## 1算法的功能：

通过一个图的权值矩阵，求出它的每两个点之间的最短路径矩阵

## Floyd算法的基本思想如下：

我们的目标是寻找从结点i到结点j的最短路径，从动态规划的角度来看，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A到任意节点B的最短路径不外乎2种可能：

1. 直接从A到B。
2. 从A经过若干个节点X到B。（这正是对目标的新的诠释）

所以，我们假设Dis(AB)为节点A到节点B的最短路径的距离，对于每一个节点X，我们检查Dis(AX) + Dis(XB) < Dis(AB)是否成立，如果成立，证明从A到X再到B的路径比A直接到B的路径短，我们便设置Dis(AB) = Dis(AX) + Dis(XB)，这样一来，当我们遍历完所有节点X，Dis(AB)中记录的便是A到B的最短路径的距离。

## 2.算法代码

很简单吧，代码看起来可能像下面这样：

for ( int i = 0; i < 节点个数; ++i )// 我们要求的是从i到j的最短路径，k代表的是i和j之间的中间结点

{

for ( int j = 0; j < 节点个数; ++j )

{

for ( int k = 0; k < 节点个数; ++k )

{

if ( Dis[i][k] + Dis[k][j] < Dis[i][j] )

{

// 找到更短路径

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

}

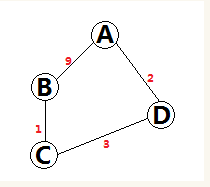
}

}

}

但是这里我们要注意循环的嵌套顺序，如果把检查所有节点X放在最内层，那么结果将是不正确的，为什么呢？因为这样便过早的把i到j的最短路径确定下来了，而当后面存在更短的路径时，已经不再会更新了。

现分析如下：



如果我们在最内层检查所有节点X（结点x代表A和B之间的中间结点），那么对于A->B，我们只能发现一条路径（而实际上不止这一条路径，还存在A->D->C->B这条路径，显然上述的循环嵌套次序会造成与实际不符的结果），就是A->B，路径距离为9。而这显然是不正确的，真实的最短路径是A->D->C->B，路径距离为6。造成错误的原因就是我们把检查所有节点X放在最内层，造成过早的把A到B的最短路径确定下来了，当确定A->B的最短路径时Dis(AC)尚未被计算。

所以，我们需要改写循环顺序，如下：

for ( int k = 0; k < 节点个数; ++k )// k代表的是i和j之间的中间结点

{

for ( int i = 0; i < 节点个数; ++i )

{

for ( int j = 0; j < 节点个数; ++j )

{

if ( Dis[i][k] + Dis[k][j] < Dis[i][j] )

{

// 找到更短路径

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

}

}

}

}

注意此种写法的优点：对于每一个节点X，我们都会把所有的i到j处理完毕后才继续检查下一个节点。

那么接下来的问题就是，我们如何找出最短路径呢？这里需要借助一个辅助数组Path，它是这样使用的：Path(AB)的值如果为P，则表示A节点到B节点的最短路径是A->...->P->B（表示A到B的最短距离所必须经过的结点）。这样一来，假设我们要找A->B的最短路径，那么就依次查找，假设Path(AB)的值为P，那么接着查找Path(AP)，假设Path(AP)的值为L，那么接着查找Path(AL)，假设Path(AL)的值为A，则查找结束，最短路径为A->L->P->B。

如何填充Path的值呢？很简单，当我们发现Dis(AX) + Dis(XB) < Dis(AB)成立时，就要把最短路径改为A->...->X->...->B，而此时，Path(XB)的值是已知的，所以，Path(AB) = Path(XB)。