第八章　图

1、复习要点

图是一种重要的非线性结构。其特点是每一个顶点都可以与其它顶点相关联。与树不同，图中各个顶点的地位都是平等的，对顶点的编号都是人为的。通常，定义图由两个集合构成：一个是顶点的非空有穷集合，一个是顶点与顶点之间关系（边）的有穷集合。对图的处理要区分有向图与无向图。其存储表示可以使用邻接矩阵，也可以使用邻接表，前者属顺序表示，后者属链接表示。本章着重讨论图的深度优先搜索算法和广度优先搜索算法，附带引入生成树与生成森林的概念。对于带权图，给出最小生成树的两种方法——Prim 算法与 Kruskal 算法，后者使用最小堆和并查集作为其辅助求解手段。在解决最短路径问题时，采用逐步求解的策略。最后，讨论作工程计划时常用的活动网络。涉及的主要概念是拓扑排序和关键路径，在解决应用问题时它们十分有用。

1.1 基本知识点

主要要求理解图的基本概念，包括图的定义、图的连通性、图的路径和路径长度、图中各顶点的度及度的度量、无向连通图的最大边数和最小边数、有向强连通图的最大边数与最小边数等。掌握图的存储表示，包括邻接矩阵和邻接表，以及这些存储表示上的典型操作，如构造、求根、找第一个邻接顶点、找下一个邻接顶点等操作的实现。要求掌握图的两种遍历算法——深度优先搜索算法和广度优先搜索算法，以及求解连通性问题的方法。此外，要求掌握构造最小生成树的 Prim 算法和 Kruskal 方法，掌握活动网络的拓扑排序算法，掌握求解关键路径的方法。需要注意的是，让某个关键活动提前完成，是否能让个工程提前完成。

1.2 算法设计

- ✧ 建立无向带权图的邻接表的算法，要求输入边的数目随机而定；
- ✧ 图的深度优先搜索的递归算法；
- ✧ 利用图的深度优先搜索的递归算法建立图的深度优先生成森林（用左孩子-右兄弟表示）的算法；
- ✧ 图的广度优先搜索算法；
- ✧ 利用图的广度优先搜索建立图的广度优先生成森林（用左孩子-右兄弟表示）的算法；
- ✧ 求解最小生成树的 Prim 算法，注意 nearvex 和 lowcost 辅助数组的变化；
- ✧ 求解最小生成树的 Kruskal 算法，注意 minheap 和 UFset 的变化；
- ✧ 求解最短路径的 Dijkstra 算法，注意 dist 辅助数组的变化；
- ✧ 有向图中求解拓扑排序的算法，要求用邻接表作为图的存储表示。注意算法执行过程中入度为零的顶点栈的变化；
- ✧ 有向图中求解拓扑排序的算法，要求用邻接矩阵作为图的存储表示。

2、难点与重点

- ✧ 图——图的定义与图的存储表示
  - (1) 邻接矩阵表示（通常是稀疏矩阵）；
  - (2) 邻接表与逆邻接表表示，要求建立算法；
  - (3) 邻接多重表（十字链表）表示。
- ✧ 深度优先遍历与广度优先遍历

(1) 生成数与生成树林的定义；
(2) 深度优先搜索算法和广度优先搜索算法；
(3) 深度优先搜索是个递归的过程，而广度优先搜索是个非递归的过程；
(4) 为防止重复访问已经访问过的顶点，需要设置一个访问标志数组 visited。

✧ 图的连通性

(1) 深度优先搜索可以遍历一个连通分量上的所有结点；
(2) 对非连通图进行遍历，可以建立一个生成森林；
(3) 对非强连通图进行遍历，可能建立一个生成森林；

✧ 最小生成树

(1) 对于连通网络，可用不会构成环路的权值最小的 n-1 条边构成最小生成树；
(2) 会画出用 Kruskal 算法及 Prim 算法构造最小生成树的过程。

✧ 单源最短路径

(1) 采用逐步求解的方式求某一顶点到其它顶点的最短路径的方法；
(2) 要求每条边的权值必须大于零。

✧ 活动网络

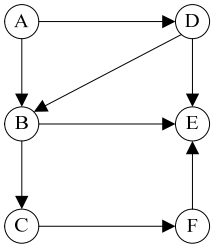
(1) 拓扑排序、关键路径、关键活动、AOE 网络；
(2) 拓扑排序将一个偏序图转化为一个全序图；
(3) 为实现拓扑排序，要建立一个栈，将所有入度为零的顶点进栈；
(4) 关键路径的计算。

3、重点概念分析

✧ **画出个顶点、2 个顶点、3 个顶点、4 个顶点和 5 个顶点的无向完全图。试证明在 n 个顶点的无向完全图中，边的条数为 n*(n-1)/2。**

在有 n 个顶点的无向完全图中，每一个顶点都有一条边与其它某一顶点相连，所以每一个顶点有 n-1 条边与其它 n-1 个顶点相连，总计 n 个顶点有 n(n-1)条边。但在无向图中，顶点 i 到顶点 j 与顶点 j 到顶点 i 是同一条边，所以总共有 n(n-1)/2 条边。

✧ **右边的有向图是强连通的吗？请列出所有的简单路径。**



判断一个有向图是否强连通，要看任一顶点出发是否能够回到该顶点。右面的有向图做不到这一点，它不是强连通的有向图，各个顶点自成强连通分量。

所谓简单路径是指该路径上没有重复的顶点。

(1) 从顶点 A 出发，到其它的各个顶点的简单路径有 A->B, A->D->B, A->B->C, A->D->B->C, A->D, A->B->E, A->D->E, A->D->B->E, A->B->C->F->E, A->D->B->C->F->E, A->B->C->F, A->D->B->C->F。
(2) 从顶点 B 出发，到其它各个顶点的简单路径有 B->C, B->C->F, B->E, B->C->F->E。
(3) 从顶点 C 出发，到其它各个顶点的简单路径有 C->F, C->F->E。
(4) 从顶点 D 出发，到其它各个顶点的简单路径有 D->B, D->B->C, D->B->C->F, D->E, D->B->E, D->B->C->F->E。
(5) 从顶点 E 出发，到其它各个顶点简单路径无。

(6) 从顶点 F 出发，到其它各个顶点的简单路径有 F->E。

✧ **用邻接矩阵表示图时，若图中有 1000 个顶点，1000 条边，则形成的邻接矩阵有多少矩阵元素？有多少非零元素？是否稀疏矩阵？**

一个图中有 1000 个顶点，其邻接矩阵中的矩阵元素有 1000^2=1000000 个。它有 1000 个非零元素（对于有向图）或 2000 个非零元素（对于无向图），因此是稀疏矩阵。

✧ **用邻接矩阵表示图时，矩阵元素的个数与顶点个数是否相关？与边的条数是否相关？**

用邻接矩阵表示图，矩阵元素的个数是顶点个数的平方，与边的条数无关。矩阵中非零元素的个数与边的条数有关。

✧ **有 n 个顶点的无向连通图至少有多少条边？有 n 个顶点的有向强连通图至少有多少条边？试举例说明。**

n 个顶点的无向连通图至少有 n-1 条边，n 个顶点的有向强连通图至少有 n 条边。特例情况是当 n=1 时，此时至少有 0 条边。

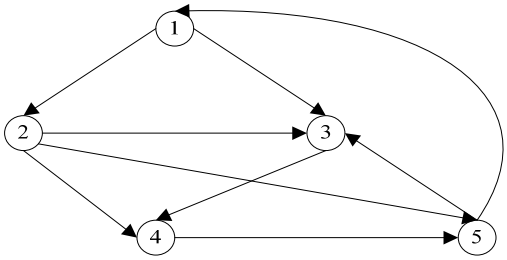
✧ **对于有 n 个顶点的无向图，采用邻接矩阵表示，如何判断以下问题：图中有多少条边？任意两个顶点 i 和 j 之间是否有边相连？任意一个顶点的度是多少？**

用邻接矩阵表示无向图时，因为是对称矩阵，对称矩阵的上三角部分或下三角部分检测一遍，统计其中的非零元素个数，就是图中的边数。如果邻接矩阵中 A[i][j]不为零，说明顶点 i 与顶点 j 之间有边相连。此外，统计出矩阵第 i 列的非零元素个数，就可得到顶点 i 的度数。

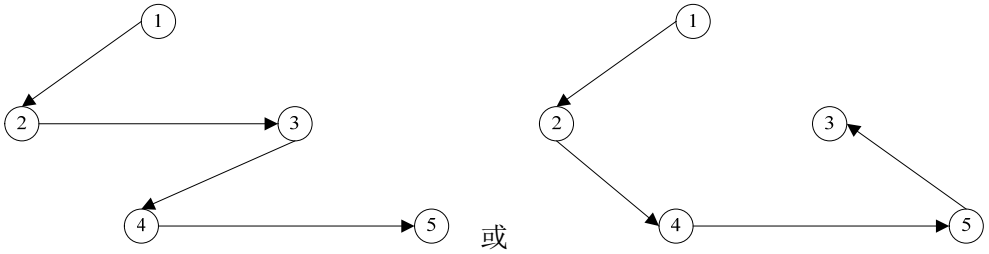
✧ **对于如右图所示的有向图，试写出：**

(1) 从顶点①出发进行深度优先搜索所得到的深度优先生成树；

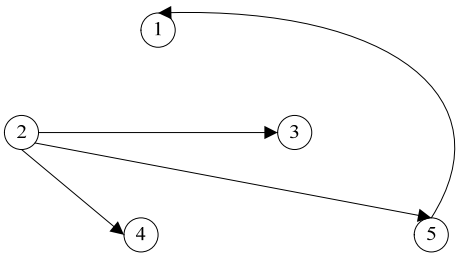
(2) 从顶点②出发进行广度优先搜索所得到的广度优先生成树。



(1) 以顶点①为根的深度优先生成树（不唯一）：



(2) 以顶点②为根的广度优先生成树：



✧ **试证明：对于一个无向图 G=(V, E)，若 G 中各顶点的度均大于或等于 2，则 G 中必有回路。**

反证法：对于一个无向图 G=(V, E)，若 G 中各顶点的度均大于或等于 2，则 G 中没有回路。此时，从某一个顶点出发，应能按拓扑有序的顺序遍历图中所有顶点。但当遍历到该顶点的另一邻接顶点时，又可能回到该顶点，没有回路的假设不成立。

4、重点习题解析

8-9 试扩充深度优先搜索算法，在遍历图的过程中建立生成森林的左孩子-右兄弟链表。算法的首部为 void Graph :: DFS(const int v, int visited[ ], TreeNode <int> *t)。其中，指针 t 指向生成森林上具有图顶点 v 信息的根结点。

【解答】

为建立生成森林，需要先给出建立生成树的算法，然后再在遍历图的过程中，通过一次次地调用这个算法，以建立生成森林。

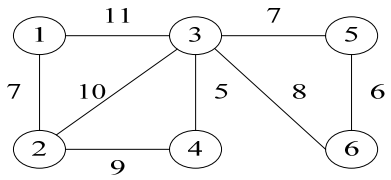
```
template <Type> void Graph <Type> ::
        DFS_Tree (const int v, int visited [ ], TreeNode <Type> *t)
{ //从图的顶点 v 出发，深度优先遍历图
  //建立以 t（已在上层算法中建立）为根的生成树
    visited[v]=1; int first=1; TreeNode <Type> *p, *q;
    int w=GetFirstNeighbor(v); //取第一个邻接顶点
    while (w!=-1)
    { //若邻接顶点存在
        if (visited[w]= =0)
        { //且该邻接顶点未访问过
            p=new TreeNode <Type> (GetValue(w)); //建立新的生成树结点
            if (first= =-1) //若根*t 还未链入任一子女
            { t->setFirstChild(p); first=0; } //新结点*p 成为根*t 的第一个子女
            else q->setNextSibling(p); //否则新结点*p 成为*q 的下一个兄弟
            q=p; //指针 q 总指示兄弟链最后一个结点
            DFS-Tree(w, visited, q); //从*q 向下建立子树
        }
        w=GetNextNeighbor(v, w); //取顶点 v 排在邻接顶点 w 的下一个邻接顶点
    }
}
```

下一个算法用于建立以左子女-右兄弟链表为存储表示的生成森林。

```
template <Type> void Graph <Type> :: DFS_Forest(Tree <Type> &T)
{ //从图的顶点出发，深度优先遍历图，建立以左子女-有兄弟链表表示的生成森林 T
```

```
    T.root=NULL; int n=NumberOfVertices( ); //顶点个数
    TreeNode <Type> *p, *q;
    int *visited=new int [n]; //建立访问标记数组
    for (int v=0; v<n; v++)
    { //逐个顶点检测
        if (visited[v]= =0)
        { //若未访问过
            p=new TreeNode <Type> (GetValue(v)); //建立新结点*p
            if (T.root= =NULL) T.root=p; //原来是空的生成森林，新结点成为根
            else q->setnextSibling(p); //否则新结点*p 成为*q 的下一个兄弟
            q=p;
            DFS_Tree(v, visited, p); //建立以*p 为根的生成树
        }
    }
}
```

8-13 编写一个完整的程序，首先定义堆和并查集的结构类型和相关操作，再定义 Kruskal 求连通网络的最小生成树算法的实现。并以下图为例，写出求解过程中堆、并查集和最小生成树的变化。



【解答】

完整的程序如下：

```
#include <iostream.h>
template <class Type> class MinHeap
{
    public:
        enum { MaxHeapSize=50; }
        MinHeap(int MaxSize=MaxHeapSize);
        MinHeap(Type Array[ ], int n);
        void Insert(const Type &ele);
        void RemoveMin(Type &Min);
        void Output( );
    private:
        void FilterDown(int start, int end);
        void FilterUp(int end);
        Type *pHeap;
        int HMaxSize;
        int CurrentSize;
};
class UFSets
{
```

```
    public:
        enum { MaxUnionSize=50 };
        UFSets(int MaxSize=MaxUnionSize);
        ~UFSets( ) { delete [ ] m_pParent; }
        void Union(int Root1, int Root2);
        int Find(int x);
    private:
        int m_iSize;
        int *m_pParent;
};
class Graph
{
    public:
        enum { MaxVerticesNum=50 };
        Graph(int Vertices=0) {CurrentVertices=Vertices; InitGraph( ); }
        void InitGraph( );
        void Kruskal( );
        int GetVerticesNum( ) { return CurrentVertices; }
    private:
        int Edge[MaxVerticesNum][MaxVerticesNum];
        int CurrentVertices;
};
class GraphEdge
{
    public:
        int head, tail;
        int cost;
        int operator <= (GraphEdge &ed);
};
GraphEdge :: operator <= (GraphEdge &ed)
{
    return this->cost<=ed.cost;
}
UFSets :: UFSets(int MaxSize)
{
    m_iSize=MaxSize;
    m_pParent=new int [m_iSize];
    for (int i=0; i<m_iSize; i++) m_pParent[i]=-1;
}
void UFSets :: Union(int Root1, int Root2)
{
    m_pParent[Root2]=Root1;
}
int UFSets :: Find(int x)
```

```
{
    while (m_pParent[x]>=0) x=m_pParent[x];
    return x;
}
template <class Type> MinHeap <Type> :: MinHeap(int MaxSize)
{
    HMaxSize=MaxSize;
    pHeap=new Type[HMaxSize];
    CurrentSize=-1;
}
template <class Type> MinHeap <Type> :: MinHeap(Type Array[ ], int n)
{
    HMaxSize=(n<MaxHeapSize)?MaxHeapSize : n;
    pHeap=new Type[HMaxSize];
    for (int i=0; i<n; i++) pHeap[i]=Array[i];
    CurrentSize=n-1;
    int iPos=(CurrentSize-1)/2;
    while (iPos>=0)
    {
        FilterDown(iPos, CurrentSize);
        iPos--;
    }
}
template <class Type> void MinHeap <Type> :: FilterDown(int start, int end)
{
    int i=start, j=2*start+1;
    Type Temp=pHeap[i];
    while (j<=end)
    {
        if (j<end && pHeap[j+1]<=pHeap[j]) j++;
        if (Temp<=pHeap[j]) break;
        pHeap[i]=pHeap[j];
        i=j; j=2*j+1;
    }
    pHeap[i]=Temp;
}
template <class Type> void MinHeap <Type> :: FilterUp(int end)
{
    int i=end, j=(end-1).2;
    Type Temp=pHeap[i];
    while (i>0)
    {
        if (pHeap[j]<=Temp) break;
        pHeap[i]=pHeap[j];
```

```
            i=j; j=(j-1)/2;
        }
        pHeap[i]=Temp;
}
template <class Type> void MinHeap <Type> :: Insert(const Type &ele)
{
        CurrentSize++;
        if (CurrentSize= =HMaxSize) return;
        pHeap[CurrentSize]=ele;
        FilterUp(CurrentSize);
}
template <class Type> void MinHeap <Type> :: RemoveMin(Type &Min)
{
        if (CurrentSize<0) return;
        Min=pHeap[0];
        pHeap[0]=pHeap[CurrentSize--];
        FilterDown(0, CurrentSize);
}
template <class Type> void MinHeap <Type> :: Output( )
{
        for (int i=0; i<=CurrentSize, i++) cout<<pHeap[i]<<” ”;
        cout<<endl;
}
void Graph :: InitGraph( )
{
        Edge[0][0]=-1; Edge[0][1]=28; Edge[0][2]=-1; Edge[0][3]=-1; Edge[0][4]=-1;
        Edge[0][5]=10; Edge[0][6]=-1; Edge[1][1]=-1; Edge[1][2]=16; Edge[1][3]=-1;
        Edge[1][4]=-1; Edge[1][5]=-1; Edge[1][6]=14; Edge[2][2]=-1; Edge[2][3]=12;
        Edge[2][4]=-1; Edge[2][5]=-1; Edge[2][6]=-1; Edge[3][3]=-1; Edge[3][4]=22;
        Edge[3][5]=-1; Edge[3][6]=18; Edge[4][4]=-1; Edge[4][5]=25; Edge[4][6]=24;
        Edge[5][5]=-1; Edge[5][6]=-1; Edge[6][6]=-1;
        for (int i=1; i<6; i++)
            for (int j=0; j<i; j++) Edge[i][j]=Edge[j][i];
}
void Graph :: Kruskal( )
{
        GraphEdge e;
        int VerticesNum=GetVerticesNum( );
        int i, j, count;
        MinHeap <GraphEdge> heap(VerticesNum*VerticesNum);
        UFSets set(VerticesNum);
        for (i=0; i<VerticesNum; i++)
        {
            for (j=i+1; j<VerticesNum; j++)
```

```
            {
                if (Edge[i][j]>0)
                {
                    e.head=i; e.tail=j; e.cost=Edge[i][j];
                    heap.Insert(e);
                }
            }
        }
        count=1;
        while (count<VerticesNum)
        {
            Heap.RemoveMin(e);
            i=set.Find(e.head);
            j=set.Find(e.tail);
            if (i!=j)
            {
                i=set.Find(e.head);
                j=set.Find(e.tail);
                if (i!=j)
                {
                    set.Union(i, j);
                    count++;
                    cout<<"("<<e.head<<","<<e.tail<<","<<e.cost<<")"<<endl;
                }
            }
        }
    }
```

8-18 利用 Dijkstra 算法的思想，设计一个求最小生成树的算法。

【解答】

计算连通网络的最小生成树的 Dijkstra 算法可描述如下：将连通网络中所有的边以方便的次序逐步加入到初始为空的生成树的边集合 T 中，每次选择并加入一条边时，需要判断它是否会与先前加入 T 的边构成回路。如果构成了回路，则从这个回路中将权值最大的边退选。

下面以邻接矩阵作为连通网络的存储表示，并以并查集作为判断是否出现回路的工具，分析算法的执行过程。算法的思路如下：

(1) 并查集初始化：将所有顶点置为只有一个顶点的的连通分量；

(2) 检查所有边；

①若边的两个端点 i 与 j 不在同一连通分量上（i 与 j 在并查集中不同根），则连通之（合并）；

②若边的两个端点 i 与 j 在同一连通分量上（i 与 j 在并查集中同根），则

——在并查集中寻找离 i 与 j 最近的共同祖先结点；

——分别从 i 与 j 向上检测具有最大权值的边；

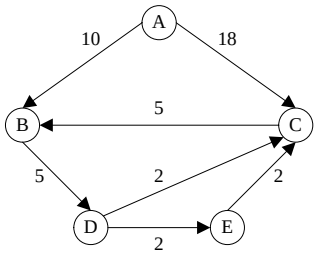
——在并查集上删除具有最大权值的边，加入新的边。

下面给出实现算法：

```
const int MaxNum=10000;
void Graph :: Dijkstra( )
{
    GraphEdge e;
    int VerticesNum=GetVerticesNum( );
    int i, j, p, q, k;
    int disjoint[VerticesNum]; //并查集
    for (i=0; i<VerticesNum; i++) disJoint[i]=-1; //并查集初始化
    for (i=0; i<VerticesNum-1; i++)
    { //检查所有的边
        for (j=i+1; j<VerticesNum; j++)
        {
            if (Edge[i][j]<MaxNum)
            { //边存在
                p=i; q=j; //判结点 i 与 j 是否在同一连通分量上
                while (disJoint[p]>=0) p=disJoint[p];
                while (disJoint[q]>=0) p=disJoint[q];
                if (p!=q) disJoint[j]=i; //i 与 j 不在同一连通分量上，连通之
            }
            else { //i 与 j 在同一连通分量上
                p=i; //寻找离结点 i 与 j 最近的祖先结点
                while (disJoint[p]>=0)
                { //每变动一个 p，就对 q 到根的路径检测一遍
                    q=j;
                    while (disJoint[q]>=0 && disJoint[q]= =disJoint[p])
                        p=disJoint[q];
                    if (disJoint[q]= =disJoint[p]) break;
                    else p=disJoint[p];
                }
                k=disJoint[p]; //结点 k 是 i 和 j 的最近的共同祖先
                p=i; q=disJoint[p]; max=-MaxNum; //从 i 到 k 找权值最大的边(s1, s2)
                while (q<=k)
                {
                    if (Edge[q][p]>max) { max=Edge[q][p]; s1=p; s2=q; }
                    p=q; q=disJoint[p];
                }
                p=j; q=disJoint[p]; max=-MaxNum; //从 j 到 k 找权值最大的边(t1, t2)
                while (q<=k)
                {
                    if (Edge[q][p]>max) { max=Edge[q][p]; t1=p; t2=q; }
                    p=q; q=disJoint[p];
                }
                max=Edge[i][j]; k1=i; k2=j;
                if (max<Edge[s1][s2]) {max=Edge[s1][s2]; k1=s1; k2=s2; }
```

```
                    if (max<Edge[t1][t2]) {max=Edge[t1][t2]; k1=t1; k2=t2; }
                    if (max!=Edge[i][j])
                    { //当 Edge[i][j]= =max 时边不变
                        if (disJoint[k1]= =k2) disJoint[k1]=-1;
                        else disJoint[k2]=-1; //删除权值最大的边
                        disJoint[j]=i; //加入新的边
                        Edge[j][i]=-Edge[j][i];
                    }
                }
            }
        }
    }
```

8-19 以右图为例，按 Dijkstra 算法计算得到的从顶点 A 到其它各个顶点的最短路径和最短路径长度。



【解答】

源点	终点	最短路径				最短路径长度			
A	B	(A, B)	(A, B)	(A, B)	(A, B)	10	10	10	10
	C	(A, C)	(A, C)	(A, C)	(A, C)	18	18	18	18
	D	—	(A, B, D)	(A, B, D)	(A, B, D)	∞	15	15	15
	E	—	—	(A, B, D, E)	(A, B, D, E)	∞	∞	17	17

8-20 在以下假设中，重写 Dijkstra 算法：

(1) 用邻接表表示带权有向图 G，其中每个边结点有 3 个域：邻接顶点 vertex，边上的权值 length 和边链表的链接指针 link。

(2) 用集合 T=V(G)-S 代替 S（已找到最短路径的顶点集合），利用链表来表示集合 T。

试比较新算法与原来的算法，计算时间是快了还是慢了，给出定量的比较。

【解答】

(1) 用邻接表表示的带权有向图的类定义：

```
const int DefaultSize=10; //缺省顶点个数
class Graph; //图的前视类定义
struct Edge
{ //边的定义
    friend class Graph;
    int vertex; //边的另一顶点位置
    float length; //边上的权值
    Edge *link; //下一条边链指针
    Edge( ) { } //构造函数
```

```
    Edge(int num, float wh) : vertec(num), Length(wh), link(NULL) { } //构造函数
    int operator < (const Edge &E) const {return length!=E.length; } //判边上的权值小否
}
struct Vertex
{ //顶点的定义
    friend class Graph;
    char data; //顶点的名字
    Edge *adj; //边链表的头指针
}
class Graph
{ //图的类定义
    private:
        Vertex *NodeTable; //顶点表（各边链表的头结点）
        int NumVertices; //当前顶点个数
        int NumEdges;  当前边数
        int GetVertexPos(const Type vertex); //给出顶点 vertex 在图中的位置
    public:
        Graph(int sz); //构造函数
        ~Graph( ); //析构函数
        int NumberOfVertices( ) { return NumVertices; } //返回图的顶点数
        int NumberOfEdges( ) { return NumEdges; } //返回图的边数
        char GetValue(int i) { return i>=0 && i<NumVertices ? NodeTable[i].data : ' '; }
        //取位置为 i 的顶点中的值
        float GetWeight(int v1, int v2); //返回边(v1, v2)上的权值
        int GetFirstNeighbor(int v); //取顶点 v 的第一个邻接顶点
        int GetNextNeighbor(int v, int w); //取顶点 v 的邻接顶点 w 的下一个邻接顶点
}
```

(2) 用集合 T=V(G)-S 代替 S（已找到最短路径的顶点集合），利用链表来表示集合 T。集合 T 用有序链表表示，数据域为顶点序号，链表 T 中的顶点都是未找到最短路径的顶点。另外设置一个数组 S，其作用是记录已找到的顶点 0 到其它各顶点的最短路径 path 及最短路径长度 len。

算法的主要思路是：

①对数组 S 及链表 T 初始化，记录顶点 0 到各个顶点的初始最短路径及其长度；

②扫描链表 T，寻找链表 T 中各个顶点到顶点 0 的当前最短路径中长度最小者，记为 u；

③在邻接表中扫描第 u 个顶点的出边表，确定每一边的邻接顶点号 k。若顶点 k 的最短路径没有选中过，比较绕过顶点 u 到顶点 k 的路径长度和原来顶点 0 到顶点 k 的最短路径长度，取其小者作为从顶点 0 到顶点 k 的新的最短路径。

④重复执行②、③步，直到图中所有顶点的最短路径长度都已选顶为止。

算法的实现如下：

```
const float MaxNum=10000000;
typedef struct info
{ //辅助数组元素：各顶点最短路径信息
    int pre; //在最短路径上前一顶点的顶点序号
    float len; //当前最短路径长度
```

```
}
info S[NumVertices]; //辅助数组：最短路径数组
List <int> T; //未选定最短路径顶点链表
int i, k, u; ListNode <int> *p;
T.MakeEmpty( );
for (i=1; i<NumVertices; i++)
{
    S[i].pre=0; S[i].len=MaxNum; //辅助数组初始化
    T.Locate(i); T.Insert(i); //形成有序链表 T
}
p=NodeTable[0].adj;
while (p!=NULL)
{ S[p->vertex].len=p->length; p=p->link; }
while (1)
{
    T.First( ); //循环检测链表 T
    if (!T.NextNotNull( ) break; //链表仅剩一个顶点，跳出循环，算法结束
    float min=MaxNum; u=0;
    while (T.NotNull( ))
    { //链表不空，还有剩余顶点未确定最短路径
        i=T.GetData( ); //取剩余顶点号
        if (S[i].len<min) { min=S[i].len; u=i; } //比较，寻找最短路径长度结点 u
        T.next( );
    }
    p=NodeTable[u].adj;
    while (p!=NULL)
    { //比较绕过顶点 u 到其它顶点 k 的路径长度
        k=p->vertex; //顶点 k 在链表 T 中表示该顶点未最终选定最短路径
        if (T.Find(k)!=NULL && S[u].len+p->length<S[k].len)
        { s[k].len=S[u].len+p->length; S[k].pre=u; } //修改
        p=p->link;
    }
    T.Find(u); T.Remove( ); //在链表 T 中删除顶点 u
}
```

8-24 设有一个有向图存储在邻接表中。试设计一个算法，按深度优先搜索策略对其进行拓扑排序，并以右图为例验证你的算法的正确性。

【解答】

(1) 利用邻接表结构，增加两个辅助数组和一个工作变量：

①记录各顶点入度 int indegree[NumVertices]；

②记录各顶点访问顺序 int visited[NumVertices]，初始时让 visited[i]=0, i=1, 2, …, NumVertices；

③访问计数 int count，初始时为 0。

(2) 拓扑排序算法：

```
void Graph :: dfs(int visited[ ], int indegree[ ], int v, int &count)
```

```
{
    count++; visited[v]=count;
    cout<<NodeTable[v].data<<endl;
    Edge *p=NodeTable[v].adj;
    while (p!=NULL)
    {
        int w=p->vertex;
        indegree[w]--;
        if (visited[w]= =0 && indegree[w]= =0) dfs(visited, indegree, w, count);
        p=p->link;
    }
}
```

主程序：

```
int i, j; Edge *p; float w;
cin>>NumVertices;
int *visited=new int[NumVertices+1];
int *indegree=new int[NumVertices+1];
for (i=1; i<NumVertices; i++)
{
    NodeTable[i].adj=NULL; cin>>NodeTable[i].data; cout<<endl;
    visited[i]=0; indegree[i]=0;
}
int count=0;
cin>>i>>j>>w; cout<<endl;
while (i!=0 && j!=0)
{
    p=new Edge(j, w);
    if (p= =NULL) { cout<<"存储分配失败！"<<endl; exit(1); }
    indegree[j]++;
    p->link=NodeTable[i].adj; NodeTable[i].adj=p;
    NumEdges++;
    cin>>i>>j>>w; cout<<endl;
}
for (i=1; i<NumVertices; i++)
    if (visited[i]= =0 && indegree[i]= =0) dfs(visited, indegree, i, count);
if (count<NumVertices) cout<<"排序失败！"<<endl;
else cout<<"排序成功！"<<endl;
delete [ ] visited; delete [ ] indegree;
```

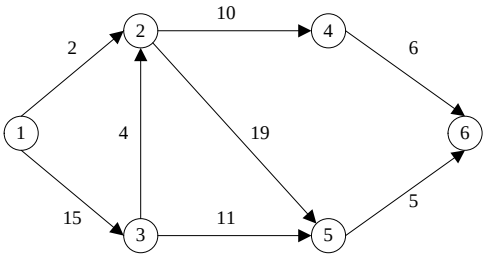
8-26 试对下图所示的 AOE 网络，解答下列问题。

(1) 这个工程最早可能在什么时间结束。

(2) 求每个事件的最早开始时间 Ve[i]和最迟允许开始时间 Vl[i]。

(3) 求每个活动的最早开始时间 e( )和最迟允许开始时间 l( )。

(4) 确定哪些活动是关键活动。画出由所有关键活动构成的图，指出哪些活动加速可使整个工程提前完成。



【解答】

按拓扑有序的顺序计算各个顶点的最早可能开始时间 Ve 和最迟允许开始时间 Vl。然后，再计算各个活动的最早可能开始时间 e 和最迟允许开始时间 l，根据 l-e 是否等于 0 来确定关键活动，从而确定关键路径。

	1 ①	2 ②	3 ③	4 ④	5 ⑤	6 ⑥
Ve	0	19	15	29	38	43
Vl	0	19	15	37	38	43

	<1, 2>	<1, 3>	<3, 2>	<2, 4>	<2, 5>	<3, 5>	<4, 6>	<5, 6>
e	0	0	15	19	19	15	29	38
l	17	0	15	27	19	27	37	38
l-e	17	0	0	8	0	12	8	0

此工程最早完成时间为 43。关键路径为<1, 3> <3, 2> <2, 5> <5, 6>。