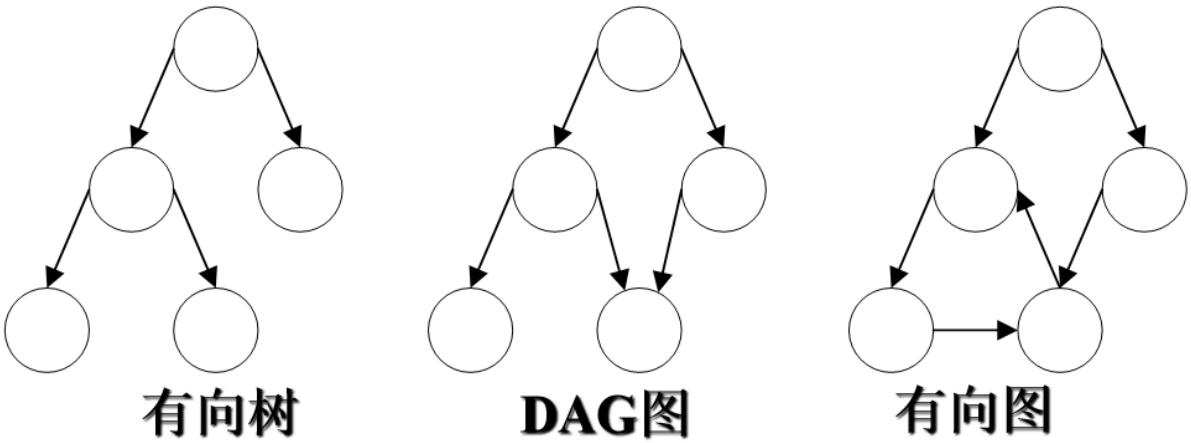
拓扑排序与关键路径

有向无环图

一个无环的有向图称作有向无环图，简称DAG图



2

DAG图是一类较有向树更一般的特殊有向图。常用来判断工程能否顺利进行，求出最短完成时间。

拓扑排序

听到**拓扑排序**你可能认为这是一种排序算法，然而实际上，**它是对有向图顶点排列成一个线性序列。**

在实际生活中，当我们要完成一系列事情时，各个活动间常常是有顺序关系或者依赖关系的，也就是在完成一件事之前必须要先做另一件事，比如说先穿袜子，再穿鞋子。这些事情都可以抽象成图论中的**拓扑排序。**

一、 相关概念

AOV网又称为"**顶点活动网络**", 用顶点表示活动，边表示活动（顶点）发生的先后关系。

我们把一条有向边起点的活动称作终点活动的前驱活动，同理终点活动成为起点活动的后继活动。

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。

拓扑排序 对于一个有向无环图(Directed Acyclic Graph简称DAG) G，将G中所有顶点排成一个线性序列，同时这个序列必须满足下面两个条件：

（1）每个顶点出现且只出现一次。

（2）若存在一条从顶点 A 到顶点 B 的路径，那么在序列中顶点 A 出现在顶点 B 的前面。

注：有向无环图（DAG）才有拓扑排序，非DAG图没有拓扑排序一说。

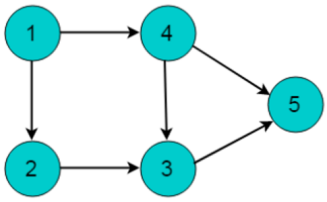
二、算法描述

基于BFS的拓扑排序

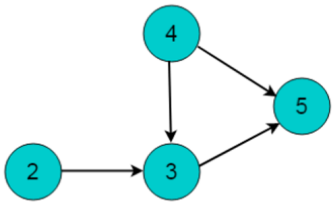
算法流程:

- 从图中选择一个入度为0的顶点，输出该顶点。
- 图中删去该顶点，以及所有以该顶点作为起点的边（即与之邻接的所有顶点入度-1）
- 重复上面两步，直到所有顶点都输出,即整个拓扑排序完成; 或没有不带前驱的顶点,这就说明此图中有回路，不可能进行拓扑排序。

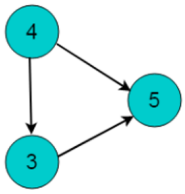
实例图解:



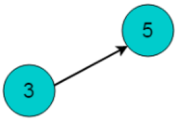
(1)



(2)



(3)



(4)

(5)

- 如上图(1)为一个有向无环图，选择图中入度为0的顶点1，输出顶点1。删除顶点1，并删除以顶点1为尾的边，删除后图为(2)。
- 继续选择入度为0的顶点。现在，图中入度为0的顶点有2和4，这里我们选择顶点2，输出顶点2。删除顶点2，并删除以顶点2为尾的边。删除后图为(3)。
- 选择入度为0的顶点4，输出顶点4.删除顶点4，并删除以顶点4为尾的边。删除后图为(4)
- 选择入度为0的顶点3，输出顶点3.删除顶点3，并删除以顶点3为尾的边。删除后图为(5)
- 最后剩余顶点5，输出顶点5，拓扑排序过程结束。最终的输出结果如下图:

性能分析

时间复杂度分析：统计所有节点入度的时间复杂性为$O（VE）$；接下来删边花费的时间也是$O(（VE）$，总花费时间为$O（VE）$。若使用队列保存入度为0的顶点，则可以将这个算法复杂度将为$O（V+E）$。

参考代码

```
#include<iostream>
#include<cstring>
using namespace std;
#define N 110
int G[N][N]; //储存有序对信息
int indegree[N];//储存入度数量
int topo[N];//储存拓扑序列
int main()
```

```
{
    int n;
    cin>>n;
    for(int i=1;i<=n;i++)
    {
        for(int j=1;j<=n;j++)
        {
            cin>>G[i][j];
            if(G[i][j]==1)//添加一条i指向j的一条边
                indegree[j]++;//j的入度+1
        }
    }

    int top=0;
        for(int i=1;i<=n;i++)
        {
            if(indegree[i]==0)
                topo[++top]=i;//入度为0就将对应点入栈
        }
        while(top>0)//当栈不为空时重复执行
        {
            int t=topo[top];//输出入度为0的元素，并删除
            cout<<t<<" ";
            top--;

            for(int i=;i<=n;i++)
            {
                if(G[t][i]) //遍历所有与它相邻的点
                {
                    G[t][i]=0;//删除边
                    indegree[i]--; //入度减1，如果变为了0，就入栈
                    if(indegree[i]==0)
                        topo[++top]=i;
                }
            }
        }
   return 0;
}
```

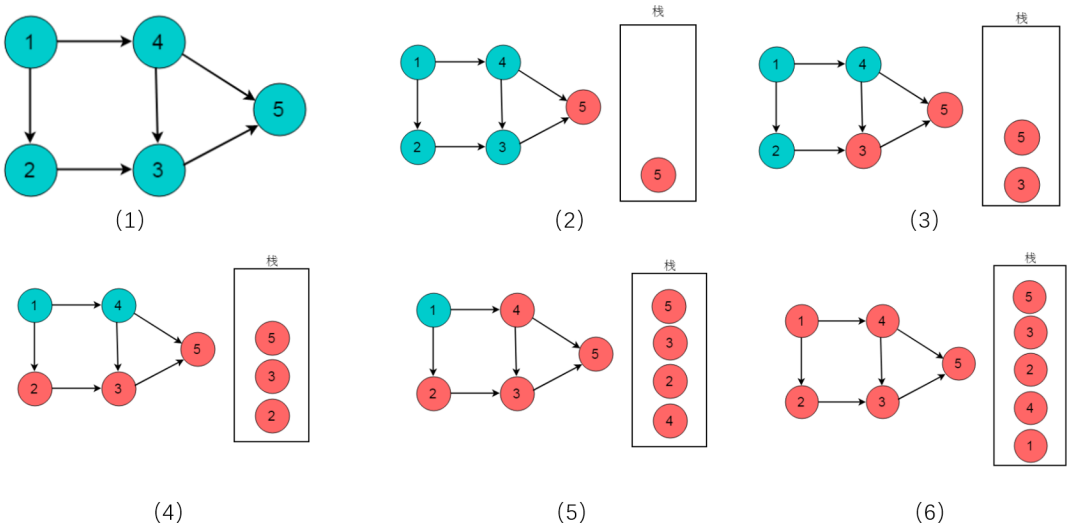
上述方法是用栈实现，也可以尝试使用队列来实现，同时如果将队列改为优先队列，还可以实现输出字典序最小的拓扑排序。

基于DFS的拓扑排序

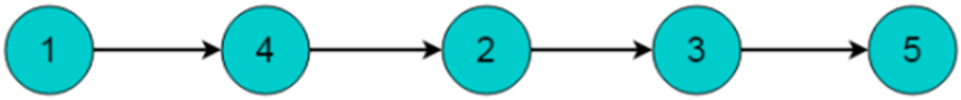
算法流程：

- 对图执行深度优先搜索。
- 在执行深度优先搜索时，若某个顶点不能继续前进，即顶点的出度为0，则将此顶点入栈。
- 最后得到栈中顺序的逆序即为拓扑排序顺序。

实例图解：



- 如上图(1)为一个有向无环图，选择起点为顶点1,，开始执行深度优先搜索。顺序为1->2->3->5。当深度优先搜索到达顶点5时，顶点5出度为0。将顶点5入栈。
- 深度优先搜索执行回退，回退至顶点3。此时顶点3的出度为0，将顶点3入栈。
- 回退至顶点2，顶点2出度为0，顶点2入栈。
- 回退至顶点1，顶点1可以前进位置为顶点4，顺序为1->4。顶点4出度为0，顶点4入栈。
- 回退至顶点1，顶点1出度为0，顶点1入栈。
- 栈的逆序输出。此顺序为拓扑排序结果：



性能分析

时间复杂度分析：首先深度优先搜索的时间复杂度为$O(V+E)$，而每次只需将完成访问的顶点存入数组中，需要$O(1)$，因而总复杂度为$O(V+E)$。

参考代码

```
#include<iostream>
#include<cstring>
using namespace std;
int n,m,topo[100];//储存最终形成的拓扑序列
int G[100][100];//储存有序对信息
int vis[100];//储存每个节点是否被访问过的信息
int top;
bool dfs(int u)
{
    vis[u]=-1;//该段代码的一个亮点，表示u节点正在被访问
    for(int v=1;v<=n;v++)
    {
        if(G[u][v])
        {
            if(vis[v]==-1)//访问到正在访问的节点，即为存在有向环
```

```
            return false;
            if(vis[v]==0)
            {
                if(!dfs(v))//深度优先遍历
                return false;
            }
        }
    }
    vis[u]=1;//返回时将该节点标记为已访问过
    topo[--top]=u;//将此节点插入拓扑序列
    return true;
}
int main()
{
     int a,b;
     cin>>n>>m;
     top=n;
     memset(G,0,sizeof(G));
     memset(vis,0,sizeof(vis));
     for(int i=0;i<m;i++)
     {
         cin>>a>>b;
         G[a][b]=1;
     }
     for(int u=1;u<=n;u++)
     {
         if(!vis[u])//如果该节点没有被访问过
         if(!dfs(u))//dfs函数对图中节点进行深度优先遍历，返回值表示拓扑排序是否存在
         {
            cout<<"存在有向环，失败退出"<<endl;
            return 0;
         }
     }
     for(int i=0;i<n;i++)
     cout<<topo[i]<<" "<<endl;
     return 0;
}
```

关键路径

一、 相关概念

AOE网：又称为"**边活动网络**"用顶点表示事件，用有向边表示活动，每个事件表示在它之前的活动已经完成，在它之后的活动可以开始。边上的权值表示活动的持续时间。

AOE网中没有入边的顶点称为始点（或源点），没有出边的顶点称为终点（或汇点）。

在AOE网中，从始点到终点具有最大路径长度（该路径上的各个活动所持续的时间之和）的路径称为**关键路径。**

对于求关键路径的算法我们可以求出工程的**最短完成时间。**

AOE网的性质：

(1) 只有在某顶点所代表的事件发生后，从该顶点出发的各活动才能开始；

(2) 只有在进入某顶点的各活动都结束，该顶点所代表的事件才能发生。

二、算法描述

关键路径，从图中源点到汇点的最长路径，也可以理解为影响整个项目周期的主分支。为了求这个路径，我们定义了以下四个参数：

ve(i)，表示事件/节点 i 的最早开始时间
vl(i)，表示事件/节点 i 的最迟开始时间
e(j)，表示活动/边 j 的最早开始时间
l(j)，表示活动/边 j 的最迟开始时间

ve(i)

节点i的最早开始时间取决于其前置节点最早的开始时间加上两个事件过渡的活动时间的最大值。

$$ve[i] = max(ve[i], ve[k] + path[k][i])$$

为什么取最大呢？因为这才能保证前置活动都完成，在每个节点都没有延迟的时候，**前置事件的最早开始时间和前置活动最早开始时间相同。**

当然，如果前置节点点有延迟时间的话还要加上这个延迟时间，不过一般不考虑，也就是这个延迟时间我们默认视它为 0 。

注：求该值的顺序要按**拓扑排序**的顺序进行，也就是从前往后的。

vl(i)

节点i 的最迟开始时间取决于其后置节点最迟的开始时间扣除两个事件过渡的活动时间的最小值。

$$vl[i] = min(vl[i], vl[k] - path[i][k])$$

在初学的时候这个地方会有两种认识偏差：

- 不应该是后置节点的最早开始时间扣除中间的权值嘛？也就是当前点的最迟开始时间不能影响后置节点的最早开始时间

  答：实际上这是一个在图上 DP 的问题，主要可能产生后效性、最优性的错误，不过这里暂时不从这方面讲。通俗的理解就是：

  第一，一个事件的最早开始时间一定是不受前置节点的最晚开始时间影响的，不然我们不就能够找到一个更早的开始时间了吗；

  第二，求解的时候我们应该一个边界对应一个边界，只要求点不影响关键路径，甚至可能出现当前节点的最迟开始时间比后置节点的最早开始时间要大，但前提是不影响整个工程的时间。

- 为什么取最小值？最迟不应该取最大值吗？

  答：一个事件可能有多个后置事件，如果当前事件的最迟开始时间是由后继事件的最迟开始时间的最大值传递过来的，相当于从后置事件的最右边界传递过来，那么，可能会导致一些更早的后置事件的最迟开始事件后移，这样就违背了我们求好的后置节点的最迟开始时间的结果了。

当然，如果当前点有延迟时间的话还要减去这个延迟时间，不过一般不考虑，也就是这个延迟时间我们默认视它为 0 。

注：求该值的顺序要按逆拓扑排序的顺序进行，也就是从后往前的。

e(j)

边 j 的最早开始时间和它的起点的事件最早开始时间相同。

$e(j) = ve(i)$，i 为j边的起点

如果起点有延迟，还要加上这个延迟时间，不过一般不考虑。

l(j)

边j的最迟开始时间为它的终点的事件最迟开始时间扣除边的边权，也就是本活动的时间，当然，如果出点如果有延迟还要减去这个延迟。

$l(j) = vl(i) - len(j)$，i为j边的终点，len为边权

时间余量

$d(j) = l(j) - e(j)$ 表示每个活动的时间余量，也就是活动能拖延的时间，当且仅当这个余量为0时，该活动为关键活动，也就是说关键活动没有可以拖延的余地。

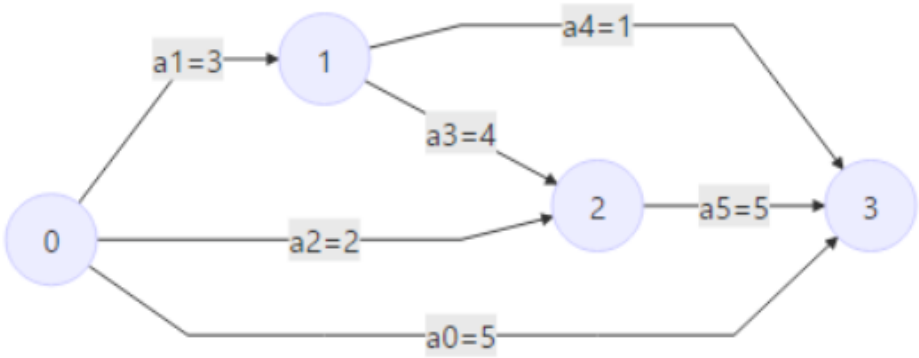
关键路径

当 $e(j) == l(j)$ 时，对应的边 $a(j)$就是关键活动，多个关键活动串接起来的边集合就是关键路径。

算法流程

- 求拓扑排序，计算$ve(i)$
- 求逆拓扑排序，计算$vl(i)$
- 计算$e(j)$、$l(j)$

实例讲解



求各个事件（结点）的最早时间（在这里我们用一个数组$ve[]$来存储各个事件的最早时间）和最晚时间（在这里我们用一个数组$vl[]$来存储各个事件的最迟时间），该工程图如上图所示。

首先进行拓扑排序（0,1,2,3）求最早时间，得到：

$ve[0] = 0;$

$ve[1] = 3;$

$ve[2] = max(ve[0] + a2, ve[1] + a3) = 7;$

$ve[3] = max(ve[0] + a0, ve[1] + a4, ve[2] + a5) = 12;$

然后逆拓扑排列（3,2,1,0）求最迟时间，得到

$vl[3] = 12;$

$vl[2] = 7;$

$vl[1] = min(vl[2] - a3, vl[3] - a4) = 3;$

$vl[0] = min(vl[3] - a0, vl[2] - a2, vl[1] - a1) = 0;$

接下来求该图“边”的最早时间用$e(ai)$表示，“边”的最迟时间用$l(ai)$表示；（注意，上面的是结点对应的的最早时间和最短时间，不要混淆）
最早事件时间，过程如下
$e(a1) = e(a2) = e(a0) = ve[0] = 0;$

$e(a3) = e(a4) = ve[1] = 3;$

$e(a5) = ve[2] = 7;$

最迟发生时间，过程如下
$l(a0) = vl[3] - a0 = 7;$

$l(a1) = vl[1] - a1 = 0;$

$l(a2) = vl[2] - a2 = 5;$

$l(a3) = vl[2] - a3 = 3;$

$l(a4) = vl[3] - a4 = 11;$

$l(a5) = vl[3] - a5 = 7;$

最后一步，建一个图表，将边的最早发生时间和边的最晚发生时间对比，如果一样，则这条边就是关键路径上的一条子路径。

空	最早发生时间	最迟发生时间	相同点
a0	0	7	×
a1	0	0	√
a2	0	5	×
a3	3	3	√
a4	3	11	×
a5	7	7	√

将相同的提取出来得到关键路径（a1,a3,a5）；

即a1->a3->a5就是这个图的关键路径。

参考代码：

```
#include<iostream>
#include<stdio.h>
#include<string.h>
#include<map>
#include<vector>
#include<sstream>
#include<list>
#include<stdlib.h>
#include<queue>
```

```cpp
using namespace std;

#define Maximum 1000

typedef struct EdgeListNode{ //边
    int adjId;
    int weight;
    EdgeListNode* next;
};

typedef struct VertexListNode{//顶点
    int in;//入度
    int data;
    EdgeListNode* firstadj; //指向边表
};

typedef struct GraphAdjList{ //图
    int vertexnumber;//顶点个数
    int edgenumber;
    VertexListNode vertextlist[Maximum];
};

//拓扑排序，返回拓扑排序结果，earlist存储每个顶点的最早开始时间
vector<int> ToplogicalSort(GraphAdjList g, int *earlist) {
    vector<int>v; //存储拓扑排序结果
    queue<int>q; //存储入度为0的顶点
    EdgeListNode *temp;
    int i, j, k, ans;

    ans = 0;
    v.clear();
    while(!q.empty()) {
        q.pop();
    }

    //找出所有入度为0的顶点
    for(i=1; i<=g.vertexnumber; i++) {
        if(g.vertextlist[i].in == 0) {
            q.push(i);
        }
    }

    while(!q.empty()) {
        i = q.front();
        v.push_back(i);
        q.pop();
        ans++;
        temp = g.vertextlist[i].firstadj;
        while(temp != NULL) {
            k = earlist[i] + temp->weight;
            if(k > earlist[temp->adjId]) {
                earlist[temp->adjId] = k;
            }
            j = --g.vertextlist[temp->adjId].in;
            if(j == 0) {
                q.push(temp->adjId);
            }
            temp = temp->next;
```

```
        }
    }

    if(ans < g.vertexnumber) {
        v.clear();
    }
    return v;
}

//求关键路径，返回存储关键路径顶点的vector
vector<int> CriticalPath(GraphAdjList g) {
    vector<int>ans;
    vector<int>path;
    int i, j, k, *earlist, *latest;
    EdgeListNode *temp;

    //earlist存储每个顶点的最早开始时间
    //latest存储每个顶点的最晚开始时间
    earlist = (int*)malloc(sizeof(int) * (g.vertexnumber+1));
    latest = (int*)malloc(sizeof(int) * (g.vertexnumber+1));
    path.clear();
    for(i=1; i<=g.vertexnumber; i++) {
        earlist[i] = 0;
    }

    ans = ToplogicalSort(g, earlist);
    for(i=1; i<=g.vertexnumber; i++) {
        latest[i] = earlist[ans[g.vertexnumber-1]];
    }
    for(i=g.vertexnumber; i>0; i--) {
        temp = g.vertextlist[i].firstadj;
        while(temp!=NULL) {
            if(latest[temp->adjId] - temp->weight < latest[i]) {
                latest[i] = latest[temp->adjId] - temp->weight;
            }
            temp = temp->next;
        }
    }

    //遍历每条边
    for(i=1; i<=g.vertexnumber; i++) {
        temp = g.vertextlist[i].firstadj;
        while(temp != NULL) {
            j = earlist[i];
            k = latest[temp->adjId] - temp->weight;
            if(j == k) { //是关键活动
                //把该活动的两个顶点加入path
                path.push_back(i);
                path.push_back(temp->adjId);
            }
            temp = temp->next;
        }
    }
    return path;
}
```