Chapter8 图

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

定义：有向图、无向图、完全图、邻接顶点、子图、权、网络（带权图）、顶点的度、入度、出度、路径、路径长度、简单路径、回路

连通图与连通分量：**在无向图中,** 若从顶点 v_1 到顶点 v_2 有路径, 则称顶点 v_1 与 v_2 是连通的。如果图中任意一对顶点都是连通的, 则称此图是连通图。非连通图的**极大**连通子图叫做连通分量。 **（说连通图指的是无向图）**

强连通图与强连通分量：在有向图中, 若对于每一对顶点 v_i 和 v_j , 都存在一条从 v_i 到 v_j 和从 v_j 到 v_i 的路径, 则称此图是强连通图。非强连通图的**极大**强连通子图叫做强连通分量。 **（说强连通图指的是有向图）**

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。

	树	图
类别	层次	网络
顶点、边数	n节点，n-1边	n节点，边不确定
起始节点	根节点	—
有无回路	无回路	可能有
连通性	连通	不一定连通
遍历方式	前序中序后序，层次	DFS，BFS

8.2 图的存储结构

8.2.1 图的抽象数据类型

```
class Graph {
//对象: 由一个顶点的非空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
};

```

8.2.2 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 改为 。

```
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
    int getVertexPos (T vertex);//给出顶点vertex在图中位置
}
```

8.2.3 邻接矩阵 (Adjacency Matrix)

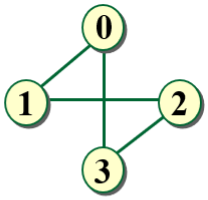
在图的邻接矩阵表示中，有一个记录各个顶点信息的顶点表，还有一个表示各个顶点之间关系的邻接矩阵。

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

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

无向图的邻接矩阵是对称的;

有向图的邻接矩阵可能是不对称的



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

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

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

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

用邻接矩阵表示的图的类定义

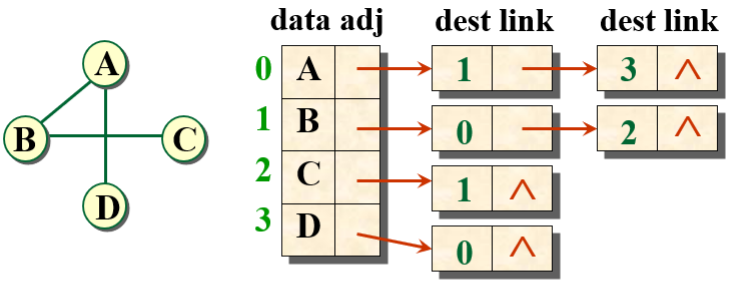
```
template <class T, class E>
class Graphmtx : public Graph<T, E> {
private:
    T *VerticesList;              //顶点表
    E **Edge;                     //邻接矩阵
    int getVertexPos (T vertex)  //给出顶点vertex在图中的位置
    {
        for (int i = 0; i < numVertices; i++)
            if (VerticesList[i] == Vertex)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

8.2.4 邻接表 (Adjacency List)

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。

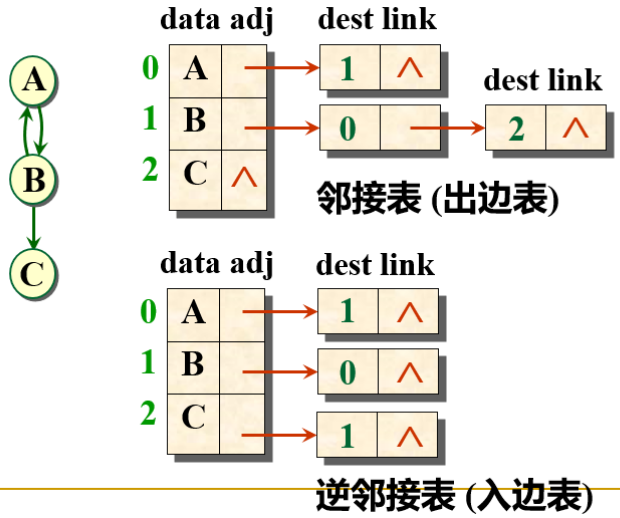
在邻接表中，同一个顶点发出的边链接在同一个边链表中，每一个链结点代表一条边（边结点），结点中有另一顶点的下标 dest 和指针 link。**对于带权图，边结点中还要保存该边的权值cost。**

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。



- 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



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。

8.2.5 邻接多重表 (Adjacency Multilist)

无向图的情形

- **边结点的结构**

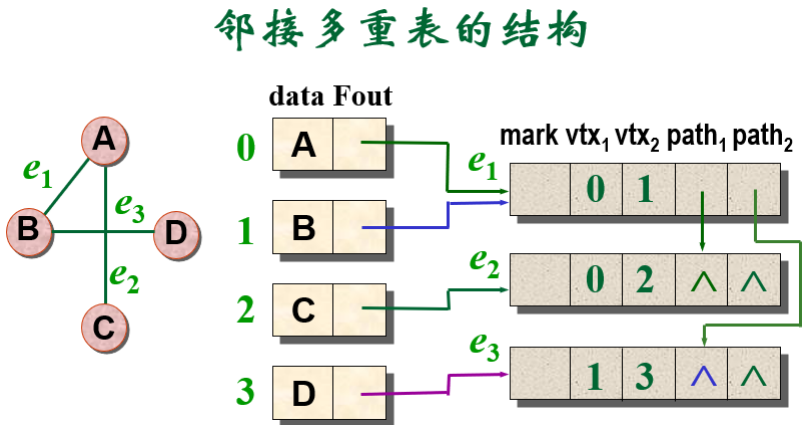
mark	vertex1	vertex2	path1	path2

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**

data	Firstout



有向图的情形

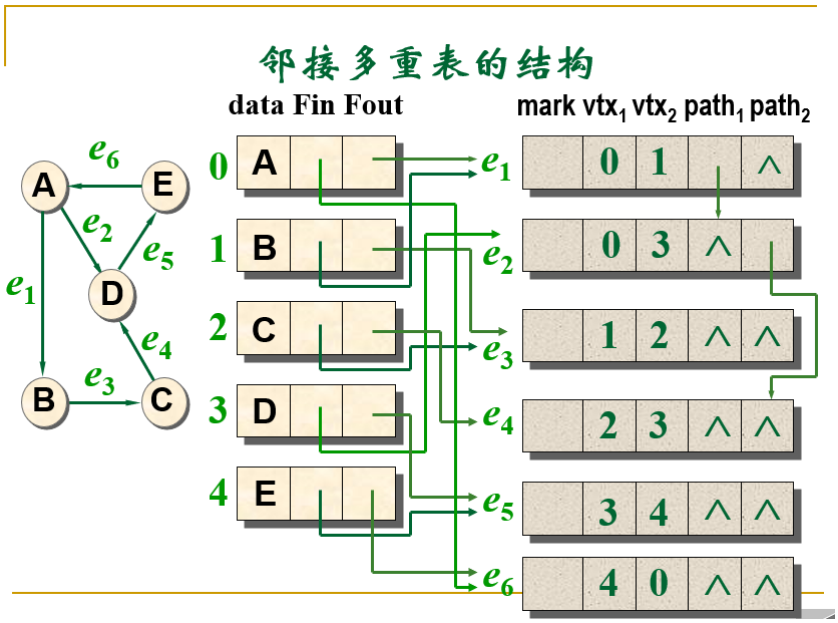
- **边结点的结构**

mark	vertex1	vertex2	path1	path2

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



8.3 图的遍历与连通性

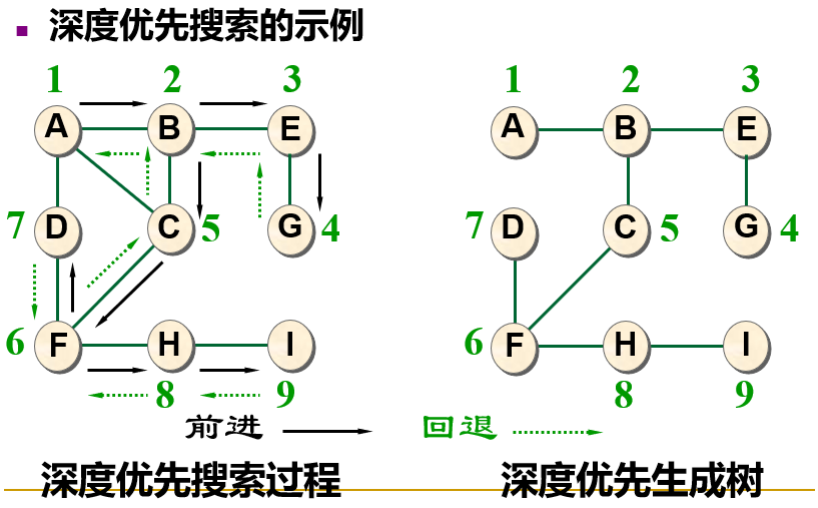
从已给的连通图中某一顶点出发，沿着一些边访遍图中所有的顶点，且使每个顶点仅被访问一次，就叫做**图的遍历 (Graph Traversal)**

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 [ ]**

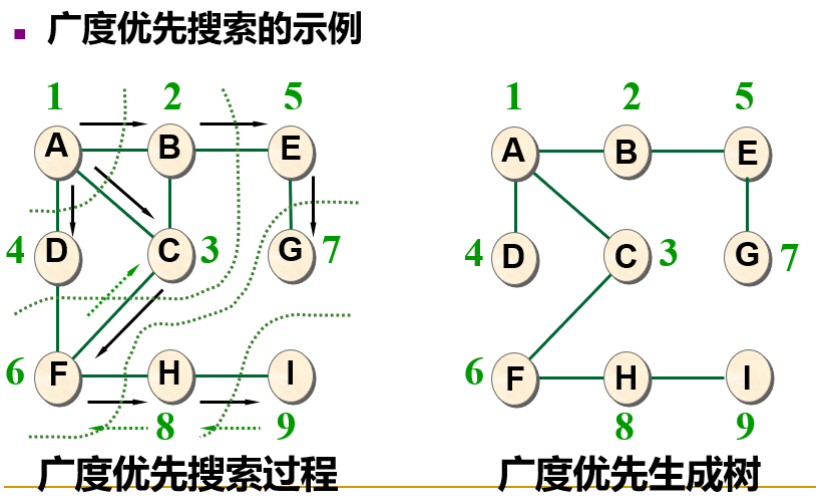
辅助数组visited[ ]的初始状态为 0, 在图的遍历过程中, 一旦某一个顶点 i 被访问, 就立即让visited[i]为1, 防止它被多次访问

8.3.1 深度优先搜索DFS (Depth First Search)




```
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
    }
};
```

8.3.2 广度优先搜索BFS (Breadth First Search)



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

```
template <class T, class E>
void BFS (Graph<T, E>& G, const T& v) {
    int i, w, n = G.NumberOfVertices();
    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;
};
```

8.3.3 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最大连通子图（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。

```
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
        }
```

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；**
  - **各边上的权值的总和达到最小。**

8.4.1 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, Æ }, 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中的所有的边, 堆中每个结点的格式为

n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。

```
#include "heap.h"
#include "UFSets.h"
template <class T, class E>
void Kruskal (Graph<T, E>& G, MinSpanTree<T, E>& MST) {
    MSTEdgeNode<T, E> ed;           //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) {
                ed.tail = u;  ed.head = v;      //插入堆
                ed.cost = G.getWeight (u, v);
                H.Insert(ed);
            }
    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(ed);           //退出具最小权值的边
        u = F.Find(ed.tail);
        v = F.Find(ed.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(ed);     //该边存入MST
            count++;
        }
    }
};
```

8.4.2 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 u0 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。

```
#include "heap.h"
template <class T, class E>
void Prim (Graph<T, E>& G, const T u0, MinSpanTree<T, E>& MST) {
    MSTEdgeNode<T, E> ed; //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
                ed.tail = u;  ed.head = v;
                ed.cost = G.getWeight(u, v);
                H.Insert(ed);           //(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(ed);                   //选堆中具最小权的边
            if (!Vmst[ed.head]) {
                MST.Insert(ed);             //加入最小生成树
                u = ed.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。

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）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**

8.5.1 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

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

③ 修改：
dist[i]←min{dist[i], dist[k]+Edge[k][i]},
对于每一个 $i \in V-S$；

④ 判断：若 $S = V$，则算法结束，否则转②。

```
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
            }
        }
    }
};
```

8.5.2 所有顶点之间的最短路径(Floyd算法)

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

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

- $A^{(0)}[i][j]$**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。**

```
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。

8.6 活动网络 (Activity Network)

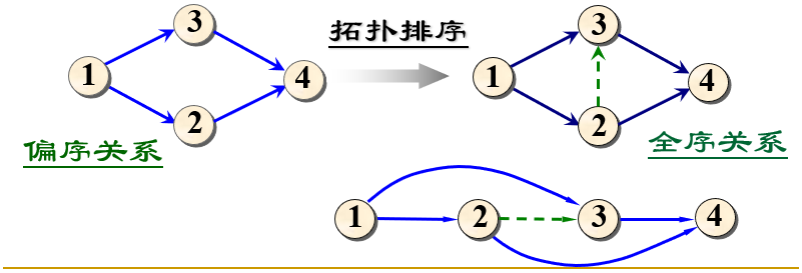
8.6.1 用顶点表示活动的网络(AOV网络)

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<Vi, Vj>表示活动Vi 必须先于活动Vj 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。

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。

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, 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。

```
template <class T, class E>
void TopologicalSort (Graph<T, E>& G) 、
{
    int i, j, w, v;
    int top = -1;                       //入度为零顶点的栈初始化
    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])                             //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
};
```

8.6.2 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。**

起始点为源点, 结束点为汇点。

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。

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。

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 $l[k] - e[k]$

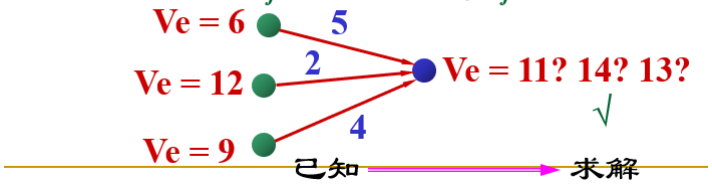
   表示活动a_k的最早可能开始时间和最迟允许开始时间的时间余量。$l[k] == e[k]$表示活动 a_k 是没有时间余量的关键活动。

- 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 $l[k] == e[k]$。

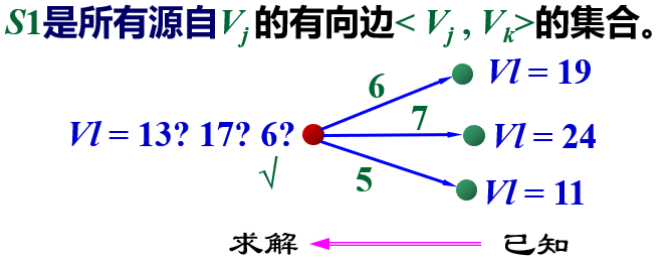
递推公式如下：

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

$S2$ 是所有指向V_j的有向边$<V_i, V_j>$的集合。



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



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

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

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

```
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[]
        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.getFirstNrighbor (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e, l
        j = G.getFirstNeighbor (i);
        while (j != -1) {
            Ae = Ve[i];  Al = Vl[k] - 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;
};
```

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。