Floyd算法

Floyd-Warshall算法（Floyd-Warshall algorithm）是解决任意两点间的最短路径的一种算法，可以正确处理有向图或负权的最短路径问题，同时也被用于计算有向图的传递闭包。

Floyd-Warshall算法的时间复杂度为O(N^3)，空间复杂度为O(N^2)。

1) 算法思想原理：

Floyd算法是一个经典的动态规划算法。用通俗的语言来描述的话，首先我们的目标是寻找从点i到点j的最短路径。

从动态规划的角度看问题，我们需要为这个目标重新做一个诠释（这个诠释正是动态规划最富创造力的精华所在）

从任意节点i到任意节点j的最短路径不外乎2种可能，1是直接从i到j，2是从i经过若干个节点k到j。所以，我们假设Dis(i,j)为节点u到节点v的最短路径的距离，对于每一个节点k，我们检查Dis(i,k) + Dis(k,j) < Dis(i,j)是否成立，如果成立，证明从i到k再到j的路径比i直接到j的路径短，我们便设置Dis(i,j) = Dis(i,k) + Dis(k,j)，这样一来，当我们遍历完所有节点k，Dis(i,j)中记录的便是i到j的最短路径的距离。

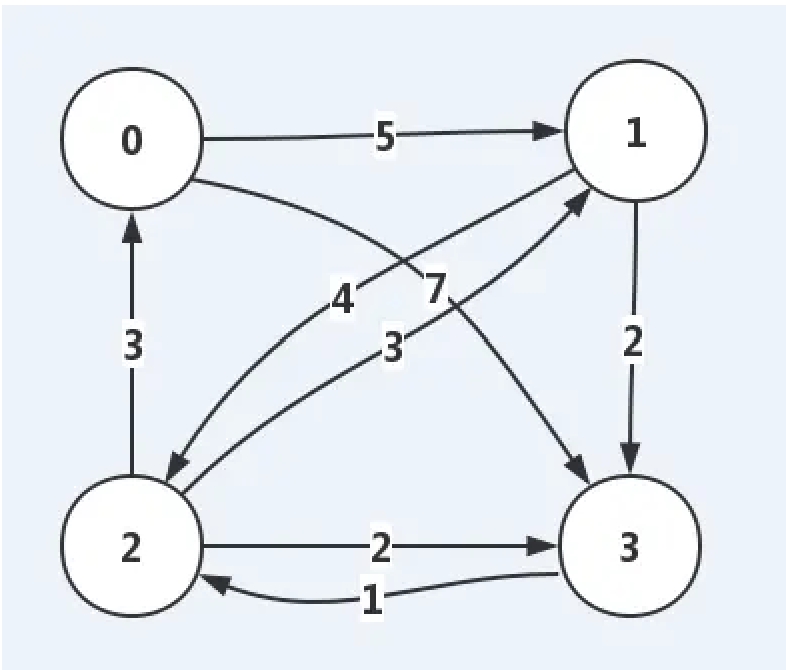
2) 算法描述：

a. 从任意一条单边路径开始。所有两点之间的距离是边的权，如果两点之间没有边相连，则权为无穷大。

b. 对于每一对顶点 u 和 v，看看是否存在一个顶点 w 使得从 u 到 w 再到 v 比已知的路径更短。如果是更新它。

Floyd算法过程矩阵的计算

）如图：存在【0,1,2,3】 4个点，两点之间的距离就是边上的数字，如果两点之间，没有边相连，则无法到达，为无穷大。 b）要让任意两点（例如从顶点a点到顶点b）之间的路程变短，只能引入第三个点（顶点k），并通过这个顶点k中转即a->k->b，才可能缩短原来从顶点a点到顶点b的路程。那么这个中转的顶点k是0~n中的哪个点呢？



算法过程

准备

1）如图 0->1距离为5，0->2不可达，距离为∞，0->3距离为7……依次可将图转化为邻接矩阵（主对角线，也就是自身到自身，我们规定距离为0，不可达为无穷大），如图矩阵 用于存放任意一对顶点之间的最短路径权值。

距离	0	1	2	3
0	0	5	∞	7
1	∞	0	4	2
2	3	3	0	2
3	∞	∞	1	0

2）再创建一个二维数组Path路径数组，用于存放任意一对顶点之间的最短路径。每个单元格的内容表示从i点到j点途经的顶点。（初始还未开始查找，默认-1）

Path	0	1	2	3
0	-1	-1	-1	-1
1	-1	-1	-1	-1
2	-1	-1	-1	-1
3	-1	-1	-1	-1

$$A_{-1}=\begin{bmatrix}0 & 5 & \infty & 7\\ \infty & 0 & 4 & 2\\ 3 & 3 & 0 & 2\\ \infty & \infty & 1 & 0\end{bmatrix}\qquad Path_{-1}=\begin{bmatrix}-1 & -1 & -1 & -1\\ -1 & -1 & -1 & -1\\ -1 & -1 & -1 & -1\\ -1 & -1 & -1 & -1\end{bmatrix}$$

开始查找

1）列举所有的路径（自己到自己不算）

距离	0	1	2	3
0	0	5	∞	7
1	∞	0	4	2
2	3	3	0	2
3	∞	∞	1	0

即为： 0 -> 1， 0 -> 2， 0 -> 3，
1 -> 0， 1 -> 2， 1 -> 3 , 2 -> 0， 1 -> 1， 1 -> 3 转化成二元数组即为：{0，1}，{0，2}，{0，3}，{1，0}，{1，2}，{1，3}，{2，0}，{2，1}，{2，3}，{3，0}，{3，1}，{3，2}

2）选择编号为0的点为中间点

{0，1}，{0，2}，{0，3}，{1，0}，{1，2}，{1，3}，{2，0}，{2，1}，{2，3}，{3，0}，{3，1}，{3，2}
从上面中二元组集合的第一个元素开始，循环执行以下过程：

```
1. 用i，j两个变量分别指向二元组里的两个元素，比如{0，1}这个二元组，i指向0；j指向1
2. 判断 (A[ i ][ 0 ]+A[ 0 ][ j ] ) < A[ i ][ j ] （即判断 i -> j，i点到j点的距离是否小于从0点中转的距离），如果false，则判断下一组二元数组。
3. 如果表达式为真，更新A[ i ] [ j ]的值为A[ i ] [ 0 ] + A[ 0 ] [ j ]，Path[ i ] [ j ]的值为点0（即设置i到j要经过0点中转）
```

{0，1}按照此过程执行之后，

{0，1}，{0，2}，{0，3}，{1，

距离	0	1	2	3
0	0	5	∞	7
1	∞	0	4	2
2	3	3	0	2
3	∞	∞	1	0

0->0 + 0->1的距离不小于0->1

Path	0	1	2	3
0	-1	-1	-1	-1
1	-1	-1	-1	-1
2	-1	-1	-1	-1
3	-1	-1	-1	-1

0->0 + 0->1的距离不小于0->1， **下一组{0，2}，{0，3}， {1，0}，{2，0}，{3，0}也同理。**

{1，2}按照此过程执行，A[1,0] 无穷大， A[0,2]也是无穷大，而A[1,4] = 4，**则1点到2点肯定不会从0点中转。**

A[1][0]无穷大同理**下一组{1，2}， {1，3}也同理。**

{2，1}按照此过程执行，A[2][0] = 3 ,A[0][1]=5， A[2][1] = 3那么 A[2][0]+ ,A[0][1] > A[2][1] ………… 依次类推，遍历二元组集合，没有0点适合做中转的

3）选择编号为1的点为中间点

4）选择编号为2的点为中间点

依次类推，遍历二元组集合{0，1}，{0，2}，{0，3}，{1，0}，{1，2}，{1，3}，{2，0}，{2，1}，{2，3}，{3，0}，{3，1}，{3，2}，当遍历{3，0}时，A[3][2] = 1 ,A[2][0]=3， A[3][0] = 不可达，那么 2点适合做从3点到0点之间的中转点。 设置距离矩阵A[3][0] = 1+3 =4， Path矩阵Path[3][0] = 2点，表示从3到0在2点中转，距离最近。

点2中转 {3，0}				
距离	0	1	2	3
0	0	5	∞	7
1	∞	0	4	2
2	3	3	0	2
3	4	∞	1	0
Path	0	1	2	3
0	-1	-1	-1	-1
1	-1	-1	-1	-1
2	-1	-1	-1	-1
3	2	-1	-1	-1

如图表示（红色单元格），从3到0，最近距离为4，在2点中转 。

依次类推，遍历完二元组集合

点2中转 {3，1}				
距离	0	1	2	3
0	0	5	∞	7
1	∞	0	4	2
2	3	3	0	2
3	4	4	1	0
Path	0	1	2	3
0	-1	-1	-1	-1
1	-1	-1	-1	-1
2	-1	-1	-1	-1
3	2	2	-1	-1

5）选择编号为3的点为中间点，最终结果

依次类推，遍历二元组集合，直到所有的顶点都做过一次中间点为止。

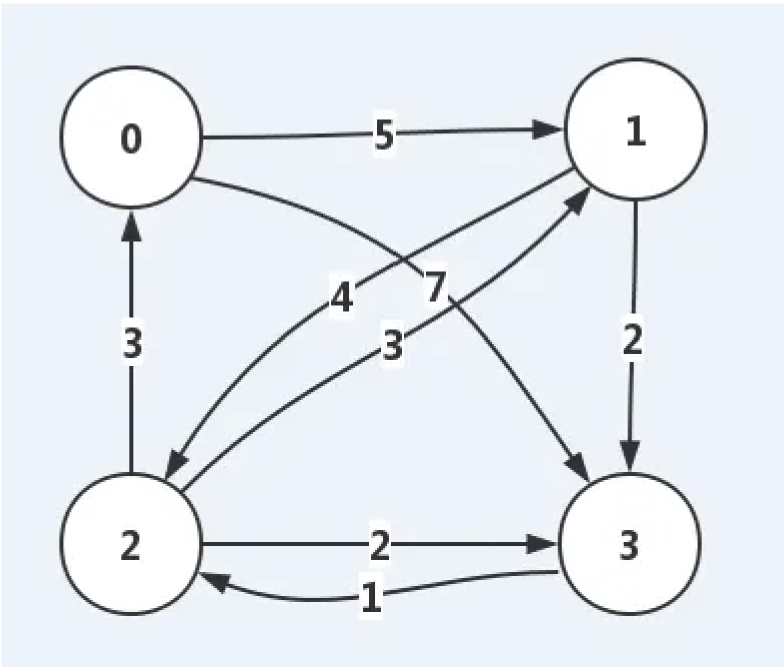
最终结果

距离	0	1	2	3
0	0	5	8	7
1	6	0	3	2
2	3	3	0	2
3	4	4	1	0

Path	0	1	2	3
0	-1	-1	3	-1
1	3	-1	3	-1
2	-1	-1	-1	-1
3	2	2	-1	-1

6）根据最终结果，就可以知道任意2点的最短距离和路径

比如1点到2点怎么走？根据路径Path矩阵,Path[1][2] = 3，表示从点3中转，即 1-> 3 ->2



6）如果中转点不止1个呢?

有时候不只通过一个点，而是经过两个点或者更多点中转会更短，即a->k1->k2b->或者a->k1->k2…->k->i…->b。 比如顶点1到顶点0，我们看数组Path Path[1][0] = 3，说明顶点3是中转点，那么再从3到0 Path[3][0] = 2，说明从3到0，顶点2是中转点，然后在从2到0 Path[2][0] = -1，说明顶点2到顶点0没有途径顶点，也就是说，可以由顶点2直接到顶点0，即它们有边连接。

最终，最短路径为1->3->2->0，距离为 A[1][0] = 6 。 显然，这是一个逐层递进，递归的过程。

代码实现

核心代码只有 4 行。

```
//Floy算法实现
void Floy(MGraph *G, int path[][maxsize], int A[][maxsize]){
    int i,j,k;
    for(i = 0; i < G->vexnum; i++){      //初始化
        for(j = 0; j < G->vexnum; j++){
            path[i][j] = -1;
            A[i][j] = G->edge[i][j];
        }
    }
    for(k=0;k < G->vexnum; k++){         //floy算法关键步骤
        for(i = 0; i < G->vexnum; i++){
            for(j = 0; j < G->vexnum; j++){
                if(A[i][j] > A[i][k]+A[k][j]){      //增加k作为中间节点是否会增加最短
路径长度
                    path[i][j] = k;
                    A[i][j] = A[i][k]+A[k][j];
                }
            }
        }
    }
}
```