关键路径 最短路和强连通分量 最小生成树

吴益强

C 版

日期：2022 年 5 月 19 日

1 拓扑排序和关键路径

1.1 AOV 网络

AOV 网（Activity On Vertex NetWork）用顶点表示活动，边表示活动（顶点）发生的先后关系。

若网中所有活动均可以排出先后顺序（任两个活动之间均确定先后顺序），则称网是拓扑有序的。

有向图或 AOV 网存在拓扑排序的充要条件：图中无环，即图是有向无环图（DAG）

算法步骤

```
在网络中选择一个入度为0的顶点输出；
在图中删除该顶点及所有以该顶点为起点的边；
重复上述过程，直至所有边均被输出。
```

1.2 AOV 代码实现

C 版

```
#include <bits/stdc++.h>
using namespace std;

const int  M  = 10001;
int matrix[M][M];
int indegree[M]; //book已排序的顶点个数

int main(int argc, char const *argv[])
{
    int i, j, a, b, k, book = 0, n, m;
    cin >> n >> m;
    for (i = 1; i <= m; ++i)
    {
        cin >> a >> b;
        matrix[a][b]=1;
        indegree[b]++;
```

```
    }
    for (i = 1; i <= n; ++i)
    {
        for (j = 1; j <= n; ++j)
        {
            if (indegree[j] == 0)
            {
                cout << j << " ";
                //遍历所有入度为0的顶点
                indegree[j] = -1;
                book++;
                for (k=1; k <= n; k++)
                {
                   if (matrix[j][k] == 1)
                    {
                        //遍历所有入度为1的顶点
                        matrix[j][k] = 0;
                        indegree[k]--;
                    }
                }
                break;
            }
        }
    }
    system("pause");
    return 0;
}
```

1.3 AOE 网络

AOE 网的定义：在带权有向图中若以顶点表示事件，有向边表示活动，边上的权值表示该活动持续的时间，这样的图简称为 AOE 网。

关键路径：是从开始点到完成点的最长路径的长度。路径的长度是边上活动耗费的时间。

如果将 AOE 网看做整个项目，那么完成整个项目至少需要多少时间?

为了求出一个给定 AOE 网的关键路径，需要知道以下 4 个统计数据：

对于 AOE 网中的顶点有两个时间：最早发生时间（用 Ve(j) 表示）和最晚发生时间（用 Vl(j) 表示）；

对于边来说，也有两个时间：最早开始时间（用 e(i) 表示）和最晚开始时间（l(i) 表示）。

算法实现：

```
#include <stdio.h>
#include <stdlib.h>
#define  MAX_VERTEX_NUM 20//最大顶点个数
#define  VertexType int//顶点数据的类型
typedef enum{false,true} bool;
```

```
//建立全局变量，保存边的最早开始时间
VertexType ve[MAX_VERTEX_NUM];
//建立全局变量，保存边的最晚开始时间
VertexType vl[MAX_VERTEX_NUM];
typedef struct ArcNode{
    int adjvex;//邻接点在数组中的位置下标
    struct ArcNode * nextarc;//指向下一个邻接点的指针
    VertexType dut;
}ArcNode;

typedef struct VNode{
    VertexType data;//顶点的数据域
    ArcNode * firstarc;//指向邻接点的指针
}VNode,AdjList[MAX_VERTEX_NUM];//存储各链表头结点的数组

typedef struct {
    AdjList vertices;//图中顶点及各邻接点数组
    int vexnum,arcnum;//记录图中顶点数和边或弧数
}ALGraph;
//找到顶点对应在邻接表数组中的位置下标
int LocateVex(ALGraph G,VertexType u){
    for (int i=0; i<G.vexnum; i++) {
        if (G.vertices[i].data==u) {
            return i;
        }
    }
    return -1;
}
//创建AOE网，构建邻接表
void CreateAOE(ALGraph **G){
    *G=(ALGraph*)malloc(sizeof(ALGraph));

    scanf("%d,%d",&((*G)->vexnum),&((*G)->arcnum));
    for (int i=0; i<(*G)->vexnum; i++) {
        scanf("%d",&((*G)->vertices[i].data));
        (*G)->vertices[i].firstarc=NULL;
    }
    VertexType initial,end,dut;
    for (int i=0; i<(*G)->arcnum; i++) {
        scanf("%d,%d,%d",&initial,&end,&dut);

        ArcNode *p=(ArcNode*)malloc(sizeof(ArcNode));
        p->adjvex=LocateVex(*(*G), end);
        p->nextarc=NULL;
        p->dut=dut;

        int locate=LocateVex(*(*G), initial);
```

```
        p->nextarc=(*G)->vertices[locate].firstarc;
        (*G)->vertices[locate].firstarc=p;
    }
}
//结构体定义栈结构
typedef struct stack{
    VertexType data;
    struct stack * next;
}stack;

stack *T;

//初始化栈结构
void initStack(stack* *S){
    (*S)=(stack*)malloc(sizeof(stack));
    (*S)->next=NULL;
}
//判断栈是否为空
bool StackEmpty(stack S){
    if (S.next==NULL) {
        return true;
    }
    return false;
}
//进栈，以头插法将新结点插入到链表中
void push(stack *S,VertexType u){
    stack *p=(stack*)malloc(sizeof(stack));
    p->data=u;
    p->next=NULL;
    p->next=S->next;
    S->next=p;
}
//弹栈函数，删除链表首元结点的同时，释放该空间，并将该结点中的数据域通过地址传
    值给变量i;
void pop(stack *S,VertexType *i){
    stack *p=S->next;
    *i=p->data;
    S->next=S->next->next;
    free(p);
}
//统计各顶点的入度
void FindInDegree(ALGraph G,int indegree[]){
    //初始化数组，默认初始值全部为0
    for (int i=0; i<G.vexnum; i++) {
        indegree[i]=0;
    }
    //遍历邻接表，根据各链表中结点的数据域存储的各顶点位置下标，在indegree数组
```

```
        相应位置+1
    for (int i=0; i<G.vexnum; i++) {
        ArcNode *p=G.vertices[i].firstarc;
        while (p) {
            indegree[p->adjvex]++;
            p=p->nextarc;
        }
    }
}

bool TopologicalOrder(ALGraph G){
    int indegree[G.vexnum];//创建记录各顶点入度的数组
    FindInDegree(G,indegree);//统计各顶点的入度
    //建立栈结构，程序中使用的是链表
    stack *S;
    //初始化栈
    initStack(&S);
    for (int i=0; i<G.vexnum; i++) {
        ve[i]=0;
    }
    //查找度为0的顶点，作为起始点
    for (int i=0; i<G.vexnum; i++) {
        if (!indegree[i]) {
            push(S, i);
        }
    }
    int count=0;
    //栈为空为结束标志
    while (!StackEmpty(*S)) {
        int index;
        //弹栈，并记录栈中保存的顶点所在邻接表数组中的位置
        pop(S,&index);
        //压栈，为求各边的最晚开始时间做准备
        push(T, index);
        ++count;
        //依次查找跟该顶点相链接的顶点，如果初始入度为1，当删除前一个顶点后，该
            顶点入度为0
        for (ArcNode *p=G.vertices[index].firstarc; p ; p=p->nextarc) {

            VertexType k=p->adjvex;

            if (!(--indegree[k])) {
                //顶点入度为0，入栈
                push(S, k);
            }
            //如果边的源点的最长路径长度加上边的权值比汇点的最长路径长度还长，
                就覆盖ve数组中对应位置的值，最终结束时，ve数组中存储的就是各顶点
```

```
                的最长路径长度。
            if (ve[index]+p->dut>ve[k]) {
                ve[k]=ve[index]+p->dut;
            }
        }
    }
    //如果count值小于顶点数量，表明有向图有环
    if (count<G.vexnum) {
        printf("该图有回路");
        return false;
    }
    return true;
}
//求各顶点的最晚发生时间并计算出各边的最早和最晚开始时间
void CriticalPath(ALGraph G){
    if (!TopologicalOrder(G)) {
        return ;
    }
    for (int i=0 ; i<G.vexnum ; i++) {
        vl[i]=ve[G.vexnum-1];
    }
    int j,k;
    while (!StackEmpty(*T)) {
        pop(T, &j);
        for (ArcNode* p=G.vertices[j].firstarc ; p ; p=p->nextarc) {
            k=p->adjvex;
            //构建Vl数组，在初始化时，Vl数组中每个单元都是18，如果每个边的汇点-
                边的权值比源点值小，就保存更小的。
            if (vl[k]-p->dut<vl[j]) {
                vl[j] = vl[k]-p->dut;
            }
        }
    }
    for (j = 0; j < G.vexnum; j++) {
        for (ArcNode*p = G.vertices[j].firstarc; p ;p = p->nextarc) {
            k = p->adjvex;
            //求各边的最早开始时间e[i],等于ve数组中相应源点存储的值
            int ee = ve[j];
            //求各边的最晚开始时间l[i]，等于汇点在vl数组中存储的值减改边的权值
            int el = vl[k]-p->dut;
            //判断e[i]和l[i]是否相等，如果相等，该边就是关键活动，相应的用*标
                记；反之，边后边没标记
            char tag = (ee==el)?'*':' ';
            printf("%3d%3d%3d%3d%3d%2c\n",j,k,p->dut,ee,el,tag);
        }
    }
}
```

```
int main(){
    ALGraph *G;
    CreateAOE(&G);//创建AOE网
    initStack(&T);
    TopologicalOrder(*G);
    CriticalPath(*G);
    return 0;
}
```

1.4 AOV 网络与 AOE 网络的关系

从定义上来看，很容易看出两种网的不同，AOV 网的活动以顶点表示，而 AOE 网的活动以有向边来表示，AOV 网的有向边仅仅表示活动的先后次序。纵观这两种网图，其实它们总体网络结构是一样的，仅仅是活动所表示的方式不同，因此可以猜想从 AOV 网转换成 AOE 网应该是可行的。

通常 AOE 网都是和关键路径联系在一起的，在 AOE 网中我们可以通过关键路径法来计算影响整个工期的关键路径，达到缩短工期的目的。在传统的 AOV 网中是没有表示活动时间的权值的，因此传统的 AOV 网无法估算工期，但是如果我们在 AOV 网中的活动结点上都标上时间属性，那么 AOV 网就可以完全转换为 AOE 网。

2 最短路 Dijkstra 算法

2.1 算法思路

```
1、通过Dijkstra计算图G中的最短路径时，需要指定起点s(即从顶点s开始计算)。
2、此外，引进两个集合S和U。S的作用是记录已求出最短路径的顶点(以及相应的最短路径
    长度)，而U则是记录还未求出最短路径的顶点(以及该顶点到起点s的距离)。
3、初始时，S中只有起点s；U中是除s之外的顶点，并且U中顶点的路径是”起点s到该顶点
    的路径”。然后，从U中找出路径最短的顶点，并将其加入到S中；接着，更新U中的顶
    点和顶点对应的路径。 然后，再从U中找出路径最短的顶点，并将其加入到S中；接
    着，更新U中的顶点和顶点对应的路径。 … 重复该操作，直到遍历完所有顶点。
```

3 有向图的强连通分量

在有向图 G 中，如果任意两个不同的顶点相互可达，则称该有向图是强连通的。有向图 G 的极大强连通子图再加任何一个顶点就会不再强连通的子图）称为 G 的强连通分量。

```
step1：对原图G进行深度优先遍历，记录每个节点的离开时间。
step2：选择具有最晚离开时间的顶点，对反图G^T进行遍历，删除能够遍历到的顶点，这
    些顶点构成一个强连通分量。
step3：如果还有顶点没有删除，继续step2，否则算法结束。
```

```
#include<iostream>
#include<cstring>
using namespace std;
const int MAXN=110;
int n; // 节点个数
bool flag[MAXN];//访问标志数组
int belg[MAXN];//存储强连通分量,其中belg[i]表示顶点i属于第belg[i]个强连通分量
int numb[MAXN];//结束时间标记,其中numb[i]表示离开时间为i的顶点
int G[MAXN][MAXN],GT[MAXN][MAXN]; //邻接矩阵,逆邻接矩阵
AdjTableadj[MAXN],radj[MAXN];//邻接表,逆邻接表
//用于第一次深搜,求得numb[1..n]的值
voidVisitOne(int cur,int &sig) // 访问原图
{
    flag[cur]=true;
    for(int i=0; i<n; ++i)
        if(G[cur][i] && flag[i] == false)
            VisitOne(i,sig);
    numb[++sig]=cur; //其中numb[i]表示离开时间为i的顶点
}
//用于第二次深搜,求得belg[1..n]的值
voidVisitTwo(int cur,int count) //访问逆图
{
    flag[cur]=true;
    belg[cur]=count;
    for(int i=0; i<n; ++i)
        if(GT[cur][i] && flag[i] == false)
            VisitOne(i,count);
}
//Kosaraju算法,返回为强连通分量个数
int Kosaraju_StronglyConnectedComponent()
{
    int sig=0;// sig代表离开时间
    //第一次深搜
    memset(flag,0,sizeof(bool)*n); //访问标志设为false
    for(int i=0; i<n; i++)
        if( flag[i] == false)
            VisitOne(i,sig);
    //第二次深搜
    memset(flag,0,sizeof(bool)*n); //访问标志设为false
    int count=0; // count为强连通分量个数
    for(int i=n; i>0; --i)
        if(flag[numb[i]] == false) //频率最高的顶点若没访问
            VisitTwo(numb[i],++count);
    return count;
}
```

4 最小生成树

对于一个无向连通带权图，每棵树的权（即树中所有边的权值总和）也可能不同
具有权最小的生成树称为最小生成树。

4.1 prim 算法

```
1).输入：一个加权连通图，其中顶点集合为V，边集合为E;
2).初始化：Vnew = {x}，其中x为集合V中的任一节点（起始点），Enew = {},为空;
3).重复下列操作，直到Vnew = V:
a.在集合E中选取权值最小的边<u，v>，其中u为集合Vnew中的元素，而v不在Vnew集合当
    中，并且v V（如果存在有多条满足前述条件即具有相同权值的边，则可任意选取其中
    之一）;
b.将v加入集合Vnew中，将<u，v>边加入集合Enew中;
4).输出：使用集合Vnew和Enew来描述所得到的最小生成树。
```

注意：时间复杂度

这里记顶点数 V，边数 E

邻接矩阵:O(v^2) 邻接表:O($Elog2(V)$)

不加堆优化的 Prim 算法适用于密集图，加堆优化的适用于稀疏图

```
void MiniSpanTree_PRIM (MGraph G, VertexType u) {
    /*  用普利姆算法從第u個頂點出發構造網G 的最小生成樹T,輸出T的各條邊。
        記🄵從頂點集U到V-U的代價最小的邊的輔助數組定義:
        struct
        {
            VertexType adjvex;
            VRtype lowcost;
        }closedge[MAX_VERTEX_NUM];
    */

    k = LocateVex(G, u);
    for (j = 0 ; j < G.vexnum; j++) {            //輔助數組初始化
        if (j != k)
            closedge[j] = {u, G.arcs[k][j].adj}; //{adjvex, lowcost}
    }
    closedge[k].lowcost = 0;                    //初始, U={u}
    for (i = 1; i < G.vexnum ; i++) {            //選擇其余G.vexnum -1 個頂點
        k = minimum(closedge);                  //求出T的下個結點: 第k結點
        //  此时 closedge[k].lowcost = MIN{ closedge[Vi].lowcost|closedge[Vi].
            lowcost>0,Vi V-U}
        printf(closedge[k].adjvex, G.vexs[k]);     //輸出生成樹的邊
        closedge[k].lowcost = 0;                //第k條邊🄵入U集
        for (j = 0; j < G.vexnum; j++) {

            //新頂點🄵入U後重新選擇最小邊
```

```
            if (G.arcs[k][j].adj < closedge[j].lowcost && closedge[j].lowcost
                !=0)
                closedge[j] = {G.vex[k], G.arcs[k][j].adj};
        }
    }
}
```

4.2 Kruskal 算法

```
1).记Graph中有v个顶点，e个边
2).新建图Graphnew，Graphnew中拥有原图中相同的e个顶点，但没有边
3).将原图Graph中所有e个边按权值从小到大排序
4).循环：从权值最小的边开始遍历每条边 直至图Graph中所有的节点都在同一个连通分量
    中
                if 这条边连接的两个节点于图Graphnew中不在同一个连通分量中
                                        添加这条边到图Graphnew中
```

时间复杂度：O(eloge)

伪代码：

```
KRUSKAL-FUNCTION(G, w)
1    F := 空集合
2    for each 图 G 中的顶点 v
3        do 将 v 加入森林 F
4    所有的边(u, v)   E依权重 w 递增排序
5    for each 边(u, v)   E
6        do if u 和 v 不在同一棵子树
7            then F := F   {(u, v)}
8                 将 u 和 v 所在的子树合并
```

hiho - 1098

```
#include<iostream>
#include<vector>
#include <climits>
#include<algorithm>
using namespace std;
int arr[100001];//用于保存节点的pre节点序号
int size[100001];//用于保存每个树的大小
void init(int n){//并查集的初始化
    for(int i=1;i<=n;i++)arr[i]=i;
    for(int i=1;i<=n;i++)size[i]=1;
}
int find(int i){//找树的祖先节点
    while(arr[i]!=i)i=arr[i];
    return i;
}
```

```
bool connect(int i,int j){//连接两个节点
    int a=find(i),b=find(j);
    if(a==b)return false;
    if(size[a]>size[b]){
        size[a]+=size[b];
        arr[b]=a;
    }else {
        size[b]+=size[a];
        arr[a]=b;
    }
    return true;
}
int main(){
    int n,m;
    cin>>n>>m;
    init(n);
    vector<int>mark(n);
    vector<vector<int>>road(m,vector<int>(3));
    for(int i=0;i<m;i++){
        cin>>road[i][0]>>road[i][1]>>road[i][2];
    }
    int totalcost=0;
    auto comp=[](vector<int>a,vector<int>b)->bool{//将所有的边按权值的大小升序
        排序，以便构造最小生成树
        return a[2]<b[2];
    };
    sort(road.begin(),road.end(),comp);
    for(int i=0,counter=1;i<m&&counter<n;i++){//注意，最小生成树的边为n-1
        if(connect(road[i][0], road[i][1])){//如果连接成功，则统计费用
            totalcost+=road[i][2];
            counter++;
        }
    }
    cout<<totalcost<<endl;//输出结果
}
```