**目录**

[**§1 概括** 2](#_Toc392184261)

[**§1 Dijkstra算法** 3](#_Toc392184262)

[**§2 Bellman-Ford算法** 4](#_Toc392184263)

[**§3 Topological Sorting** 6](#_Toc392184264)

[**§4 Floyd-Warshall算法** 7](#_Toc392184265)

[**Attachment: Topology Sorting** 10](#_Toc392184266)

[**Attachment: Related problems** 11](#_Toc392184267)

**参考文献：**

<http://dsqiu.iteye.com/blog/1689163>

<http://blog.csdn.net/ditian1027/article/details/25874229>

<http://www.geeksforgeeks.org/shortest-path-for-directed-acyclic-graphs/>

**§1 概括**

In graph theory, the shortest path problem is the problem of finding a path between two vertices (or nodes) in a graph such that the sum of the weights of its constituent edges is minimized.

This is analogous to the problem of finding the shortest path between two intersections on a road map: the graph's vertices correspond to intersections and the edges correspond to road segments, each weighted by the length of its road segment.

The problem has the following variations:

* The single-source shortest path problem, in which we have to find shortest paths from a source vertex v to all other vertices in the graph.
* The single-destination shortest path problem, in which we have to find shortest paths from all vertices in the directed graph to a single destination vertex v. This can be reduced to the single-source shortest path problem by reversing the arcs in the directed graph.
* The all-pairs shortest path problem, in which we have to find shortest paths between every pair of vertices v, v' in the graph.

常用的最短路径算法有：Dijkstra算法、Bellman-Ford算法、Topological Sorting、Floyd-Warshall算法、Johnson算法

最短路径算法可以分为单源点最短路径和全源最短路径。

单源点最短路径有Dijkstra、Bellman-Ford算法和Topological Sorting，其中Dijkstra算法主要解决所有边的权为非负的单源点最短路径，Bellman-Ford算法可以适用权值有负值的问题。当图为DAG(directed acyclic graph)类型的图(权值可以为负)时，可以用topological sorting方法来获得O(n)的时间复杂度，比Bellman-Ford算法高效的多。

全源最短路径主要有Floyd-Warshall算法和Johnson算法，其中Floyd算法可以检测图中的负环并可以解决不包括负环的图中全源最短路径问题，Johnson算法相比Floyd-Warshall算法，效率更高。

算法性能分析

在分别讲解这四个算法之前先来理清下这个四个算法的复杂度：Dijkstra算法直接实现时间复杂度是O(n²),空间复杂度是O(n)(保存距离和路径)，二叉堆实现时间复杂度变成O((V+E)logV)，Fibonacci Heap可以将复杂度降到O(E+VlogV)；Bellman-Ford算法时间复杂度是O(V\*E)，SPFA是时间复杂度是O(kE)；Floyd-Warshall算法时间复杂度是O(n³)，空间复杂度是O(n²)；Johnson算法时间复杂度是O( V \* E \* lgd(V) )，比Floyd-Warshall算法效率高。

**§1 Dijkstra算法**

**Dijkstra算法思想**

Dijkstra算法思想为：设G=(V,E)是一个带权有向图（无向可以转化为双向有向），把图中顶点集合V分成两组，第一组为已求出最短路径的顶点集合（用S表示，初始时S中只有一个源点，以后每求得一条最短路径 , 就将 加入到集合S中，直到全部顶点都加入到S中，算法就结束了），第二组为其余未确定最短路径的顶点集合（用U表示），按最短路径长度的递增次序依次把第二组的顶点加入S中。在加入的过程中，总保持从源点v到S中各顶点的最短路径长度不大于从源点v到U中任何顶点的最短路径长度。此外，每个顶点对应一个距离，S中的顶点的距离就是从v到此顶点的最短路径长度，U中的顶点的距离，是从v到此顶点只包括S中的顶点为中间顶点的当前最短路径长度。

**Dijkstra算法具体步骤**

（1）初始时，S只包含源点，即S＝{v}，v的距离dist[v]为0。U包含除v外的其他顶点，U中顶点u距离dis[u]为边上的权值（若v与u有边） ）或∞（若u不是v的出边邻接点即没有边<v,u>）。

（2）从U中选取一个距离v(dist[k])最小的顶点k，把k，加入S中（该选定的距离就是v到k的最短路径长度）。

（3）以k为新考虑的中间点，修改U中各顶点的距离；若从源点v到顶点u（u∈ U）的距离（经过顶点k）比原来距离（不经过顶点k）短，则修改顶点u的距离值，修改后的距离值的顶点k的距离加上边上的权(即如果dist[k]+w[k,u]<dist[u]，那么把dist[u]更新成更短的距离dist[k]+w[k,u])。

（4）重复步骤（2）和（3）直到所有顶点都包含在S中(要循环n-1次)。

**Dijkstra算法实现**

╔

直接实现

最简单的实现方法就是，在每次循环中，再用一个循环找距离最短的点，然后用任意的方法更新与其相邻的边，时间复杂度显然为O(n²)

对于空间复杂度：如果只要求出距离，只要n的附加空间保存距离就可以了（距离小于当前距离的是已访问的节点，对于距离相等的情况可以比较编号或是特殊处理一下）。如果要求出路径则需要另外V的空间保存前一个节点，总共需要2n的空间。

╝

╔

二叉堆实现

使用二叉堆(Binary Heap)来保存没有扩展过的点的距离并维护其最小值，并在访问每条边的时候更新，可以把时间复杂度变成O((V+E)logV)。

当边数远小于点数的平方时，这种算法相对来说有很好的效果。但是当E=O(V2)时（有时候表现为不限制边的条数），用二叉堆的优化反倒会更慢。因为此时的复杂度是O(V+V\*2logV)，小于不用堆的实现的O(n²)的复杂度。

另外此时要用邻接表保存边，使得扩展边的总复杂度为O(E)，否则复杂度不会减小。

空间复杂度：这种算法需要一个二叉堆，及其反向指针，另外还要保存距离，所以所用空间为3V。如果保存路径则为4V。

具体思路：先将所有的点插入堆，并将值赋为极大值(maxint/maxlongint)，将原点赋值为0，通过松弛技术（relax）进行更新以及设定为扩展。

╝

╔

菲波那契堆实现

用类似的方法，使用Fibonacci Heap可以将复杂度降到O(E+VlogV)，但实现比较麻烦。因此这里暂不列举。

╝

**§2 Bellman-Ford算法**

**Bellman-Ford算法思想**

Bellman-Ford算法能在更普遍的情况下（存在负权边）解决单源点最短路径问题。对于给定的带权（有向或无向）图 G=（V,E），其源点为s，加权函数 w是 边集 E 的映射。对图G运行Bellman-Ford算法的结果是一个布尔值，表明图中是否存在着一个从源点s可达的负权回路。若不存在这样的回路，算法将给出从源点s到 图G的任意顶点v的最短路径d[v]。

**Bellman-Ford算法流程：**

（1）    初始化：将除源点外的所有顶点的最短距离估计值 d[v] ←+∞, d[s] ←0;

（2）    迭代求解：反复对边集E中的每条边进行松弛操作，使得顶点集V中的每个顶点v的最短距离估计值逐步逼近其最短距离；（运行|v|-1次）

（3）    检验负权回路：判断边集E中的每一条边的两个端点是否收敛。如果存在未收敛的顶点，则算法返回false，表明问题无解；否则算法返回true，并且从源点可达的顶点v的最短距离保存在 d[v]中。

算法描述如下：

Bellman-Ford(G,w,s) ：boolean   //图G ，边集 函数 w ，s为源点

     for each vertex v ∈ V（G） do        //初始化 1阶段

         d[v] ←+∞

     d[s] ←0;                             //1阶段结束

     for i=1 to |v|-1 do               //2阶段开始，双重循环。

        for each edge（u,v） ∈E(G) do //边集数组要用到，穷举每条边。

           If d[v]> d[u]+ w(u,v) then      //松弛判断

              d[v]=d[u]+w(u,v)               //松弛操作   2阶段结束

     for each edge（u,v） ∈E(G) do

         If d[v]> d[u]+ w(u,v) then

           Exit false

   Exit true

**描述性证明：**

   首先指出，图的任意一条最短路径既不能包含负权回路，也不会包含正权回路，因此它最多包含|v|-1条边。

   其次，从源点s可达的所有顶点如果 存在最短路径，则这些最短路径构成一个以s为根的最短路径树。Bellman-Ford算法的迭代松弛操作，实际上就是按顶点距离s的层次，逐层生成这棵最短路径树的过程。

   在对每条边进行1遍松弛的时候，生成了从s出发，层次至多为1的那些树枝。也就是说，找到了与s至多有1条边相联的那些顶点的最短路径；对每条边进行第2遍松弛的时候，生成了第2层次的树枝，就是说找到了经过2条边相连的那些顶点的最短路径……。因为最短路径最多只包含|v|-1 条边，所以，只需要循环|v|-1 次。

每实施一次松弛操作，最短路径树上就会有一层顶点达到其最短距离，此后这层顶点的最短距离值就会一直保持不变，不再受后续松弛操作的影响。（但是，每次还要判断松弛，这里浪费了大量的时间，怎么优化？单纯的优化是否可行？）

   如果没有负权回路，由于最短路径树的高度最多只能是|v|-1，所以最多经过|v|-1遍松弛操作后，所有从s可达的顶点必将求出最短距离。如果 d[v]仍保持 +∞，则表明从s到v不可达。

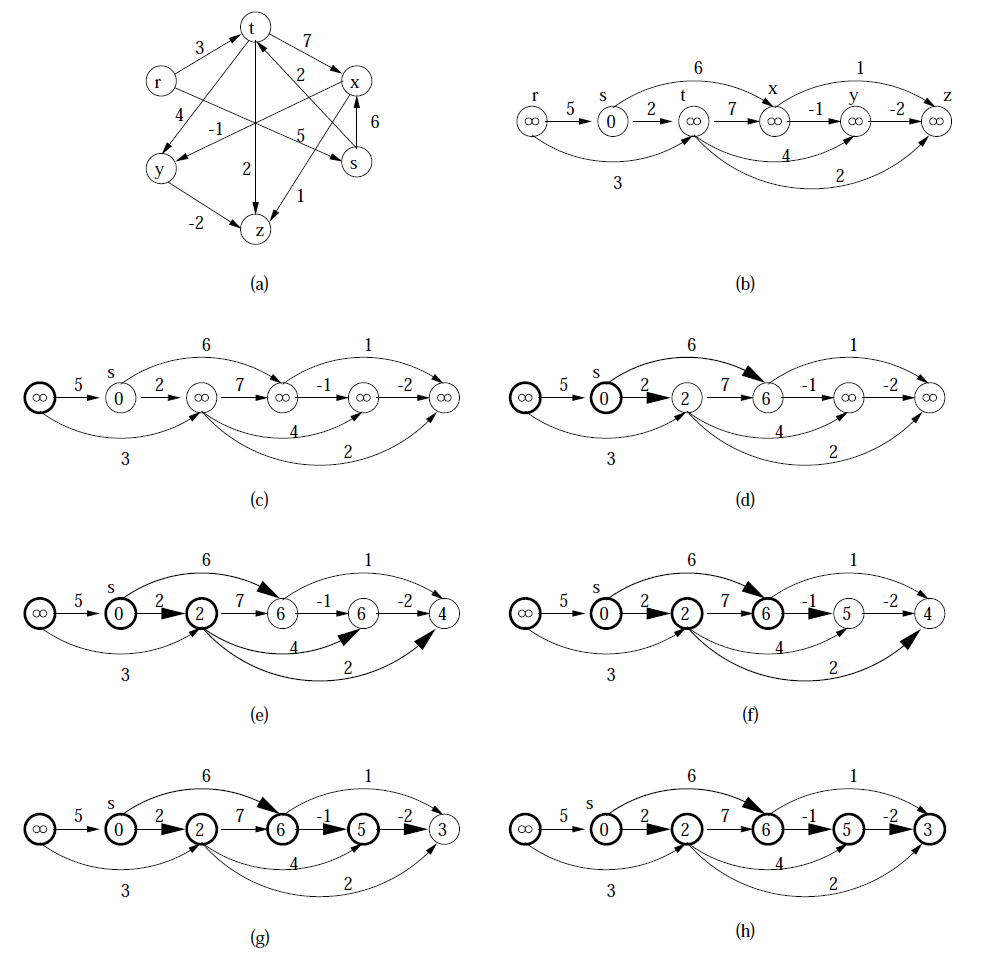
如果有负权回路，那么第 |v|-1 遍松弛操作仍然会成功，这时，负权回路上的顶点不会收敛。

╝

**§3 Topological Sorting**

Given a Weighted Directed Acyclic Graph and a source vertex in the graph, find the shortest paths from given source to all other vertices.  
For a general weighted graph, we can calculate single source shortest distances in O(VE) time using[Bellman–Ford Algorithm](http://www.geeksforgeeks.org/dynamic-programming-set-23-bellman-ford-algorithm/). For a graph with no negative weights, we can do better and calculate single source shortest distances in O(E + VLogV) time using [Dijkstra’s algorithm](http://www.geeksforgeeks.org/greedy-algorithms-set-7-dijkstras-algorithm-for-adjacency-list-representation/). Can we do even better for Directed Acyclic Graph (DAG)? We can calculate single source shortest distances in O(V+E) time for DAGs. The idea is to use [Topological Sorting](http://www.geeksforgeeks.org/topological-sorting/).

We initialize distances to all vertices as infinite and distance to source as 0, then we find a topological sorting of the graph. [Topological Sorting](http://www.geeksforgeeks.org/topological-sorting/) (see attachment of this document) of a graph represents a linear ordering of the graph (See below, figure (b) is a linear representation of figure (a) ). Once we have topological order (or linear representation), we one by one process all vertices in topological order. For every vertex being processed, we update distances of its adjacent using distance of current vertex.

Following figure is taken from [this](http://www.utdallas.edu/~sizheng/CS4349.d/l-notes.d/L17.pdf)source. It shows step by step process of finding shortest paths.  
[](http://d2o58evtke57tz.cloudfront.net/wp-content/uploads/TopologicalSort.png)

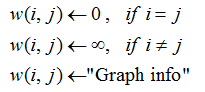
Following is complete algorithm for finding shortest distances.  
**1)** Initialize dist[] = {INF, INF, ….} and dist[s] = 0 where s is the source vertex.  
**2)** Create a toplogical order of all vertices.  
**3)**Do following for every vertex u in topological order.  
………..Do following for every adjacent vertex v of u  
………………if (dist[v] > dist[u] + weight(u, v))  
………………………dist[v] = dist[u] + weight(u, v)

We can see the reason of topological sorting is to make sure just before we traverse to/visit one node, the minimum path to the node has been calculated.

**§4 Floyd-Warshall算法**

Floyd-Warshall算法可用来计算带权图任意两顶点间的最短路径。边的权值可正可负(相比之下**Dijkstra Algorithm**要求**边的权值只能为非负**)，但**不能有负权的环**。之前学习过Floyd算法，今天重点从动态规划的角度审视它。

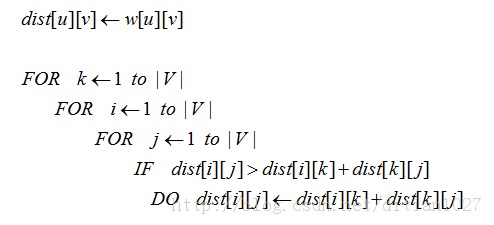
图是用邻接矩阵表示的，w(i, j)表示边(i, j)的权值。首先初始化图：



接下来看状态转移方程。Floyd算法的关键在于“**中继节点、松弛操作**”，从动态规划的思想出发，**是否添加一个新的顶点为中继**，**对应着转移前后的两种状态**，具体说来：



可以看到Floyd-Warshall是对中继顶点的状态进行动态规划。以上给出的是递归形式，常见的还是**bottom-up形式**的Floyd算法：



空间复杂度O(|V|^2)，时间复杂度O(|V|^3)。

***一个例子：poj1125 Stockbroker Grapevine***

就是原始的的Floyd算法，时间复杂度O(n^2)，空间复杂度O(n^3)。

#include <stdio.h>

#include <string.h>

#define MAXN 105

#define INFINITE (0xffff)

int dist[MAXN][MAXN]; //初始化的时候存图的信息；最终存放的是Floyd-Warshall处理后的结果

int EveryMax[MAXN]; //记录每一个stock broker传播时间的最大值，以便在其中找到最小值。数组下表是stock broker的编号，对应的值是传播时间

void InitGraph(int n)

{

for (int i=1; i<=n; ++i)

{

for (int j=1; j<=n; ++j)

{

dist[i][j]=INFINITE;

}

}

for (int i=1; i<=n; ++i)

{

dist[i][i]=0;

}

int pairs, to, wght;

for (int i=1; i<=n; ++i)

{

scanf("%d", &pairs);

for (int j=0; j<pairs; ++j)

{

scanf("%d %d", &to, &wght);

dist[i][to]=wght;

}

}

}

void FloydWarshall(int n)

{

//the essential of Floyd-Warshall algorithm

for (int k=1; k<=n; ++k) //choosing relay vertices

{

for (int i=1; i<=n; ++i) //from i

{

for (int j=1; j<=n; ++j) //to j

{

//perform relaxation

if(dist[i][j]>dist[i][k]+dist[k][j])

{

dist[i][j]=dist[i][k]+dist[k][j];

}

}

}

}

memset(EveryMax, 0, sizeof(EveryMax));

for (int i=1; i<=n; ++i)

{

for (int j=1; j<=n; ++j)

{

if(dist[i][j]>EveryMax[i]) EveryMax[i]=dist[i][j];

}

}

int minofmax=INFINITE;

int minidx;

for (int i=1; i<=n; ++i)

{

if(EveryMax[i]<minofmax)

{

minofmax=EveryMax[i];

minidx=i;

}

}

printf("%d %d\n", minidx, minofmax);

}

int main()

{

int n;

while (1)

{

scanf("%d", &n);

if (!n) break;

InitGraph(n);

FloydWarshall(n);

}

return 0;

}

**Attachment: Topology Sorting**

Topological sorting for Directed Acyclic Graph (DAG) is a linear ordering of vertices such that for every directed edge uv, vertex u comes before v in the ordering. Topological Sorting for a graph is not possible if the graph is not a DAG.

For example, a topological sorting of the following graph is “5 4 2 3 1 0″. There can be more than one topological sorting for a graph. For example, another topological sorting of the following graph is “4 5 2 3 1 0″. The first vertex in topological sorting is always a vertex with in-degree as 0 (a vertex with no in-coming edges).

[](http://d2o58evtke57tz.cloudfront.net/wp-content/uploads/graph.png)

**Topological Sorting vs Depth First Traversal (DFS)**:  
In [DFS](http://www.geeksforgeeks.org/depth-first-traversal-for-a-graph/), we print a vertex and then recursively call DFS for its adjacent vertices. In topological sorting, we need to print a vertex before its adjacent vertices. For example, in the given graph, the vertex ’5′ should be printed before vertex ’0′, but unlike [DFS](http://www.geeksforgeeks.org/depth-first-traversal-for-a-graph/), the vertex ’4′ should also be printed before vertex ’0′. So Topological sorting is different from DFS. For example, a DFS of the above graph is “5 2 3 1 0 4″, but it is not a topological sorting

**Algorithm to find Topological Sorting:**  
We recommend to first see implementation of DFS [here](http://www.geeksforgeeks.org/depth-first-traversal-for-a-graph/). We can modify [DFS](http://www.geeksforgeeks.org/depth-first-traversal-for-a-graph/)to find Topological Sorting of a graph. In [DFS](http://www.geeksforgeeks.org/depth-first-traversal-for-a-graph/), we start from a vertex, we first print it and then recursively call DFS for its adjacent vertices. In topological sorting, we use a temporary stack. We don’t print the vertex immediately, we first recursively call topological sorting for all its adjacent vertices, then push it to a stack. Finally, print contents of stack. Note that a vertex is pushed to stack only when all of its adjacent vertices (and their adjacent vertices and so on) are already in stack.#include <stdio.h>

**Attachment: Related problems**

For shortest path problems in [computational geometry](http://en.wikipedia.org/wiki/Computational_geometry), see [Euclidean shortest path](http://en.wikipedia.org/wiki/Euclidean_shortest_path).

The [travelling salesman problem](http://en.wikipedia.org/wiki/Traveling_salesman_problem) is the problem of finding the shortest path that goes through every vertex exactly once, and returns to the start. Unlike the shortest path problem, which can be solved in polynomial time in graphs without negative cycles, the travelling salesman problem is [NP-complete](http://en.wikipedia.org/wiki/NP-complete) and, as such, is believed not to be efficiently solvable for large sets of data (see [P = NP problem](http://en.wikipedia.org/wiki/P_%3D_NP_problem)). The problem of [finding the longest path](http://en.wikipedia.org/wiki/Longest_path_problem) in a graph is also NP-complete.

The [Canadian traveller problem](http://en.wikipedia.org/wiki/Canadian_traveller_problem) and the stochastic shortest path problem are generalizations where either the graph isn't completely known to the mover, changes over time, or where actions (traversals) are probabilistic.

The shortest multiple disconnected path [[7]](http://en.wikipedia.org/wiki/Shortest_path_problem#cite_note-7) is a representation of the primitive path network within the framework of [Reptation theory](http://en.wikipedia.org/wiki/Reptation_theory).

The [widest path problem](http://en.wikipedia.org/wiki/Widest_path_problem) seeks a path so that the minimum label of any edge is as large as possible.

#include <string.h>

#define MAXN 105

#define INFINITE (0xffff)

int dist[MAXN][MAXN]; //初始化的时候存图的信息；最终存放的是Floyd-Warshall处理后的结果

int EveryMax[MAXN]; //记录每一个stock broker传播时间的最大值，以便在其中找到最小值。数组下表是stock broker的编号，对应的值是传播时间

void InitGraph(int n)

{

for (int i=1; i<=n; ++i)

{

for (int j=1; j<=n; ++j)

{

dist[i][j]=INFINITE;

}

}

for (int i=1; i<=n; ++i)

{

dist[i][i]=0;

}

int pairs, to, wght;

for (int i=1; i<=n; ++i)

{

scanf("%d", &pairs);

for (int j=0; j<pairs; ++j)

{

scanf("%d %d", &to, &wght);

dist[i][to]=wght;

}

}

}

void FloydWarshall(int n)

{

//the essential of Floyd-Warshall algorithm

for (int k=1; k<=n; ++k) //choosing relay vertices

{

for (int i=1; i<=n; ++i) //from i

{

for (int j=1; j<=n; ++j) //to j

{

//perform relaxation

if(dist[i][j]>dist[i][k]+dist[k][j])

{

dist[i][j]=dist[i][k]+dist[k][j];

}

}

}

}

memset(EveryMax, 0, sizeof(EveryMax));

for (int i=1; i<=n; ++i)

{

for (int j=1; j<=n; ++j)

{

if(dist[i][j]>EveryMax[i]) EveryMax[i]=dist[i][j];

}

}

int minofmax=INFINITE;

int minidx;

for (int i=1; i<=n; ++i)

{

if(EveryMax[i]<minofmax)

{

minofmax=EveryMax[i];

minidx=i;

}

}

printf("%d %d\n", minidx, minofmax);

}

int main()

{

int n;

while (1)

{

scanf("%d", &n);

if (!n) break;

InitGraph(n);

FloydWarshall(n);

}

return 0;

}