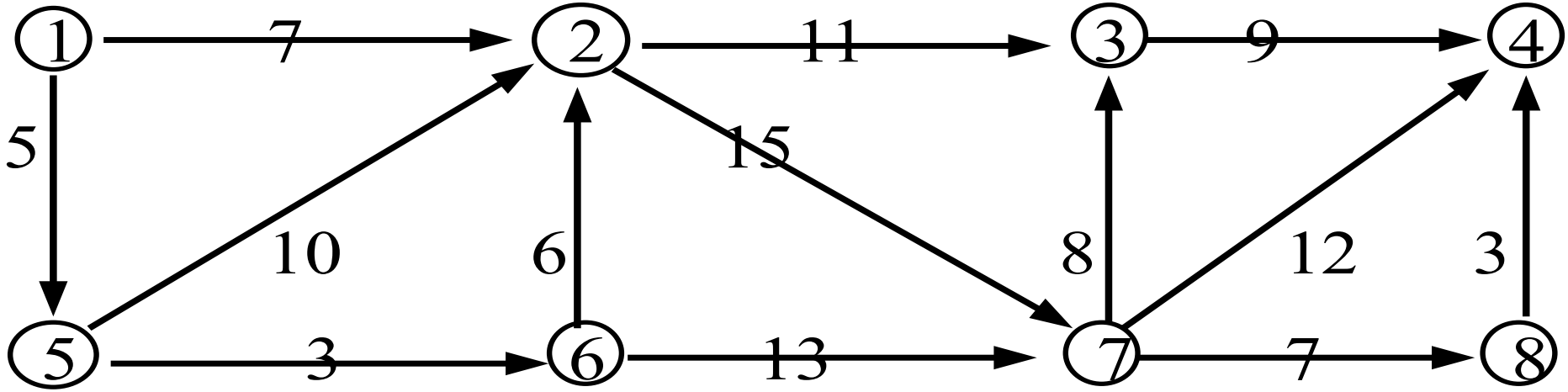
序号	顶点1	顶点2	顶点3	顶点4
Dist	10	50	30	60
path	0	3	0	2



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。

求：源点1到所有点的最短路径



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 $A^{(-1)}[i][j] = \mathbf{Edge}[i][j]$;

$$A^{(k)}[i][j] = \min\{A^{(k-1)}[i][j],\ A^{(k-1)}[i][k] + A^{(k-1)}[k][j]\},\ k = 0,1,\ldots,n-1$$

- $A^{(0)}[i][j]$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。

求各对顶点间最短路径的算法

```
template <class T, class E>
void Floyd (Graph<T, E>& G, E 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。
```

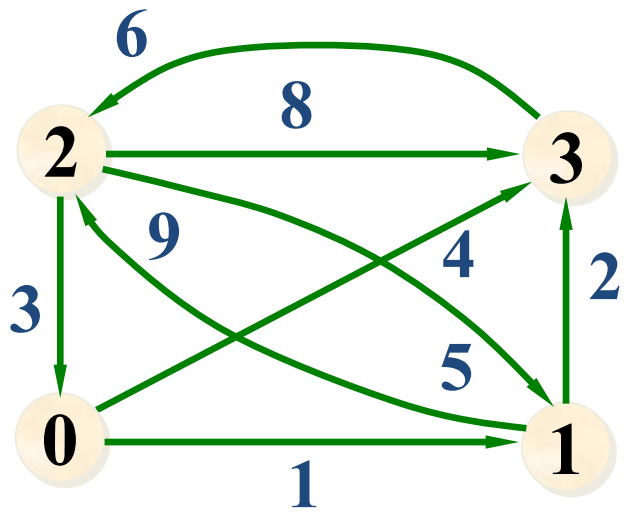
```
for (i = 0; i < n; i++)                    //矩阵a与path初始化
    for (j = 0; j < n; j++) {
        a[i][j] = G.getWeight (i, j);
            if (i != j && a[i][j] < maxValue) path[i][j] = i;
            else path[i][j] = 0;
    }
for (k = 0; k < n; k++)
  //针对每一个k, 产生a(k)及path(k)
    for (i = 0; i < 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。

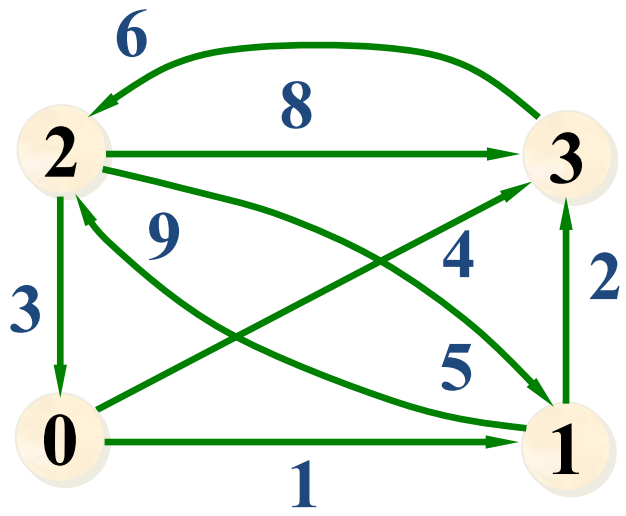
	$A^{(-1)}$				$A^{(0)}$				$A^{(1)}$				$A^{(2)}$				$A^{(3)}$			
	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3
0	0	1	∞	4	0	1	∞	4	0	1	10	3	0	1	10	3	0	1	9	3
1	∞	0	9	2	∞	0	9	2	∞	0	9	2	12	0	9	2	11	0	8	2
2	3	5	0	8	3	4	0	7	3	4	0	6	3	4	0	6	3	4	0	6
3	∞	∞	6	0	∞	∞	6	0	∞	∞	6	0	9	10	6	0	9	10	6	0

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



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



- 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。

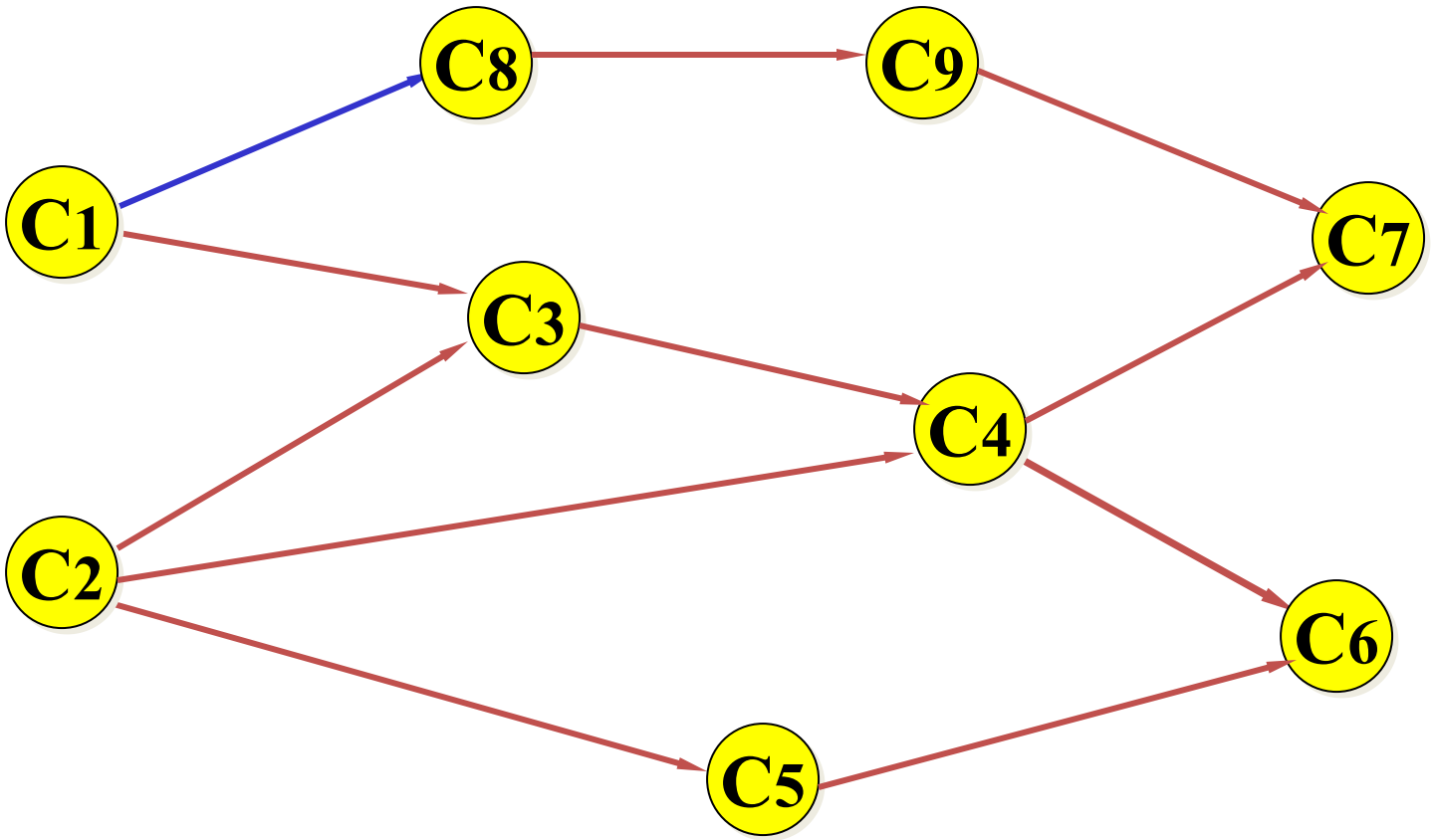
活动网络 (Activity Network)

用顶点表示活动的网络
(activity on vertices, 即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。

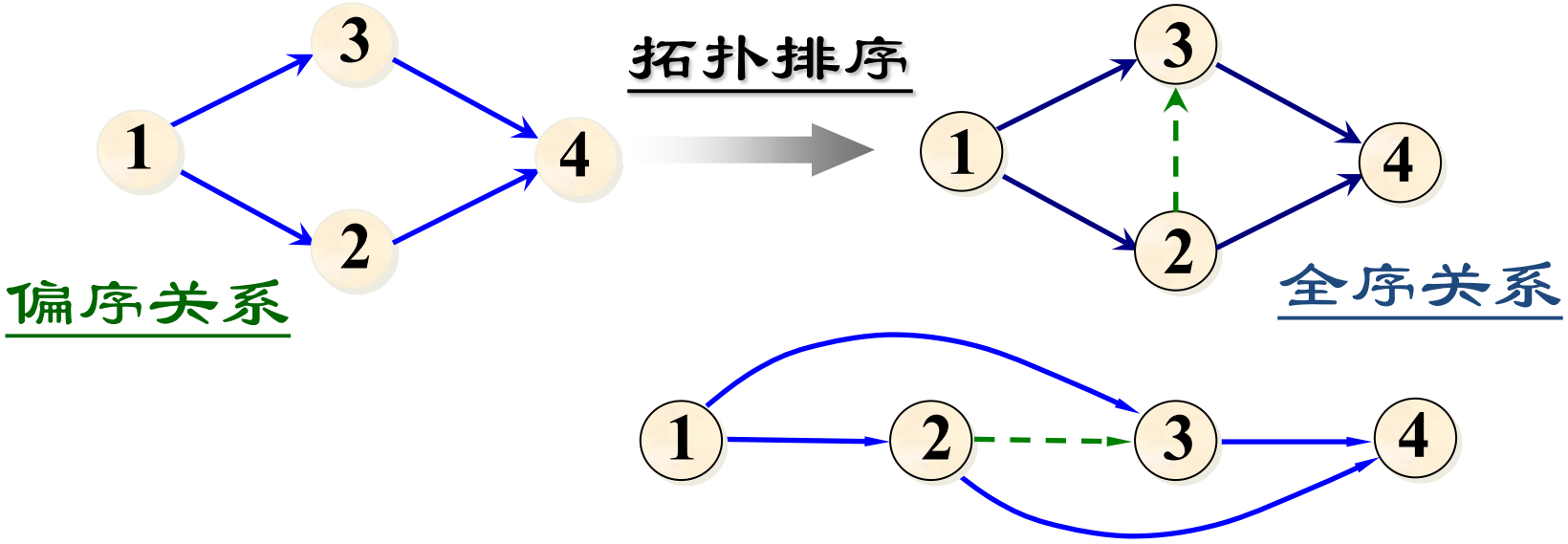
课程代号	课程名称	先修课程
C_1	高等数学	
C_2	程序设计基础	
C_3	离散数学	C_1, C_2
C_4	数据结构	C_3, C_2
C_5	高级语言程序设计	C_2
C_6	编译方法	C_5, C_4
C_7	操作系统	C_4, C_9
C_8	普通物理	C_1
C_9	计算机原理	C_8



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。

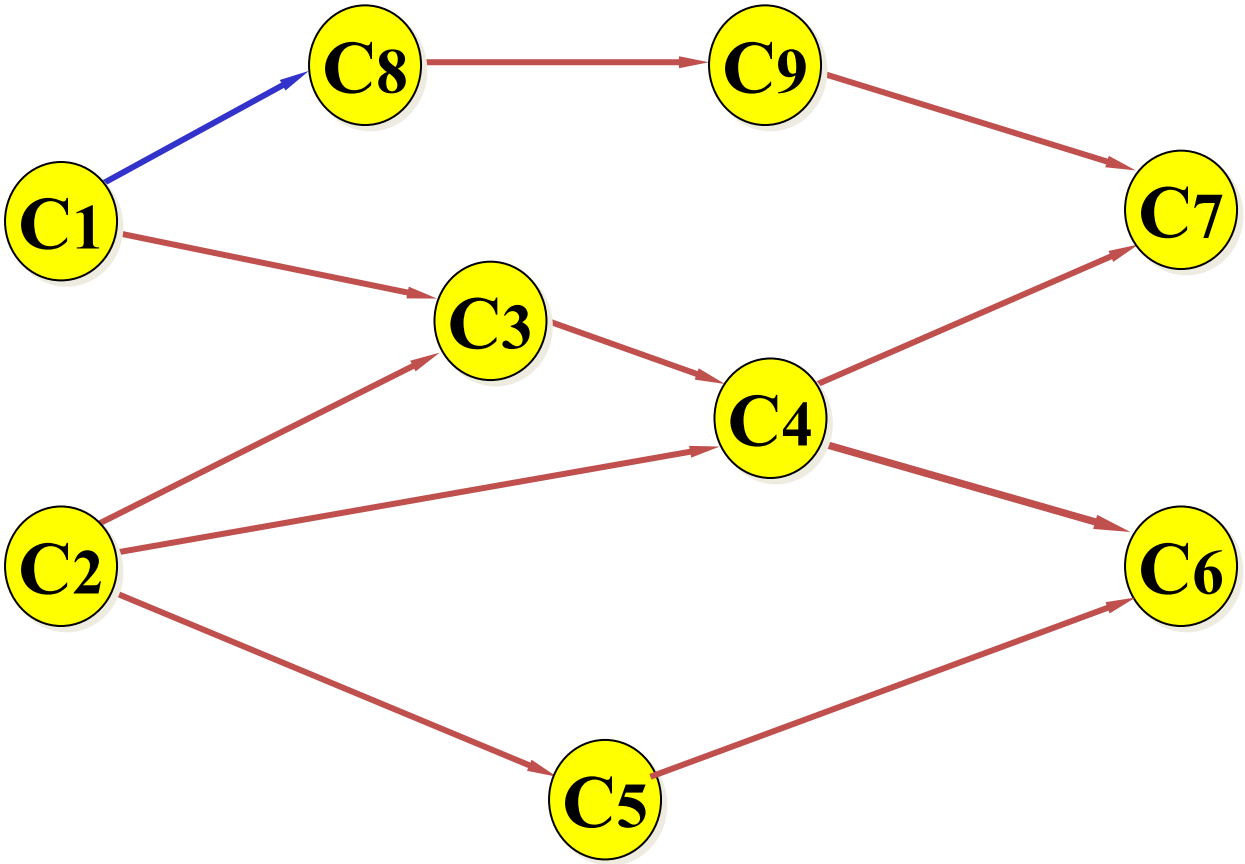
- 如果通过拓扑排序能将AOV网络的所有顶点都排入一个拓扑有序的序列中，则该网络中必定不会出现有向环。

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

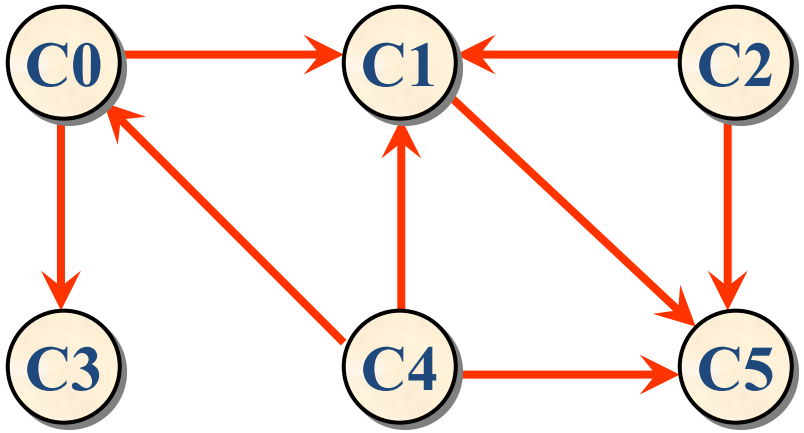


进行拓扑排序的方法

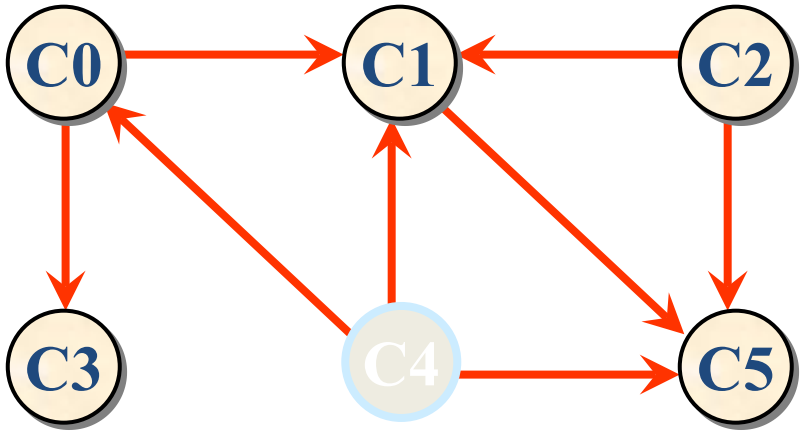
① 输入AOV网络。令 n 为顶点个数。
② 在AOV网络中选一个没有直接前驱的顶点，并输出之；
③ 从图中删去该顶点，同时删去所有它发出的有向边；
④ 重复以上②、③步，直到

- 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。

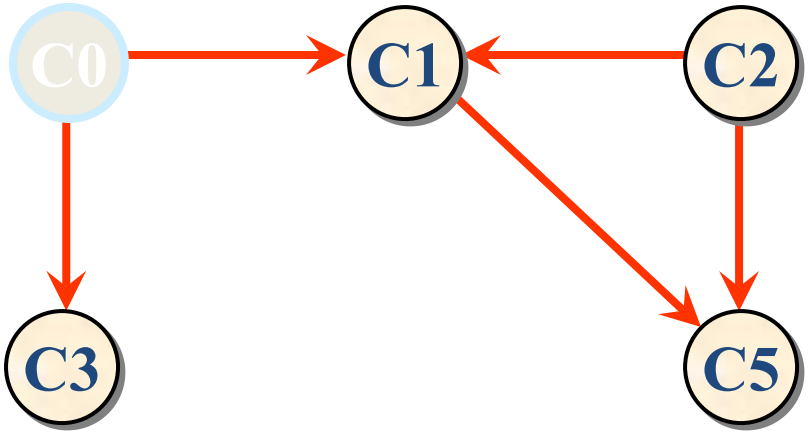
拓扑排序的过程



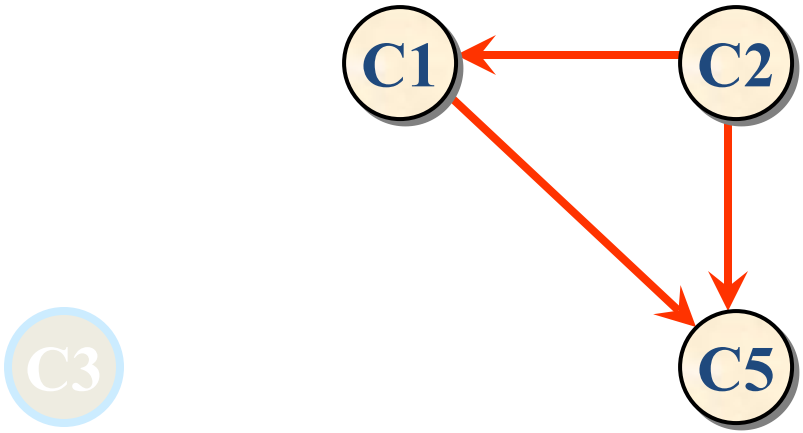
(a) 有向无环图



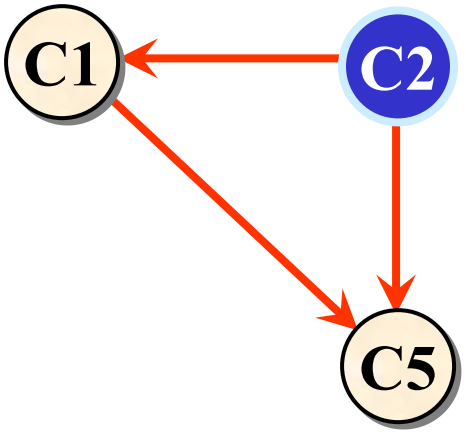
(b) 输出顶点C4



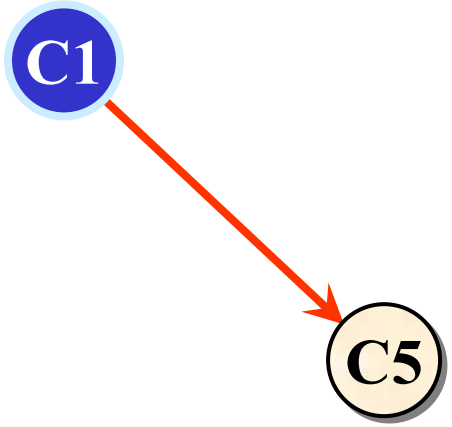
(c) 输出顶点C0



(d) 输出顶点C3



(e) 输出顶点C2

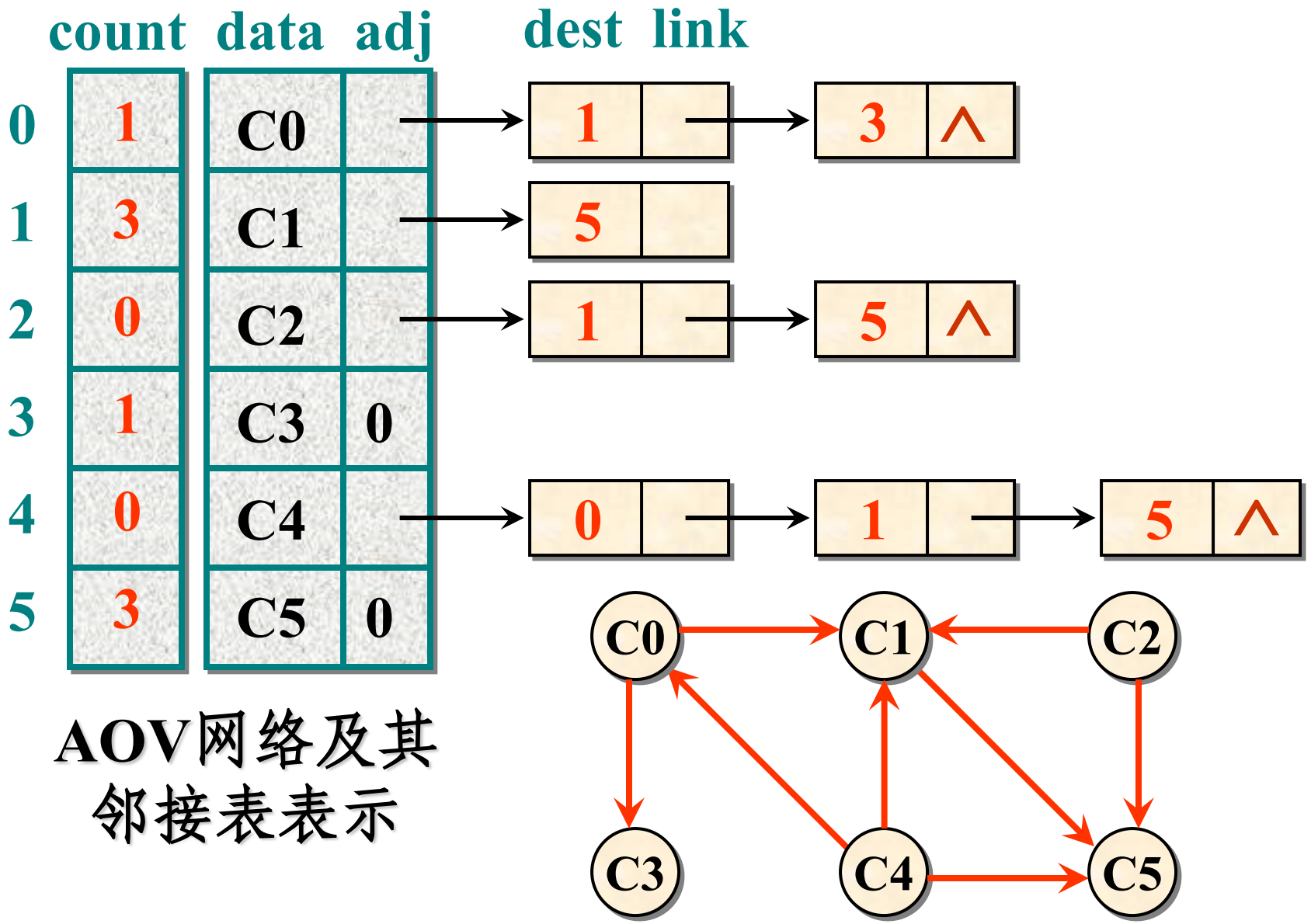


(f) 输出顶点C1

(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，也排出了先后次序关系。



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。

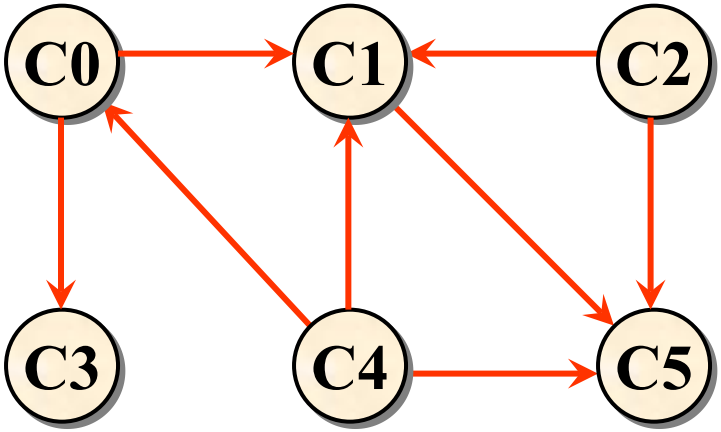
- 在算法实现时，为了建立入度为零的顶点栈，可以不另外分配存储空间，直接利用入度为零的顶点的count[ ]数组元素。设立一个栈顶指针top，指示当前栈顶位置，即某一个入度为零的顶点。栈初始化时置top = -1。

- 将顶点i 进栈时执行以下指针的修改：

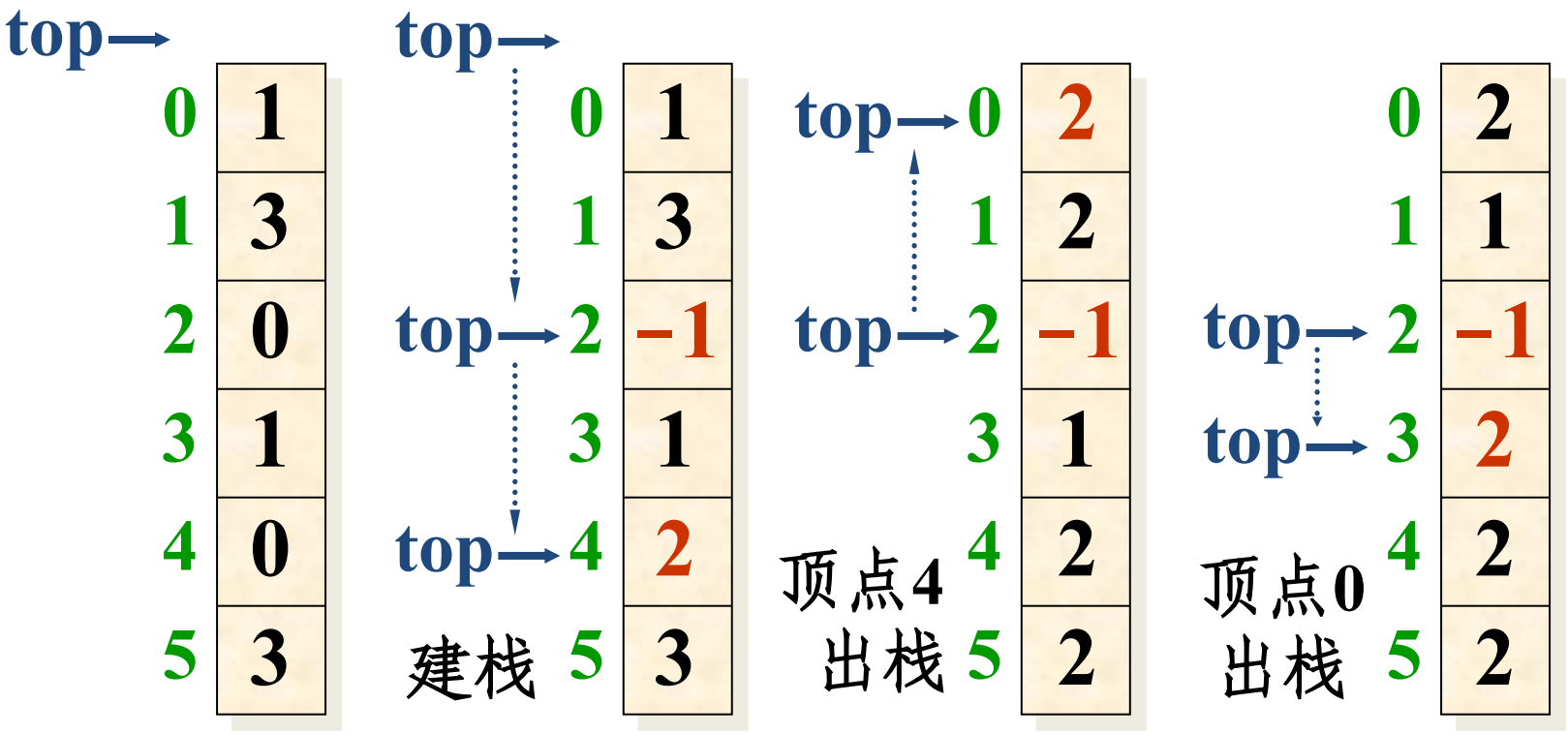
```
count[i] = top;  top = i ;
// top指向新栈顶i, 原栈顶元素序号在count[i]中
```

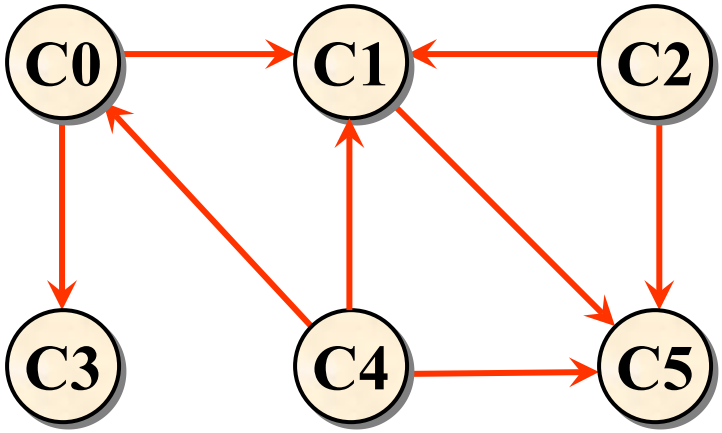
- 退栈操作可以写成：

```
j = top;  top = count[top];
//位于栈顶的顶点记于 j, top退到次栈顶
```

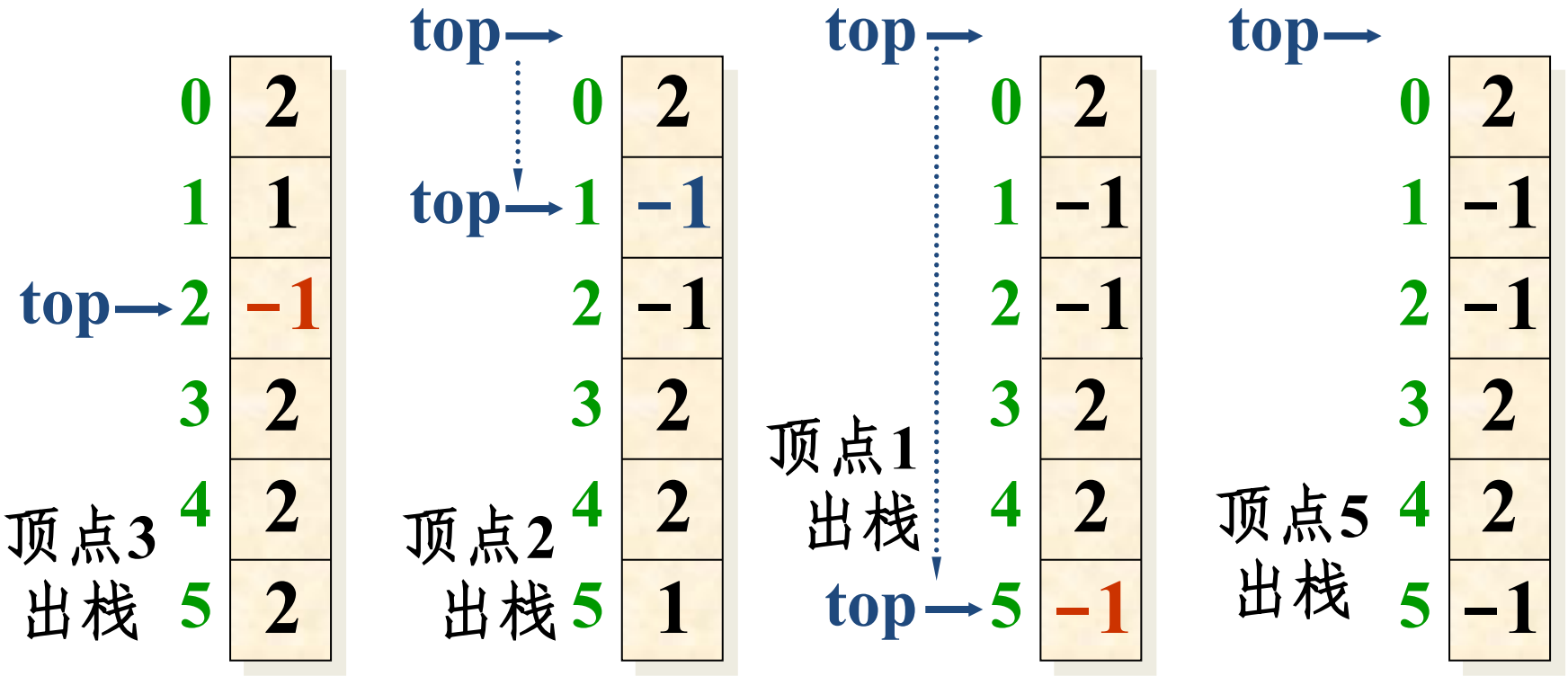


拓扑排序时入度为零的顶点栈在count[]中的变化





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

```
template <class T, class E>
void TopologicalSort (Graph<T, E>& G) {
    int i, j, w, v;
    int top = -1;              //入度为零的顶点栈初始化
    int n = G.NumberOfVertices();   //网络中顶点个数
    int *count = new int[n];
                               //入度数组兼入度为零的顶点栈
    for (i = 0; i < n; i++) count[i] = 0;
    cin >> i >> j;             //输入一条边(i, j)
    while (i > -1 && i < n && j > -1 && j < n) {
        G.insertEdge (i, j);  count[j]++;
```

```
    cin >> i >> j;
}
for (i = 0; i < n; i++)      //检查网络所有顶点
    if (count[i] == 0)       //入度为零的顶点进栈
        { 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
        v = top;  top = count[top];          //退栈v
```

```
        cout << G.getValue(v) << "  " << endl;   //输出
        w = G.GetFirstNeighbor(v);
        while (w != -1) {                  //扫描顶点v的出边表
            count[w]--;                    //邻接顶点入度减一
            if (count[w]==0)               //入度减至零, 进栈
              { count[w] = top;  top = w; }
            w = G.GetNextNeighbor (v, w);
        }    //一个顶点输出后，调整其邻接顶点入度
    }     //输出一个顶点，继续for循环
};
```

- 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。

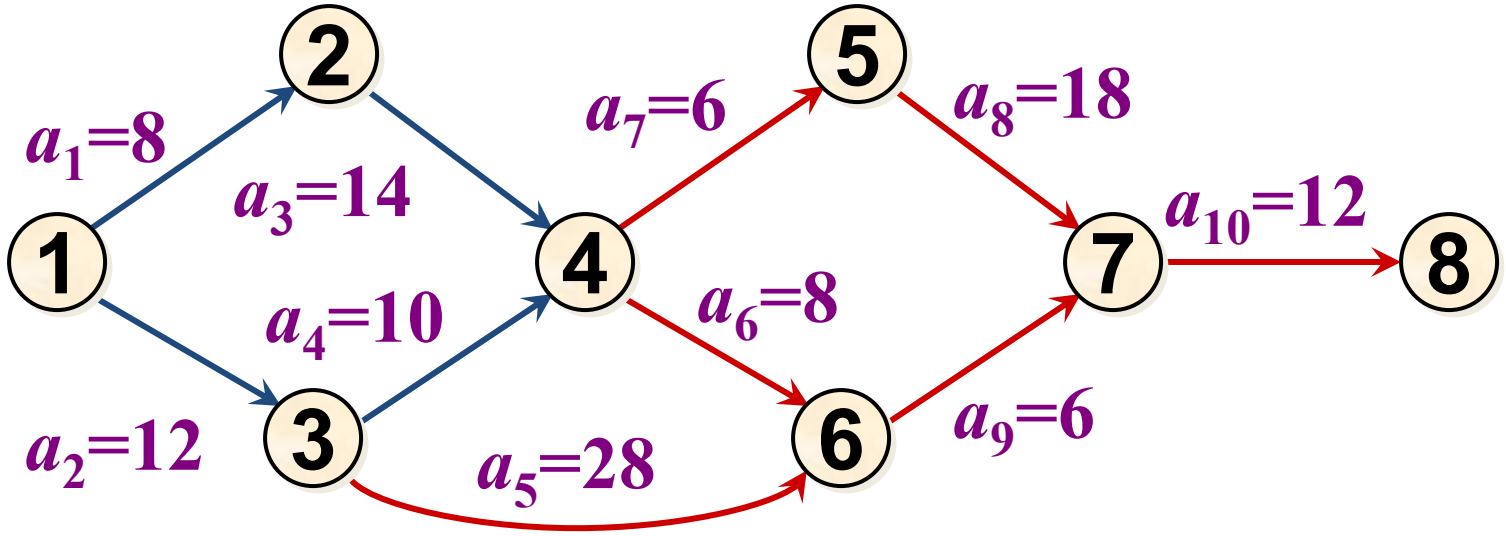
- AOE网络在某些工程估算方面非常有用。例如, 可以使人们了解:
  - 完成整个工程至少需要多少时间(假设网络中没有环)?
  - 为缩短完成工程所需的时间, 应当加快哪些活动?

- 整个工程只有一个开始点和一个完成点，开始点（即入度为零的顶点）称为源点，结束点（即出度为零的顶点）称为汇点。

- 从源点到各个顶点、以及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$Ae[k]$：
   设活动a_k在边$<V_i, V_j>$上，则$Ae[k]$是从源点V_0到顶点V_i的最长路径长度。因此，

   $$Ae[k] = Ve[i]。$$

4. 活动a_k的最迟允许开始时间$A\ l[k]$：

$Al[k]$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。

$Al[k] = Vl[j] - dur(<i, j>)$。

其中，$dur(<i, j>)$是完成 a_k 所需的时间。

5. 时间余量 $Al[k] - Ae[k]$：

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。$Al[k] == Ae[k]$表示活动 a_k 是没有时间余量的关键活动。

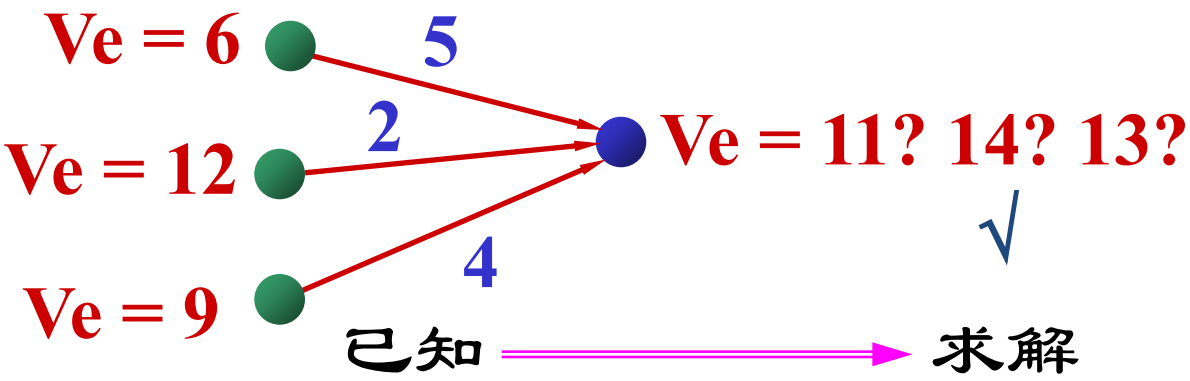
- 为找出关键活动，需要求各个活动的 $Ae[k]$ 与 $Al[k]$，以判别是否 $Al[k] == Ae[k]$。

- 为求得$Ae[k]$与$Al[k]$，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

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

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

$S2$ 是所有指向V_j的有向边$<V_i,V_j>$的集合。

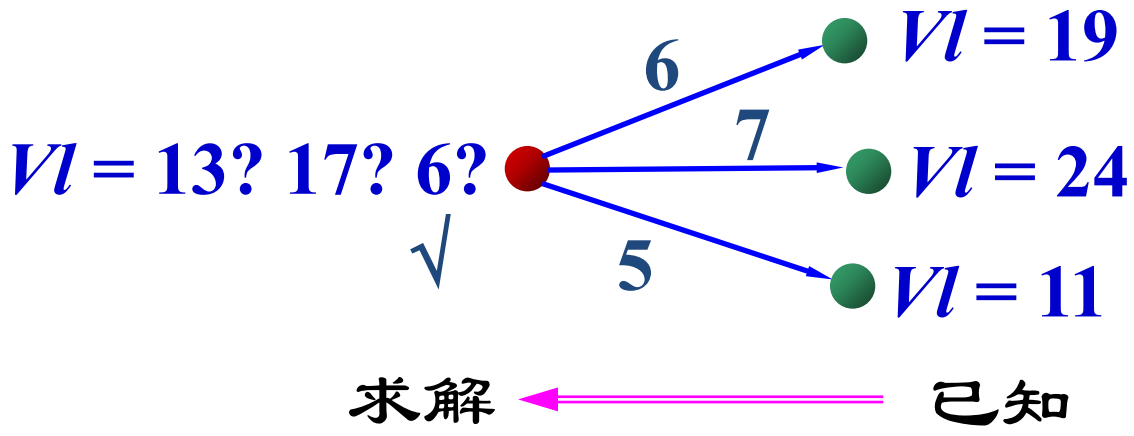


- 从$Vl[n\text{-}1] = Ve[n\text{-}1]$开始，反向递推

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

$$<V_j, V_k> \in S1, j = n\text{-}2, n\text{-}3, \ldots, 0$$

$S1$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

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

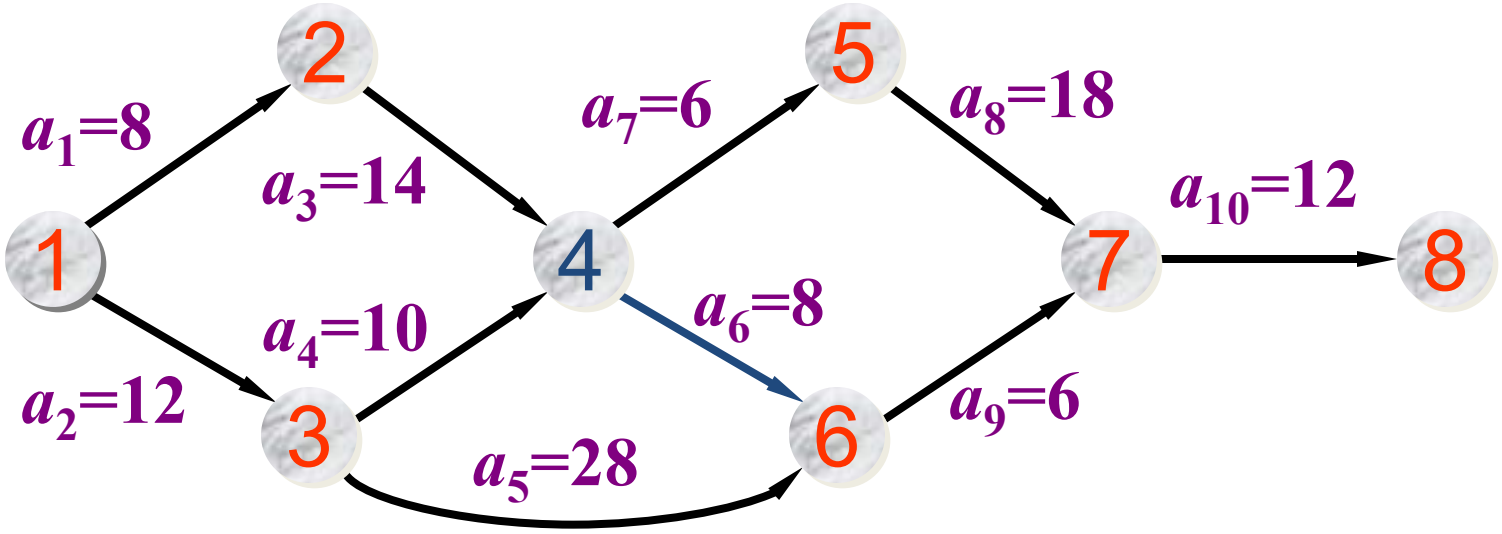
  $Al[k] = Vl[j] - dur(<V_i,V_j>)$; $k = 1, 2, \ldots, e$。

  这样就得到计算关键路径的算法。

- 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。

	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
Ae	0	0	8	12	12	22	22	28	40	46
Al	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

```
template <class T, class E>
void CriticalPath(Graph<T, E>& G) {
    int i, j, k;    E Ae, Al, dur;
    int n = G.NumberOfVertices();
    E *Ve = new E[n];  E *Vl = new E[n];
    for (i = 0; i < n; i++) Ve[i] = 0;
     for (i = 0; i < n; i++) {  //正向计算Ve[]，假设已经
    按拓扑排序从0到n-1编号
        j = G.getFirstNeighbor(i);
        while (j != -1) {
            dur = G.getWeight (i, j);
```

```
        if (Ve[i]+dur > Ve[j]) Ve[j] = Ve[i]+dur;
        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.getFirstNeighbor (j);
    while (k != -1) {
        dur = G.getWeight (j, k);
        if (Vl[k]-dur < Vl[j]) Vl[j] = Vl[k]-dur;
        k = G.getNextNeighbor (j, k);
    }
}
```

```
    for (i = 0; i < n; i++) {          //求各活动的Ae, Al
        j = G.getFirstNeighbor (i);
        while (j != -1) {
            Ae = Ve[i];  Al = Vl[j]-G.getWeight(i, j);
            if (Al == Ae)
                cout << "<" << G.getValue(i) << ","
                     << G.getValue(j) << ">"
                     << "是关键活动" << endl;
            j = G.getNextNeighbor (i, j);
        }
    }
    delete [] Ve;  delete [] Vl;
};
```

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 *Ae*[*k*] 和 *Al*[*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。
- 如果任一关键活动延迟，整个工程就要延迟。