图论中的常用经典算法

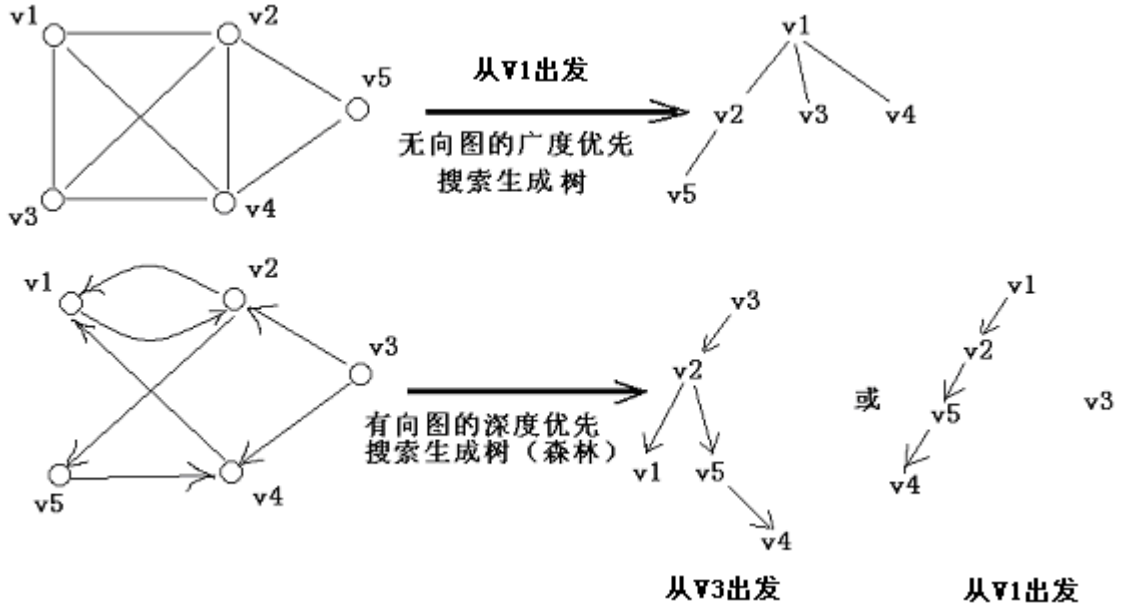
第一节　最小生成树算法

一、生成树的概念

若图是连通的无向图或强连通的有向图，则从其中任一个顶点出发调用一次 bfs 或 dfs 后便可以系统地访问图中所有顶点；若图是有根的有向图，则从根出发通过调用一次 dfs 或 bfs 亦可系统地访问所有顶点。在这种情况下，图中所有顶点加上遍历过程中经过的边所构成的子图称为原图的生成树。

对于不连通的无向图和不是强连通的有向图，若有根或者从根外的任意顶点出发，调用一次 bfs 或 dfs 后不能系统地访问所有顶点，而只能得到以出发点为根的连通分支（或强连通分支）的生成树。要访问其它顶点则还需要从没有访问过的顶点中找一个顶点作为起始点，再次调用 bfs 或 dfs，这样得到的是生成森林。

由此可以看出，一个图的生成树是不唯一的，不同的搜索方法可以得到不同的生成树，即使是同一种搜索方法，出发点不同亦可导致不同的生成树。如下图：



但不管如何，

我们都可以证明：具有 n 个顶点的带权连通图，其对应的生成树有 n-1 条边。

二、求图的最小生成树算法

严格来说，如果图 G=（V，E）是一个连通的无向图，则把它的全部顶点 V 和一部分边 E’构成一个子图 G’，即 G’=（V，　E’），且边集 E’能将图中所有顶点连通又不形成回路，则称子图 G’是图 G 的一棵生成树。

对于加权连通图，生成树的权即为生成树中所有边上的权值总和，权值最小的生成树称为图的最小生成树。

求图的最小生成树具有很高的实际应用价值，比如下面的这个例题。

例 1、城市公交网

[问题描述]

有一张城市地图，图中的顶点为城市，无向边代表两个城市间的连通关系，边上的权为在这两个城市之间修建高速公路的造价，研究后发现，这个地图有一个特点，即任一对城市都是连通的。现在的问题是，要修建若干高速公路把所有城市联系起来，问如何设计可使得工程的总造价最少。

[输入]

n（城市数，1<=n<=100）

e（边数）

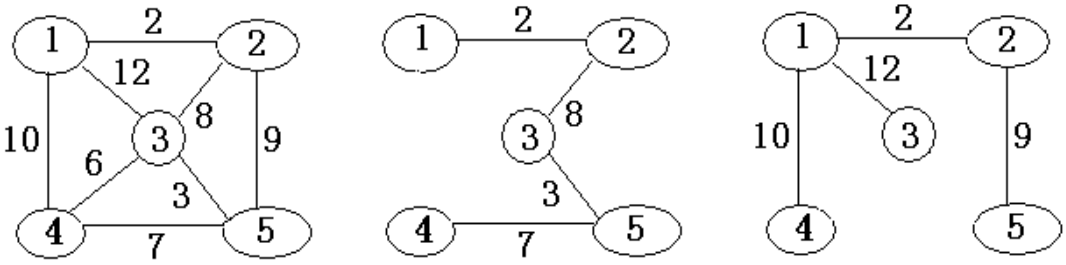
以下 e 行，每行 3 个数 i,j,w_{ij}，表示在城市 i,j 之间修建高速公路的造价。

[输出]

n-1 行，每行为两个城市的序号，表明这两个城市间建一条高速公路。

[举例]

下面的图（A）表示一个 5 个城市的地图，图（B）、（C）是对图（A）分别进行深度优先遍历和广度优先遍历得到的一棵生成树，其权和分别为 20 和 33，前者比后者好一些，但并不是最小生成树，最小生成树的权和为 19。



图（A）　　图（B）　　图（C）

[问题分析]

出发点：具有 n 个顶点的带权连通图，其对应的生成树有 n-1 条边。

那么选哪 n-1 条边呢？设图 G 的度为 n，G=（V，E），我们介绍两种基于贪心的算法，Prim 算法和 Kruskal 算法。

1、用 Prim 算法求最小生成树的思想如下：

①设置一个顶点的集合 S 和一个边的集合 TE，S 和 TE 的初始状态均为空集；

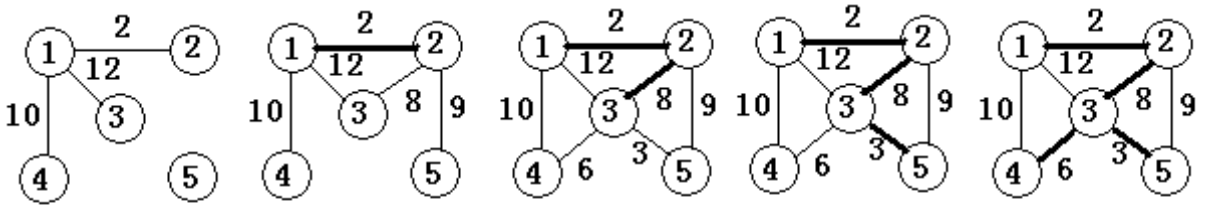
②选定图中的一个顶点 K，从 K 开始生成最小生成树，将 K 加入到集合 S；

③重复下列操作，直到选取了 n-1 条边：

选取一条权值最小的边（X，Y），其中 X∈S，not（Y∈S）；

将顶点 Y 加入集合 S，边（X，Y）加入集合 TE；

④得到最小生成树 T =（S，TE）



上图是按照 Prim 算法，给出了例题中的图（A）最小生成树的生成过程（从顶点 1 开始）。其中图（E）中的 4 条粗线将 5 个顶点连通成了一棵最小生成树。Prim 算法的正确性可以通过反证法证明。

因为操作是沿着边进行的，所以数据结构采用边集数组表示法，下面给出 Prim 算法构造图的最小生成树的具体算法框架。

① 从文件中读入图的邻接矩阵 g；

② 边集数组 elist 初始化；

```
For i:=1 To n-1 Do
  Begin
    elist[i].fromv:=1; elist[i].endv:=i+1; elist[i].weight:=g[1,i+1];
  End;
```

③ 求出最小生成树的 n-1 条边；

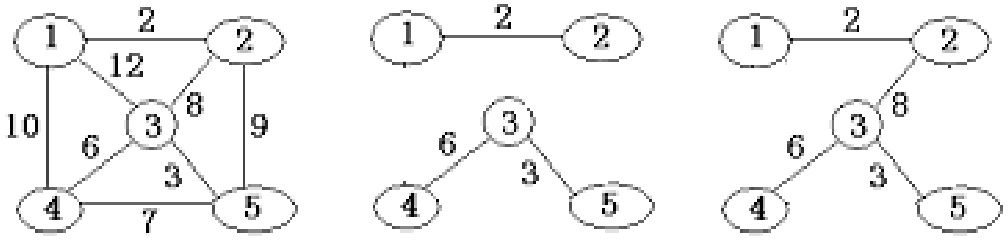
```
For k:=1 To n-1 Do
  Begin
    min:=maxint; m:=k;
    For j:=k To n-1 Do   {查找权值最小的一条边}
      If elist[j].weight<min Then Begin min:=elist[j].weight; m:=j; End;
    If m<>k Then Begin t:=elist[k]; elist[k]:=elist[m]; elist[m]:=t; End;
                                          {把权值最小的边调到第 k 个单元}
    j:=elist[k].endv;   {j 为新加入的顶点}
    For i:=k+1 To n-1 Do  {修改未加入的边集}
      Begin s:=elist[i].endv;  w:=g[j,s];
            If w<elist[i].weight
              Then Begin elist[i].weight:=w; elist[i].fromv:=j; End;
      End;
  End;
```

④ 输出；

2、用 Kruskal 算法求最小生成树的思想如下：

设最小生成树为 T=（V，TE），设置边的集合 TE 的初始状态为空集。将图 G 中的边按权值从小到大排好序，然后从小的开始依次选取，若选取的边使生成树 T 不形成回路，则把它并入 TE 中，保留作为 T 的一条边；若选取的边使生成树形成回路，则将其舍弃；如此进行下去，直到 TE 中包含 n-1 条边为止。最后的 T 即为最小生成树。如何证明呢？

下图是按照 Kruskal 算法给出了例题中图（A）最小生成树的生成过程：



Kruskal 算法在实现过程中的关键和难点在于：如何判断欲加入的一条边是否与生成树中已保留的边形成回路？我们可以将顶点划分到不同的集合中，每个集合中的顶点表示一个无回路的连通分量，很明显算法开始时，把所有 n 个顶点划分到 n 个集合中，每个集合只有一个顶点，表明顶点之间互不相通。当选取一条边时，若它的两个顶点分属于不同的集合，则表明此边连通了两个不同的连通分量，因每个连通分量无回路，所以连通后得到的连通分量仍不会产生回路，因此这条边应该保留，且把它们作为一个连通分量，即把它的两个顶点所在集合合并成一个集合。如果选取的一条边的两个顶点属于同一个集合，则此边应该舍弃，因为同一个集合中的顶点是连通无回路的，若再加入一条边则必然产生回路。

下面给出利用 Kruskal 算法构造图的最小生成树的具体算法框架。

① 将图的存储结构转换成边集数组表示的形式 elist，并按照权值从小到大排好序；

② 设数组 C[1..n-1]用来存储最小生成树的所有边，C[i]是第 i 次选取的可行边在排好序的 elist 中的下标；

③ 设一个数组 S[1..n]，S[i]都是集合，初始时 S[i]= [i]。

```
 i:=1; {获取的第 i 条最小生成树的边}
j:=1; {边集数组的下标}
While i<=n-1 Do
  Begin
    For k:=1 To n Do Begin      {取出第 j 条边，记下两个顶点分属的集合序号}
      If elist[j].fromv in s[k]  Then m1:=k;
      If elist[j].endv in s[k] Then m2:=k;
```

```
        End;
        If m1<>m2 Then Begin  {找到的 elist 第 j 条边满足条件，作为第 i 条边保留}
                        C[i]:=j;
                        i:=i+1;
                         s[m1]:=s[m1]+s[m2]; {合并两个集合}
                         s[m2]:=[ ];   {另一集合置空}
                       End;
        j:=j+1;   {取下条边，继续判断}
      End;
```

④ 输出最小生成树的各边：elist[C[i]]

3、总结

以上两个算法的时间复杂度均为 O（n*n）。**参考程序见 Prim.pas 和 Kruskal.pas。**

请大家用以上两种算法完成例 1。

三、应用举例

例 2、最优布线问题（wire.pas，wire.exe）

[问题描述]

学校有 n 台计算机，为了方便数据传输，现要将它们用数据线连接起来。两台计算机被连接是指它们时间有数据线连接。由于计算机所处的位置不同，因此不同的两台计算机的连接费用往往是不同的。

当然，如果将任意两台计算机都用数据线连接，费用将是相当庞大的。为了节省费用，我们采用数据的间接传输手段，即一台计算机可以间接的通过若干台计算机（作为中转）来实现与另一台计算机的连接。

现在由你负责连接这些计算机，你的任务是使任意两台计算机都连通（不管是直接的或间接的）。

[输入格式]

输入文件 wire.in，第一行为整数 n（2<=n<=100），表示计算机的数目。此后的 n 行，每行 n 个整数。第 x+1 行 y 列的整数表示直接连接第 x 台计算机和第 y 台计算机的费用。

[输出格式]

输出文件 wire.out，一个整数，表示最小的连接费用。

[样例输入]

```
3
0 1 2
1 0 1
2 1 0
```

[样例输出]

2（注：表示连接 1 和 2，2 和 3，费用为 2）

[问题分析]

本题是典型的求图的最小生成树问题，我们可以利用 Prim 算法或者 Kruskal 算法求出，下面的程序在数据结构上对 Kruskal 算法做了一点修改，具体细节请看程序及注解。

[参考程序]

```
Program wire(Input, Output);
var g : Array [1..100, 1..100] Of Integer; {邻接矩阵}
    l : Array [0..100] Of Integer;      {l[i]存放顶点i到当前已建成的生成树中
                                          任意一顶点j的权值g[i,j]的最小值}
```

```
    u : Array [0..100] Of Boolean;      { u[i]=True，表示顶点i还未加入到生成树中；
                                          u[i]=False，表示顶点I已加入到生成树中 }
    n, i, j, k, total : Integer;
Begin
  Assign(Input, 'wire.in');
  Reset(Input);
  Assign(Output, 'wire.out');
  Rewrite(Output);
  Readln(n);
  For i := 1 To n Do Begin
                       For j := 1 To n Do Read(g[i, j]);
                       Readln;
                     End;
  Fillchar(l, sizeof(l), $7F);  {初始化为maxint}
  l[1] := 0;                    {开始时生成树中只有第1个顶点}
  Fillchar(u, sizeof(u), 1);    {初始化为True，表示所有顶点均未加入}
  For i := 1 To n Do
    Begin
      k := 0;
      For j := 1 To n Do   {找一个未加入到生成树中的顶点，记为k，
                                要求k到当前生成树中所有顶点的代价最小}
        If u[j] And (l[j] < l[k])  Then k := j;
      u[k] := False;         {顶点k加入生成树}
      For j := 1 To n Do     {找到生成树中的顶点j，要求g[k,j]最小}
        If u[j] And (g[k, j] < l[j]) Then l[j] := g[k, j];
    End;
  total := 0;
  For i := 1 To n Do  Inc(total, l[i]);   {累加}
  Writeln(total);
  Close(Input);
  Close(Output);
End.
```

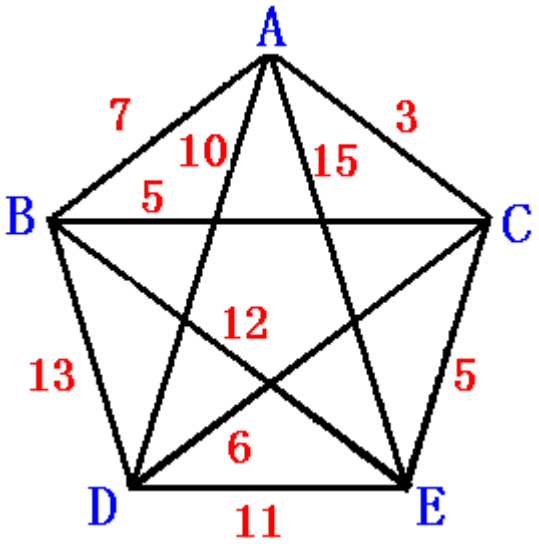
第二节　最短路径算法

最短路径是图论中的一个重要问题，具有很高的实用价值，也是信息学竞赛中常见的一类中等难度的题目，这类问题很能联系实际，考察学生的建模能力，反映出学生的创造性思维，因为有些看似跟最短路径毫无关系的问题也可以归结为最短路径问题来求解。本文就简要分析一下此类问题的模型、特点和常用算法。

在带权图 G=（V，E）中，若顶点 Vi,Vj 是图 G 的两个顶点，从顶点 Vi 到 Vj 的路径长度定义为路径上各条边的权值之和。从顶点 Vi 到 Vj 可能有多条路径，其中路径长度最小的一条路径称为顶点 Vi 到 Vj 的最短路径。一般有两类最短路径问题：一类是求从某个顶点（源点）到其它顶点（终点）的最短路径；另一类是求图中每一对顶点间的最短路径。

对于不带权的图，只要人为的把每条边加上权值 1，即可当作带权图一样处理了。

例 1、假设 A、B、C、D、E 各个城市之间旅费如下图所示。某人想从城市 A 出发游览各城市一遍，而所用旅费最少，试编程输出结果。



[问题分析]

解这类问题时，很多同学往往不得要领，采用穷举法把所有可能的情况全部列出，再找出其中旅费最少的那条路径；或者采用递归（深搜）找出所有路径，再找出旅费最少的那条。但这两种方法都是费时非常多的解法，如果城市数目多的话则很可能要超时了。

实际上我们知道，递归（深搜）之类的算法一般用于求所有解问题（例如求从 A 出发每个城市都要走一遍一共有哪几种走法？），所以这些算法对于求最短路径这类最优解问题显然是不合适的。

首先，对于这类图，我们都应该先建立一个邻接矩阵，存放任意两点间的数据（距离、费用、时间等），以便在程序中方便调用，上图的邻接矩阵如下：

```
const dis:array[1..5,1..5] of integer =( ( 0, 7, 3,10,15),
                                         ( 7, 0, 5,13,12),
                                         ( 3, 5, 0, 6, 5),
                                         (10,13, 6, 0,11),
                                         (15,12, 5,11, 0) );
```

以下介绍几种常见的、更好的算法。

一、 宽度优先搜索

宽搜也并不是解决这类问题的优秀算法，这里只是简单介绍一下算法思路，为后面的优秀算法做个铺垫。具体如下：

1、 从 A 点开始依次展开得到 AB、AC、AD、AE 四个新结点（第二层结点），当然每个新结点要记录下其旅费；

2、 再次由 AB 展开得到 ABC、ABD、ABE 三个新结点（第三层结点），而由 AC 结点可展开得到 ACB、ACD、ACE 三个新结点，自然由 AD 可以展开得到 ADB、ADC、ADE，由 AE 可以展开得到 AEB、AEC、AED 等新结点，对于每个结点也须记录下其旅费；

3、 再把第三层结点全部展开，得到所有的第四层结点：ABCD、ABCE、ABDC、ABDE、ABEC、ABED、……、AEDB、AEDC，每个结点也需记录下其旅费；

4、 再把第四层结点全部展开，得到所有的第五层结点：ABCDE、ABCED、……、AEDBC、AEDCB，每个结点也需记录下其旅费；

5、 到此，所有可能的结点均已展开，而第五层结点中旅费最少的那个就是题目的解了。

由上可见，这种算法也是把所有的可能路径都列出来，再从中找出旅费最少的那条，显而易见也是一种很费时的算法。

二、 A＊算法

A＊算法是在宽度优先搜索算法的基础上，每次并不是把所有可展开的结点展开，而是对所有没有展开的结点，利用一个自己确定的估价函数对所有没展开的结点进行估价，从而找出最应该被展开的结点（也就是说我们要找的答案最有可能是从该结点展开），而把该结点展开，直到找到目标结点为止。

这种算法最关键的问题就是如何确定估价函数，估价函数越准，则能越快找到答案。A＊算法实现起来并不难，只不过难在找准估价函数，大家可以自己找相关资料学习 A*算法。

三、等代价搜索法

等代价搜索法也是在宽度优先搜索的基础上进行了部分优化的一种算法，它与 A＊算法的相似之处都是每次只展开某一个结点（不是展开所有结点），不同之处在于：它不需要去另找专门的估价函数，而是以该结点到 A 点的距离作为估价值，也就是说，等代价搜索法是 A＊算法的一种简化版本。它的大体思路是：

1、 从 A 点开始依次展开得到 AB（7）、AC（3）、AD（10）、AE（15）四个新结点，把第一层结点 A 标记为已展开，并且每个新结点要记录下其旅费（括号中的数字）；

2、 把未展开过的 AB、AC、AD、AE 四个结点中距离最小的一个展开，即展开 AC（3）结点，得到 ACB（8）、ACD（16）、ACE（13）三个结点，并把结点 AC 标记为已展开；

3、 再从未展开的所有结点中找出距离最小的一个展开，即展开 AB（7）结点，得到 ABC（12）、ABD（20）、ABE（19）三个结点，并把结点 AB 标记为已展开；

4、 再次从未展开的所有结点中找出距离最小的一个展开，即展开 ACB（8）结点，……；

5、 每次展开所有未展开的结点中距离最小的那个结点，直到展开的新结点中出现目标情况（结点含有 5 个字母）时，即得到了结果。

由上可见，A＊算法和等代价搜索法并没有象宽度优先搜索一样展开所有结点，只是根据某一原则（或某一估价函数值）每次展开距离 A 点最近的那个结点（或是估价函数计算出的最可能的那个结点），反复下去即可最终得到答案。虽然中途有时也展开了一些并不是答案的结点，但这种展开并不是大规模的，不是全部展开，因而耗时要比宽度优先搜索小得多。

例 2、题目基本同例 1，现在把权定义成距离，现在要求 A 点到 E 点的最短路径，但并不要求每个城市都要走一遍。

[问题分析]

既然不要求每个点都要走一遍，只要距离最短即可，那么普通的宽度优先搜索已经没有什么意义了，实际上就是穷举。那么等代价搜索能不能再用在这题上呢？答案是肯定的，但到底搜索到什么时候才能得到答案呢？这可是个很荆手的问题。

是不是搜索到一个结点是以 E 结束时就停止呢？显然不对。

那么是不是要把所有以 E 为结束的结点全部搜索出来呢？这简直就是宽度优先搜索了，显然不对。

实际上，应该是搜索到：当我们确定将要展开的某个结点（即所有未展开的结点中距离最小的那个点）的最后一个字母是 E 时，这个结点就是我们所要求的答案！因为比这个结点大的点再展开得到的解显然不可能比这个结点优！

那么，除了等代价搜索外，有没有其它办法了呢？下面就介绍这种求最短路径问题的其它几种成熟算法。

四、宽度优先搜索+剪枝

搜索之所以低效，是因为在搜索过程中存在着大量的重复和不必要的搜索。因此，提高搜索效率的关键在于减少无意义的搜索。假如在搜索时已经搜出从起点 A 到点 B 的某一条路径的长度是 X，那么我们就可以知道，从 A 到 B 的最短路径长度必定≤X，因此，其他从 A 到 B 的长度大于或等于 X 的路径可以一律剔除。具体实现时，可以开一个数组 h[1..n]，n 是结点总数，h[i]表示从起点到结点 i 的最短路径长度。算法流程如下：

1、 初始化：

将起点 start 入队，h[start]:=0，h[k]:=maxlongint(1<=k<=n，且 k≠start)。

2、repeat

取出队头结点赋给 t；

while t 有相邻的结点没被扩展

begin

t 扩展出新的结点 newp；

如果 h[t]+w[t,newp] <h[newp]，

则将 newp 入队，把 h[newp]的值更新为 h[t]+w[t,newp]；

```
            end
      until 队列空;
```

以上算法实现的程序如下：

```
const  maxn=100;
       maxint=maxlongint div 4;
       maxq=10000;
var    h:array[1..maxn] of longint;
       g:array[1..maxn,1..maxn] of longint;
       n,i,j:longint;

procedure bfs;
var head,tail,i,t:longint;
    q:array[1..maxq] of longint;
begin
     for i:=1 to n do h[i]:=maxint;
     h[1]:=0;
     q[1]:=1;
     head:=0;tail:=1;
     repeat
           head:=head+1;
           t:=q[head];
           for i:=1 to n do
               if (g[t,i]<>maxint)and(h[t]+g[t,i]<h[i]) then
                  begin
                       tail:=tail+1;
                       q[tail]:=i;
                       h[i]:=h[t]+g[t,i];
                  end;
     until head=tail;
end;

begin
     assign(input,'data.in');
     reset(input);
     read(n);
     for i:=1 to n do
         for j:=1 to n do
         begin
              read(g[i,j]);
              if (g[i,j]<=0)and(i<>j) then g[i,j]:=maxint;
         end;

     bfs;

     for i:=2 to n do
         writeln('From 1 To ',i,' Weigh ',h[i]);
     close(input);
end.
```

五、迭代法

该算法的中心思想是：任意两点 i,j 间的最短距离（记为 Dij）会等于从 i 点出发到达 j 点的以任一点为中转点的所有可能的方案中，距离最短的一个。即：

Dij = min { Dij , Dik+Dkj }，1<=k<=n。

这样，我们就找到了一个类似动态规划的表达式，只不过这里我们不把它当作动态规划去处理，而是做一个二维数组用以存放任意两点间的最短距离，利用上述公式不断地对数组中的数据进行处理，直到各数据不再变化为止，这时即可得到 A 到 E 的最短路径。

算法流程如下：

D[i]表示从起点到 i 的最短路的长度，g 是邻接矩阵，s 表示起点；

1、D[i]:=g[s,i] (1<=i<=n);

2、repeat

```
      c:=false; {用以判断某一步是否有某个 Dij 值被修改过}
      for j:=1 to n do
        for k:=1 to n do
          if D[j]>D[k]+g[k,j] then
              begin D[j]:= D[k]+g[k,j];  c:=true; end;
   Until not c;
```

这种算法是产生这样一个过程：不断地求一个数字最短距离矩阵中的数据的值，而当所有数据都已经不能再变化时，就已经达到了目标的平衡状态，这时最短距离矩阵中的值就是对应的两点间的最短距离。

这个算法实现的程序如下：

```
const  maxn=100;
       maxint=maxlongint div 4;
var    D:array[1..maxn] of longint;
       g:array[1..maxn,1..maxn] of longint;
       n,i,j,k:longint;
       c:boolean;
begin
     assign(input,'data.in');
     reset(input);
     read(n);
     for i:=1 to n do
         for j:=1 to n do
         begin
              read(g[i,j]);
              if (g[i,j]<=0)and(i<>j) then g[i,j]:=maxint;
         end;

     for i:=1 to n do D[i]:=g[1,i];
     repeat
           c:=false;
           for j:=1 to n do
               for k:=1 to n do             {k 是中转点}
                   if D[j]>D[k]+g[k,j] then
                      begin
                           D[j]:=D[k]+g[k,j];
                           c:=true;
                      end;
```

```
    until not c;

    for i:=2 to n do
        writeln('From 1 To ',i,' Weigh ',D[i]);
    close(input);
end.
```

六、动态规划

动态规划算法已经成为了许多难题的首选算法。某些最短路径问题也可以用动态规划来解决，通常这类最短路径问题所对应的图必须是有向无回路图。因为如果存在回路，动态规划的无后效性就无法满足。

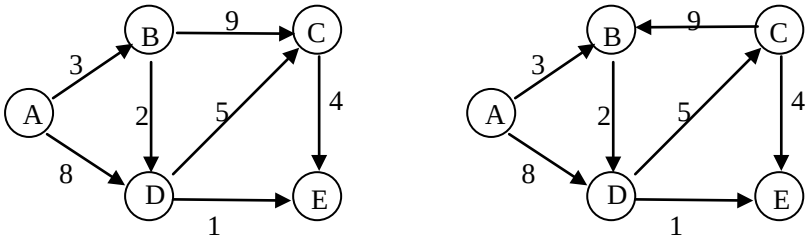
我们知道，动态规划算法与递归算法的不同之处在于它们的算法表达式：

递归：类似 f(n)=x1*f(n-1)+x2*f(n-2)……，即可以找到一个确定的关系的表达式；

动态规划：类似 f(n)=min(f(n-1)+x1,f(n-2)+x2……)，即我们无法找到确定关系的表达式，只能找到这样一个不确定关系的表达式，f(n)的值是动态的，随着 f(n-1),f(n-2)等值的改变而确定跟谁相关。

为了给问题划分阶段，必须对图进行一次拓扑排序（见下一节内容），然后按照拓扑排序的结果来动态规划。

譬如，有如下两个有向图：



右图因为存在回路而不能用动态规划。而左图是无回路的，所以可以用动态规划解决。

对左图拓扑排序，得到的序列是 A、B、D、C、E。

设 F(E)表示从 A 到 E 的最短路径长度，然后按照拓扑序列的先后顺序进行动态规划：

F(A)=0

F(B)=min{ F(A) }+3=3

F(D)=min{ F(A)+8, F(B)+2 }=5

F(C)=min{ F(B)+9, F(D)+5 }=10

F(E)=min{ F(D)+1, F(C)+4 }=6

总的式子是：F(i)=min{ F(k)+dis(i,k) }，k 与 i 必须相连，且在拓扑序列中，k 在 i 之前。

这个算法的参考程序如下：

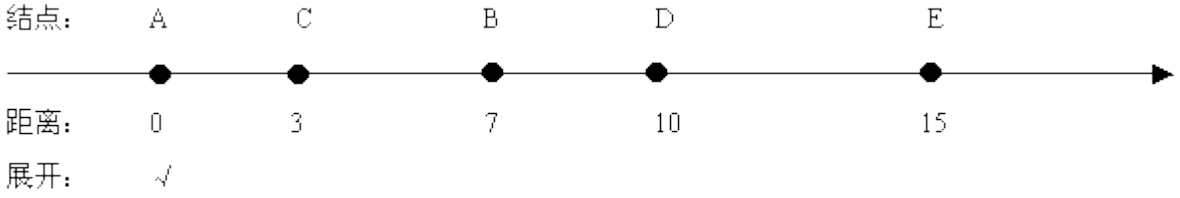
```
const  maxn=100;
       maxint=maxlongint div 4;
var    g:array[1..maxn,1..maxn] of longint;{有向图的邻接矩阵}
       pre:array[1..maxn] of longint; {pre[i]记录结点 i 的入度}
       tp:array[1..maxn] of longint;  {拓扑排序得到的序列}
       s:array[1..maxn] of longint;   {记录最短路径长度}
       n,i,j,k:longint;

begin
    assign(input,'data.in');
    reset(input);
    read(n);
    fillchar(pre,sizeof(pre),0);
    for i:=1 to n do
        for j:=1 to n do
        begin
```

```
            read(g[i,j]);
            if g[i,j]>0 then
              pre[j]:=pre[j]+1; {如果存在一条有向边 i→j，就把 j 的入度加 1}
        end;

    for i:=1 to n do  {拓扑排序}
    begin
         j:=1;
         while (pre[j]<>0) do j:=j+1;  {找入度为 0 的结点}
         pre[j]:=-1;
         tp[i]:=j;
         for k:=1 to n do
             if g[j,k]>0 then
                pre[k]:=pre[k]-1;
    end;

    filldword(s,sizeof(s)div 4,maxint); {s 数组中的单元初始化为 maxint}
    s[1]:=0;   {默认起点是 1 号结点}
    for i:=2 to n do     {动态规划}
        for j:=1 to i do
            if (g[tp[j],tp[i]]>0)and
               (s[tp[j]]+g[tp[j],tp[i]]<s[tp[i]])then
                s[tp[i]]:=s[tp[j]]+g[tp[j],tp[i]];

    for i:=2 to n do
        writeln('From 1 To ',i,' Weigh ',s[i]);
    close(input);
end.
```

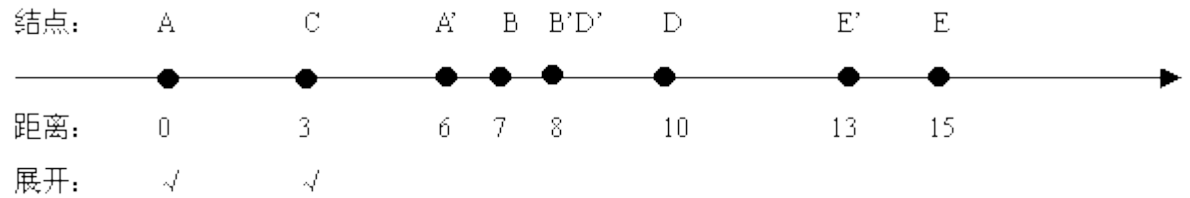
七、标号法

标号法是一种非常直观的求最短路径的算法，单从分析过程来看，我们可以用一个数轴简单地表示这种算法：

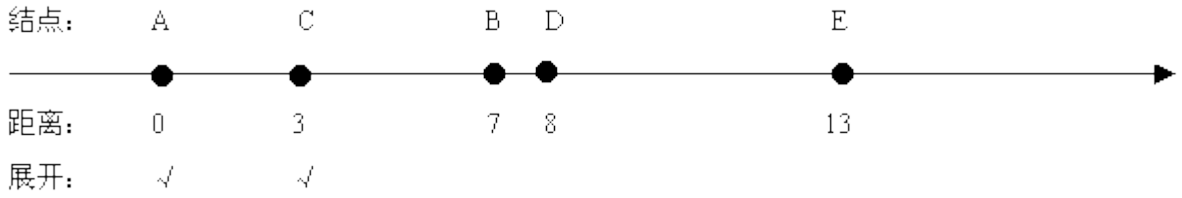
1、 以 A 点为 0 点，展开与其相邻的点，并在数轴中标出。



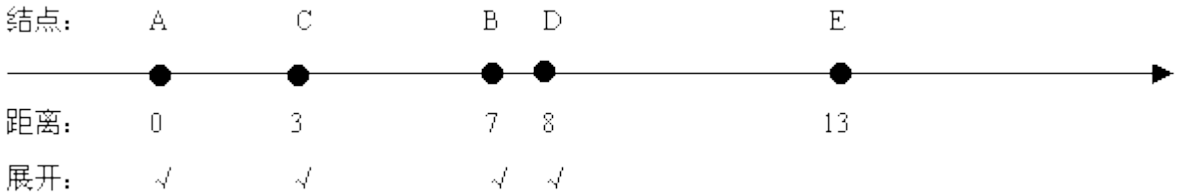
2、因为 C 点离起点 A 最近，因此可以断定 C 点一定是由 A 直接到 C 点这条路径是最短的（因为 A、C 两点间没有其它的点，所以 C 点可以确定是由 A 点直接到达为最短路径）。因而就可以以已经确定的 C 点为当前展开点，展开与 C 点想连的所有点 A’、B’、D’、E’。



3、由数轴可见，A 与 A’点相比，A 点离原点近，因而保留 A 点，删除 A’点，相应的，B、B’点保留 B 点，D、D’保留 D’，E、E’保留 E’，得到下图：



4、此时再以离原点最近的未展开的点 B 联接的所有点，处理后，再展开离原点最近未展开的 D 点，处理后得到如下图的最终结果：



5、由上图可以得出结论：点 C、B、D、E 就是点 A 到它们的最短路径（注意：这些路径并不是经过了所有点，而是只经过了其中的若干个点，而且到每一个点的那条路径不一定相同）。因而 A 到 E 的最短距离就是 13。至于它经过了哪几个点大家可在上述过程中加以记录即可。

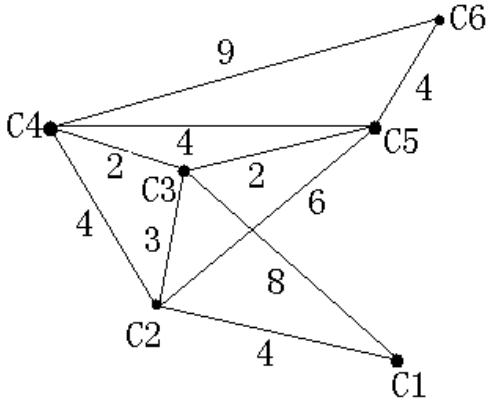
标号法的参考程序如下：

```
const  maxn=100;
       maxint=maxlongint div 4;
var    g:array[1..maxn,1..maxn] of longint; {邻接矩阵}
       mark:array[1..maxn] of boolean;      {用来标志某个点是否已被扩展}
       s:array[1..maxn] of longint;         {存储最短路径长度}
       n,i,j,k:longint;

begin
     assign(input,'data.in');
     reset(input);
     read(n);
     for i:=1 to n do
         for j:=1 to n do
         begin
              read(g[i,j]);
              if (i<>j)and(g[i,j]=0) then g[i,j]:=maxint;
         end;

     fillchar(mark,sizeof(mark),false);   {mark 初始化为 false}
     mark[1]:=true;  {将起点标志为已扩展}
     for i:=1 to n do s[i]:=g[1,i];  {s 数组初始化}
     repeat
          k:=0;
          for j:=1 to n do  {挑选离原点最近的点}
              if (not mark[j])and((k=0)or(s[k]>s[j])) then
                 k:=j;
          if k<>0 then
             begin
                  mark[k]:=true;
                  for j:=1 to n do   {扩展结点 k}
                      if (not mark[j])and(s[k]+g[k,j]<s[j]) then
                         s[j]:=s[k]+g[k,j];
```

```
            end;
    until k=0;

    for i:=2 to n do
        writeln('From 1 To ',i,' Weigh ',s[i]);
    close(input);
end.
```

八、Dijkstra 算法（从一个顶点到其余各顶点的最短路径，单源最短路径）

例 3、如下图，假设 $C_1, C_2, C_3, C_4, C_5, C_6$ 是六座城市，他们之间的连线表示两城市间有道路相通，连线旁的数字表示路程。请编写一程序，找出 C_1 到 C_i 的最短路径(2≤i≤6)，输出路径序列及最短路径的路程长度。



[问题分析]

对于一个含有 n 个顶点和 e 条边的图来说，从某一个顶点 Vi 到其余任一顶点 Vj 的最短路径，可能是它们之间的边（Vi，Vj），也可能是经过 k 个中间顶点和 k+1 条边所形成的路径(1≤k≤n-2)。下面给出解决这个问题的 Dijkstra 算法思想。

设图 G 用邻接矩阵的方式存储在 GA 中，GA[i,j]=maxint 表示 Vi，Vj 是不关联的，否则为权值（大于 0 的实数）。设集合 S 用来保存已求得最短路径的终点序号，初始时 S=[Vi]表示只有源点，以后每求出一个终点 Vj，就把它加入到集合中并作为新考虑的中间顶点。设数组 dist[1..n]用来存储当前求得的最短路径，初始时 Vi，Vj 如果是关联的，则 dist[j]等于权值，否则等于 maxint，以后随着新考虑的中间顶点越来越多，dist[j]可能越来越小。再设一个与 dist 对应的数组 path[1..n]用来存放当前最短路径的边，初始时为 Vi 到 Vj 的边，如果不存在边则为空。

执行时，先从 S 以外的顶点（即待求出最短路径的终点）所对应的 dist 数组元素中，找出其值最小的元素（假设为 dist[m]），该元素值就是从源点 Vi 到终点 Vm 的最短路径长度，对应的 path[m]中的顶点或边的序列即为最短路径。接着把 Vm 并入集合 S 中，然后以 Vm 作为新考虑的中间顶点，对 S 以外的每个顶点 Vj，比较 dist[m]+GA[m,j]的 dist[j]的大小，若前者小，表明加入了新的中间顶点后可以得到更好的方案，即可求得更短的路径，则用它代替 dist[j]，同时把 Vj 或边（Vm，Vj）并入到 path[j]中。重复以上过程 n-2 次，即可在 dist 数组中得到从源点到其余各终点的最段路径长度，对应的 path 数组中保存着相应的最段路径。

对于上图，采用 Dijkstra 算法找出 C_1 到 C_i 之间的最短路径(2≤i≤6)的过程如下：

初始时：

	1	2	3	4	5	6
Dist	0	4	8	maxint	maxint	maxint
Path	C_1	C_1, C_2	C_1, C_3			

第一次：选择 m=2，则 S=[C_1, C_2]，计算比较 dist[2]+GA[2,j]与 dist[j]的大小

	1	2	3	4	5	6
Dist	0	4	7	8	10	maxint
Path	C_1	C_1, C_2	C_1, C_2, C_3	C_1, C_2, C_4	C_1, C_2, C_5	

第二次：选择 m=3，则 S=[C_1, C_2, C_3]，计算比较 dist[3]+GA[3,j]与 dist[j]的大小

	1	2	3	4	5	6
Dist	0	4	7	8	9	maxint

Path	C_1	C_1,C_2	C_1,C_2,C_3	C_1,C_2,C_4	C_1,C_2,C_3,C_5	

第三次：选择 m=4，S=[C_1,C_2,C_3,C_4]，计算比较 dist[4]+GA[4,j]与 dist[j]的大小

	1	2	3	4	5	6
Dist	0	4	7	8	9	17
Path	C_1	C_1,C_2	C_1,C_2,C_3	C_1,C_2,C_4	C_1,C_2,C_3,C_5	C_1,C_2,C_4,C_6

第四次：选择 m=5，则 S=[C_1,C_2,C_3,C_4,C_5]，计算比较 dist[5]+GA[5,j]与 dist[j]的大小

	1	2	3	4	5	6
Dist	0	4	7	8	9	13
Path	C_1	C_1,C_2	C_1,C_2,C_3	C_1,C_2,C_4	C_1,C_2,C_3,C_5	C_1,C_2,C_3,C_5,C_6

因为该图的度 n=6，所以执行 n-2=4 次后结束，此时通过 dist 和 path 数组可以看出：

C_1到C_2的最短路径为：C_1——C_2，长度为：4；

C_1到C_3的最短路径为：C_1——C_2——C_3，长度为：7；

C_1到C_4的最短路径为：C_1——C_2——C_4，长度为：8；

C_1到C_5的最短路径为：C_1——C_2——C_3——C_5，长度为：9；

C_1到C_6的最短路径为：C_1——C_2——C_3——C_5——C_6，长度为：13；

下面给出具体的 Dijkstra 算法框架（注：为了实现上的方便，我们用一个一维数组 s[1..n]代替集合 S，用来保存已求得最短路径的终点集合，即如果 s[j]=0 表示顶点 Vj 不在集合中，反之，s[j]=1 表示顶点 Vj 已在集合中）。

```
Procedure Dijkstra(GA,dist,path,i)；    {表示求 Vi 到图 G 中其余顶点的最短路径，GA 为
                  图 G 的邻接矩阵，dist 和 path 为变量型参数，其中 path 的基类型为集合}
  Begin
    For j:=1 To n Do Begin      {初始化}
                       If j<>i Then s[j]:=0
                               Else s[j]:=1;
                       dist[j]:=GA[i,j];
                       If dist[j]<maxint        {maxint 为假设的一个足够大的数}
                         Then path[j]:=[i]+[j]
                         Else path[j]:=[ ];
                     End;
    For k:=1 To n-2 Do
      Begin
        w:=maxint; m:=i;
        For j:=1 To n Do   {求出第 k 个终点 Vm}
          If (s[j]=0) and (dist[j]<w) Then  Begin m:=j; w:=dist[j];  End;
        If m<>i Then s[m]:=1 else exit; {若条件成立，则把 Vm 加入到 S 中，否则退出循环，因为剩余的终
                  点，其最短路径长度均为 maxint，无需再计算下去}
        For j:=1 To n Do    {对 s[j]=0 的更优元素作必要修改}
          If (s[j]=0) and (dist[m]+GA[m,j]<dist[j])
            Then Begin
                   Dist[j]:=dist[m]+GA[m,j];
                   path[j]:=path[m]+[j];
                 End;
      End;
End;
```

九、Floyed 算法

例 4、求任意一对顶点之间的最短路径。

[问题分析]

这个问题的解法有两种：一是分别以图中的每个顶点为源点共调用 n 次 Dijkstra 算法，这种算法的时间复杂度为 O（n^3）；另外还有一种算法：Floyed 算法，它的思路简单，但时间复杂度仍然为 O（n^3），下面介绍 Floyed 算法。

设具有 n 个顶点的一个带权图 G 的邻接矩阵用 GA 表示，再设一个与 GA 同类型的表示每对顶点之间最短路径长度的二维数组 A，A 的初值等于 GA。Floyed 算法需要在 A 上进行 n 次运算，每次以 V_k（1≤k≤n）作为新考虑的中间点，求出每对顶点之间的当前最短路径长度，最后依次运算后，A 中的每个元素 A[i，j]就是图 G 中从顶点 V_i 到顶点 V_j 的最短路径长度。再设一个二维数组 P[1..n,1..n]，记录最短路径，其元素类型为集合类型 set of 1..n。

Floyed 算法的具体描述如下：

```
Procedure Floyed(GA，A，P);
  Begin
    For i:=1 To n Do        {最短路径长度数组和最短路径数组初始化}
      For j:=1 To n Do
        Begin
          A[i,j]:=GA[i,j];
          If A[i,j]<maxint  Then p[i,j]:=[i]+[j]
                            Else p[i,j]:=[ ];
        End;
    For k:=1 To n Do      {n 次运算}
      For i:=1 To n Do
        For j:=1 To n Do
          Begin
            If (i=k)or(j=k)or(i=j) Then Continue;
                 {无需计算，直接进入下一轮循环}
            If A[i,k]+A[k,j]<A[i,j] Then Begin  {找到更短路径、保存}
                                          A[i,j]: = A[i,k]+A[k,j];
                                          P[i,j]: = P[i,k]+P[k,j];
                                        End;
          End;
  End;
```

对于上图，大家可以运用 Floyed 算法，手工或编程试着找出任意一对顶点之间的最短路径及其长度。

十、总结与思考

最短路径问题的求解还不止这几种算法，比如还有分枝定界等等，而且大家也可以创造出各种各样的新算法来。不同的最短路径问题到底用哪种算法，以及还需要对该种算法作什么改动，是非常重要的，这种能力往往是很多同学所欠缺的，这需要大家在平常的训练中多做这类题目，还要多总结，以达到熟能生巧的境界。

在学习完最短路径后，有没有人想到：能不能修改这些算法，实现求最长路径的问题呢？这种发散性的思维是值得称赞的，对于不存在回路的有向图，这种算法是可行的。但需要提醒的是：如果有向图出现了回路，按照最长路径的思想和判断要求，则计算可能沿着回路无限制的循环下去。这就引出了一个问题：如何判断一个有向图中是否存在回路？可以用 bfs 或 dfs 在搜的过程检查这个点是否在前面出现过；当然也可以用下面的拓扑排序算法。

第三节 拓扑排序算法

在日常生活中，一项大的工程可以看作是由若干个子工程（这些子工程称为“活动”）组成的集合，这些子工程（活动）之间必定存在一些先后关系，即某些子工程（活动）必须在其它一些子工程（活动）完成之后才能开

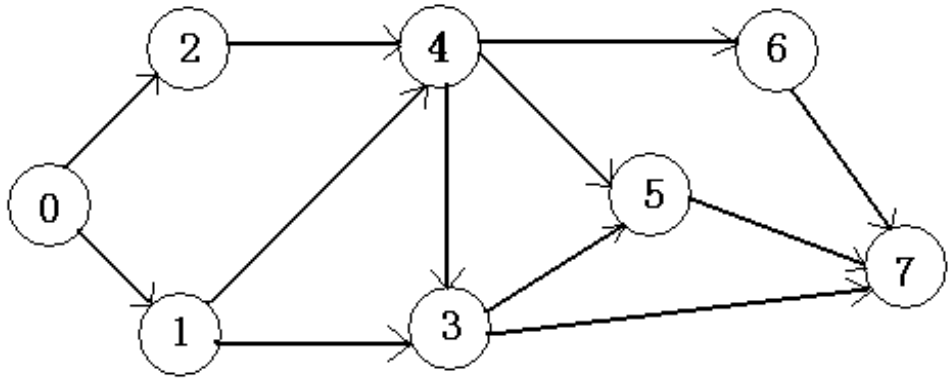
始，我们可以用有向图来形象地表示这些子工程（活动）之间的先后关系，子工程（活动）为顶点，子工程（活动）之间的先后关系为有向边，这种有向图称为“顶点活动网络”，又称“AOV 网”。

在 AOV 网中，有向边代表子工程（活动）的先后关系，即有向边的起点活动是终点活动的前驱活动，只有当起点活动完成之后终点活动才能进行。如果有一条从顶点 Vi 到 Vj 的路径，则说 Vi 是 Vj 的前驱，Vj 是 Vi 的后继。如果有弧<Vi，Vj>，则称 Vi 是 Vj 的直接前驱，Vj 是 Vi 的直接后继。

一个 AOV 网应该是一个有向无环图，即不应该带有回路，否则必定会有一些活动互相牵制，造成环中的活动都无法进行。

把不带回路的 AOV 网中的所有活动排成一个线性序列，使得每个活动的所有前驱活动都排在该活动的前面，这个过程称为“拓扑排序”，所得到的活动序列称为“拓扑序列”。

需要注意的是 AOV 网的拓扑序列是不唯一的，如对下图进行拓扑排序至少可以得到如下几种拓扑序列：02143567、01243657、02143657、01243567。



在上图所示的 AOV 网中，工程 1 和过程 2 显然可以同时进行，先后无所谓；但工程 4 却要等工程 1 和工程 2 都完成以后才可进行；工程 3 要等到工程 1 和工程 4 完成以后才可进行；工程 5 又要等到工程 3、工程 4 完成以后才可进行；工程 6 则要等到工程 4 完成后才能进行；工程 7 要等到工程 3、工程 5、过程 6 都完成后才能进行。可见由 AOV 网构造拓扑序列具有很高的实际应用价值。

其实，构造拓扑序列的拓扑排序算法思想很简单：**只要选择一个入度为 0 的顶点并输出，然后从 AOV 网中删除此顶点及以此顶点为起点的所有关联边；重复上述两步，直到不存在入度为 0 的顶点为止，若输出的顶点数小于 AOV 网中的顶点数，则输出“有回路信息”，否则输出的顶点序列就是一种拓扑序列。**

对上图进行拓扑排序的过程如下：

1、 选择顶点 0（唯一），输出 0， 删除边<0，1>，<0，2>；
2、 选择顶点 1（不唯一，可选顶点 2），输出 1， 删除边<1，3>，<1，4>；
3、 选择顶点 2（唯一），输出 2， 删除边<2，4>；
4、 选择顶点 4（唯一），输出 4， 删除边<4，3>，<4，5>，<4,6>；
5、 选择顶点 3（不唯一，可选顶点 6），输出 3， 删除边<3，5>，<3，7>；
6、 选择顶点 5（不唯一，可选顶点 6），输出 5， 删除边<5，7>；
7、 选择顶点 6（唯一），输出 6， 删除边<6，7>；
8、 选择顶点 7（唯一），输出 7， 结束。

输出的顶点数 m=8，与 AOV 网中的顶点数 n=8 相等，所以输出一种拓扑序列：01243567。

为了算法实现上的方便，我们采用邻接表存储 AOV 网，不过稍做修改，在顶点表中增加一个记录顶点入度的域 id，具体的拓扑排序算法描述如下：

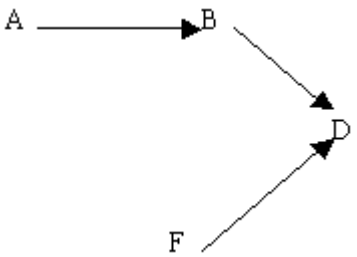
```
Procedure TopSort(dig:graphlist); {用邻接表 dig 存储图 G}
Var  n,top,i,j,k,m: Integer;
       P:graphlist;
Begin
   n:=dig.adjv;          {取顶点总个数}
   top:=0;               {堆栈、初始化}
```

```
  For i:=1 To n Do     {对入度为0的所有顶点进行访问，序号压栈}
    If dig.adj[i].id = 0  Then Begin
                                  dig.adj[i].id:=top;
                                  top:=i;
                                End;
  m:=0;      {记录输出的顶点个数}
  While top<>0  Do           {栈不空}
    Begin
      j:=top;                    {取一个入度为0的顶点序号}
      top:=dig.adj[top].id;    {出栈、删除当前处理的顶点、指向下个入度为0的顶点}
      Write(dig.adj[top].v);   {输出顶点序号}
      m:=m+1;
      p:=dig.adj[j].link;      {指向Vj邻接表的第一个邻接点}
      While p<>nil  Do         {删除所有与Vj相关边}
        Begin
          k:=p^.adjv;          {下一个邻接点}
          dig.adj[k].id:= dig.adj[k].id - 1;      {修正相应点的入度}
          If dig.adj[k].id = 0 Then Begin         {入度为0的顶点入栈}
                                      dig.adj[k].id:=top;
                                      top:=k;
                                    End;
          p:=p^.next;          {沿边表找下一个邻接点}
        End;
    End;
    If m<n Then Writeln( 'no solution!' );      {有回路}
End;
```

思考：拓扑排序一般用在哪些场合呢？
解答：如判回路或图的动态规划过程中分阶段。

例1、士兵排队（soldier）
问题描述：

有n个士兵（1≤n≤26），编号依次为A、B、C，…… 队列训练时，指挥官要把一些士兵从高到矮依次排成一行。但现在指挥官不能直接获得每个人的身高信息，只能获得“p1比p2高”这样的比较结果（p1，p2∈{'A'，…，'Z'}），记为p1>p2。例如A>B，B>D，F>D。士兵的身高关系如图所示：



对应的排队方案有三个：AFBD、FABD、ABFD。

输入：（soldier.in）

第一行：一个整数k

第二至第k+1行：每行两个大写字母（中间和末尾都没有空格），代表两个士兵，且第一个士兵高度大于第二个士兵。

输出：（soldier.out）

一个只包含大写字母的字符序列，表示排队方案（只要一种方案即可）。

输入样例：（soldier.in）

```
3
AB
BD
FD
```

输出样例：（soldier.out）

```
AFBD
```

问题分析：

士兵的身高关系对应一张有向图，图中的顶点对应一个士兵，有向边<v_i，v_j>表示士兵 i 高于士兵 j。我们按照从高到矮将士兵排出一个线形的顺序关系，即为对有向图的顶点进行拓扑排序。

参考程序：

```
program soldier;
var  g:array['A'..'Z','A'..'Z'] of 0..1; {图的邻接矩阵}
     d:array['A'..'Z'] of longint;       {记录各顶点的入度}
     s:array['A'..'Z'] of boolean;       {用来记录出现过的士兵名}
     ans:string;
     ch,i:char;

procedure readata;   {读入，构图}
var i,j,k:longint;
    a,b:char;
begin
     fillchar(g,sizeof(g),0);       {g、d、s 初始化}
     fillchar(d,sizeof(d),0);
     fillchar(s,sizeof(s),false);
     readln(k);                     {读入边的条数}
     for i:=1 to k do
     begin
          readln(a,b);
          g[a,b]:=1;                  {构造有向边 a→b}
          d[b]:=d[b]+1;               {将 b 的入度加 1}
          s[a]:=true;                 {将 a、b 标记为出现过}
          s[b]:=true;
     end;
end;

begin  {main}
     assign(input,'soldier.in');
     reset(input);
     assign(output,'soldier.out');
     rewrite(output);
     readata;
     ans:='';    {拓扑排序}
```

```
    repeat
        ch:='A';
        while (ch<='Z') and not( (s[ch])and(d[ch]=0) ) do ch:=chr(ord(ch)+1);
        if ch<='Z' then
          begin
            s[ch]:=false;
            ans:=ans+ch;
            for i:='A' to 'Z' do
              if (s[i])and(g[ch,i]=1) then d[i]:=d[i]-1;
          end;
    until ch>'Z';
    writeln(ans);
    close(input);close(output);
end.
```

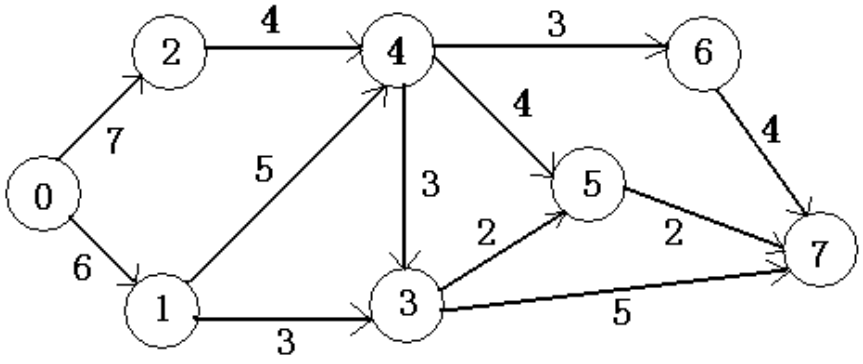
第四节　关键路径算法

利用 AOV 网络，对其进行拓扑排序能对工程中活动的先后顺序作出安排。但一个活动的完成总需要一定的时间，为了能估算出某个活动的开始时间，找出那些影响工程完成时间的最主要的活动，我们可以利用带权的有向网，图中的边表示活动，边上的权表示完成该活动所需要的时间，一条边的两个顶点分别表示活动的开始事件和结束事件，这种用边表示活动的网络，称为“AOE 网”。

其中，有两个特殊的顶点（事件），分别称为源点和汇点，源点表示整个工程的开始，通常令第一个事件（事件 1）作为源点，它只有出边没有入边；汇点表示整个工程的结束，通常令最后一个事件（事件 n）作为汇点，它只有入边没有出边；其余事件的编号为 2 到 n-1。

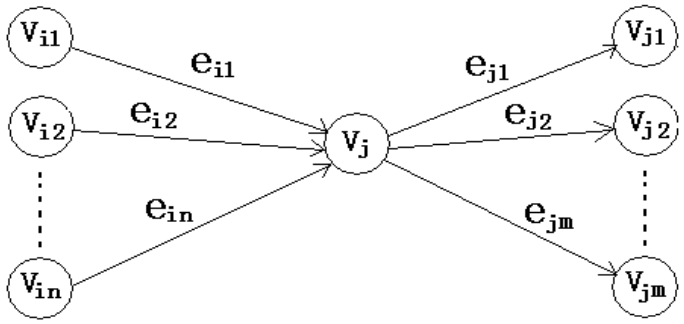
在实际应用中，AOE 网应该是没有回路的，并且存在唯一的入度为 0 的源点和唯一的出度为 0 的汇点。

下图表示一个具有 12 个活动的 AOE 网。图中有 8 个顶点，分别表示事件 0 到 7，其中，0 表示开始事件，7 表示结束事件，边上的权表示完成该活动所需要的时间。



AOE 网络要研究的问题是完成整个工程至少需要多少时间？哪些活动是影响工程进度的关键？

下面先讨论一个事件的最早发生时间和一个活动的最早开始时间。如下图，事件 V_j 必须在它的所有入边活动 e_{ik}（$1 \leqslant k \leqslant n$）都完成后才能发生。活动 e_{ik}（$1 \leqslant k \leqslant n$）的最早开始时间是与它对应的起点事件 V_{ik} 的最早发生时间。所有以事件 V_j 为起点事件的出边活动 e_{jk}（$1 \leqslant k \leqslant m$）的最早开始时间都等于事件 V_j 的最早发生时间。所以，我们可以从源点出发按照上述方法，求出所有事件的最早发生时间。



设数组 earliest[1..n]表示所有事件的最早发生时间，则我们可以按照拓扑顺序依次计算出 earliest[k]：

earliest[1]=0

earliest[k]=max{earliest[j]+dut[j,k]}

（其中，事件 j 是事件 k 的直接前驱事件，dut[j,k]表示边<j，k>上的权）

对于上图，用上述方法求 earliest[0..7]的过程如下：

earliest[0]=0

earliest[1]=earliest[0]+dut[0,1]=0+6=6

earliest[2]=earliest[0]+dut[0,2]=0+7=7

earliest[4]=max{earliest[1]+dut[1,4]，earliest[2]+dut[2,4]}

=max{6+5,7+4}

=11

earliest[3]=max{earliest[1]+dut[1,3]，earliest[4]+dut[4,3]}

=max{6+3,11+3}

=14

earliest[5]=max{earliest[3]+dut[3,5]，earliest[4]+dut[4,5]}

=max{14+2,11+4}

=16

earliest[6]=earliest[4]+dut[4,6]=11+3=14

earliest[7]=max{earliest[3]+dut[3,7]，earliest[5]+dut[5,7]，earliest[6]+dut[6,7]}

=max{14+5,16+2,14+4}

=19

最后得到的 earliest[7]就是汇点的最早发生时间，从而可知整个工程至少需要 19 天完成。

但是，在不影响整个工程按时完成的前提下，一些事件可以不在最早发生时间发生，而向后推迟一段时间，我们把事件最晚必须发生的时间称为该事件的最迟发生时间。同样，有些活动也可以推迟一段时间完成而不影响整个工程的完成，我们把活动最晚必须开始的时间称为该活动的最迟开始时间。一个事件在最迟发生时间内仍没发生，或一个活动在最迟开始时间内仍没开始，则必然会影响整个工程的按时完成。事件 V_j的最迟发生时间应该为：它的所有直接后继事件 V_{jk}（$1\leqslant k\leqslant m$）的最迟发生时间减去相应边<V_j，V_{jk}>上的权（活动 e_{jk}需要时间），取其中的最小值。且汇点的最迟发生时间就是它的最早发生时间，再按照逆拓扑顺序依次计算出所有事件的最迟发生时间，设用数组 lastest[1..n]表示，即：

lastest[n]=earliest[n]

lastest[j]=min{lastest[k]-dut[j,k]}

（其中，事件 k 是事件 j 的直接后继事件，dut[j,k]表示边<j，k>上的权）

对于上图，用上述方法求 lastest [0..7]的过程如下：

lastest[7]=earliest[7]=19

lastest[6]=lastest[7]-dut[6,7]=19-4=15

lastest[5]=lastest[7]-dut[5,7]=19-2=17

lastest[3]=min{lastest[5]-dut[3,5]，lastest[7]-dut[3,7]}

=min{17-2,19-5}

=14

lastest[4]=min{lastest[3]-dut[4,3]，lastest[5]-dut[4,5]，lastest[6]-dut[4,6]}

=min{14-3,17-4,15-3}

=11

lastest[2]=lastest[4]-dut[2,4]=11-4=7

lastest[1]=min{lastest[3]-dut[1,3]，lastest[4]-dut[1,4]}

=min{14-3,11-5}

=6

lastest[0]=min{lastest[1]-dut[0,1]，lastest[2]-dut[0,2]}

=min{6-6,7-7}

=0

计算好每个事件的最早和最迟发生时间后，我们可以很容易地算出每个活动的最早和最迟开始时间，假设分别用 actearliest 和 actlastest 数组表示，设活动 i 的两端事件分别为事件 j 和事件 k,如下所示：

活动 i

事件 j ——————————> 事件 k

则：actearliest[i]=earliest[j]

actlastest[i]=lastest[k]-dut[j,k]

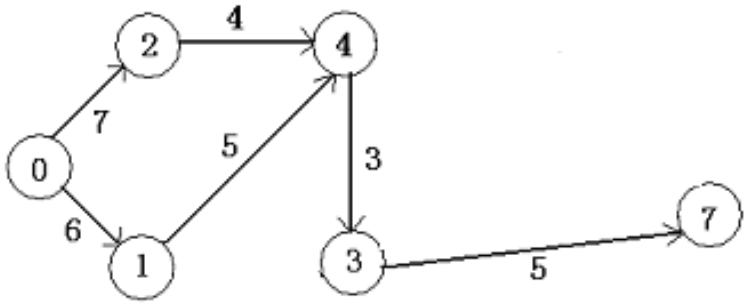
对于上图，用上述方法求得所有活动的最早和最迟开始时间如下表：

活动	<0,1>	<0,2>	<1,3>	<1,4>	<2,4>	<3,5>	<3,7>	<4,3>	<4,5>	<4,6>	<5,7>	<6,7>
最早	0	0	6	6	7	14	14	11	11	11	16	14
最迟	0	0	11	6	7	15	14	11	13	12	17	15
余量	0	0	5	0	0	1	0	0	2	1	1	1

上表中的余量（称为开始时间余量）是该活动的最迟开始时间减去最早开始时间，余量不等于 0 的活动表示该活动不一定要在最早开始时间时就进行，可以拖延一定的余量时间再进行，也不会影响整个工程的完成。而余量等于 0 的活动必须在最早开始时间时进行，而且在规定的工期内完成，否则将影响整个工程的完成。

我们把开始时间余量为 0 的活动称为“关键活动”，由关键活动所形成的从源点到汇点的每一条路径称为“关键路径”。

上图所示的 AOE 网的关键路径如下图所示。



细心的读者可能已经发现，其实关键路径就是从源点到汇点具有最大路径长度的那些路径。这很容易理解，因为整个工程的工期就是按照最长路径计算出来的。很显然，要想缩短整个工程的工期，就应该想法设法去缩短关键活动的持续时间。读者可以根据上面的思想编程求出 AOE 网的关键路径。