**《离散数学》课程实验报告**

# 《离散数学》课程实验报告6

利用warshall算法求解传递闭包

作 者 姓 名： 毛凌骏

学 号： 2053058

指 导 教 师： 唐剑锋

学院、 专业： 软件学院 软件工程

同济大学

Tongji University

1. **题目简介**

**1.1实验内容：**

本实验的目的是希望利用warshall算法求解有限集上给定关系的传递闭包，熟悉关系的闭包运算。对于输入的任意关系矩阵，程序首先应当判断关系矩阵的正确性，若错误比如出现非布尔类型的矩阵元素则报错，否则打印warshall算法的求解结果。

**1.2实验环境：**

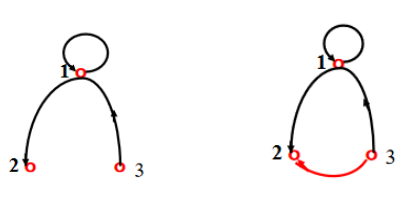
采用C＋＋编程语言，Visual Studio Code 2019实验环境实现。

1. **解题思路**

**2.1 传递闭包的概念与求解：**

传递闭包是指包含原关系且满足传递性的最小关系。例如，如果X是人的集合而R是关系“为父子”，则R的传递闭包是关系“x是y的祖先”。再比如，如果X是空港的集合而关系xRy为“从空港x到空港y有直航”，则R的传递闭包是“可能经一次或多次航行从x飞到 y”。对于任何关系 R，R 的传递闭包总是存在的，[传递关系](https://baike.baidu.com/item/%E4%BC%A0%E9%80%92%E5%85%B3%E7%B3%BB?fromModule=lemma_inlink" \t "https://baike.baidu.com/item/%E4%BC%A0%E9%80%92%E9%97%AD%E5%8C%85/_blank)的任何家族的[交集](https://baike.baidu.com/item/%E4%BA%A4%E9%9B%86/13014743?fromModule=lemma_inlink" \t "https://baike.baidu.com/item/%E4%BC%A0%E9%80%92%E9%97%AD%E5%8C%85/_blank)也是传递的。进一步的，可以证明至少存在一个包含 R 的传递关系，能够包含 R 的所有传递关系的交集。

关系的传递闭包在关系图中体现为可达关系，若两个点之间通过若干边相连，则认为这两点是可达的，因此可在这两点间直接连边代表可达关系。在关系矩阵中该思路则体现为若矩阵第i行第j列的逻辑值为1，且矩阵第j行第k列的逻辑值为1，则将关系矩阵第i行第k列置1。假设关系矩阵是n阶的，该过程需要反复执行n-1次，最终得到的即为原关系的传递闭包。求关系传递闭包的传统算法时间复杂度为，为简化算法，计算机科学家们研发出了时间复杂度仅为的warshall算法，。

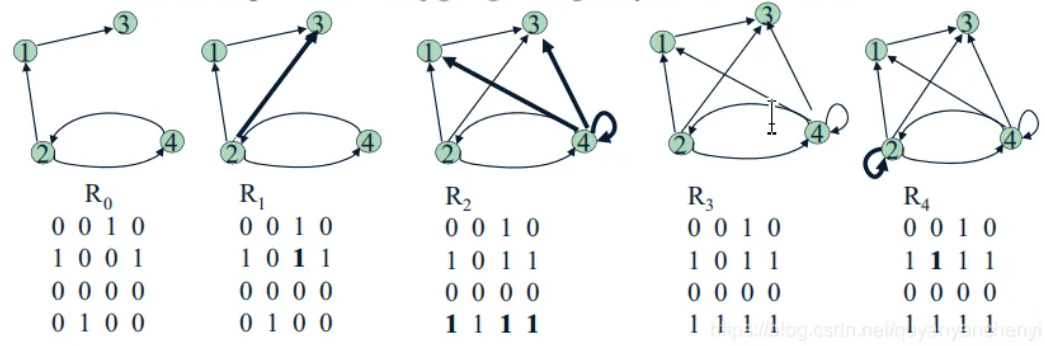


**2.2 warshall算法核心思想：**

warshall算法相较于传统算法复杂度上少了一阶，其使用了一条路径“内点”的概念。即如a，b，c，.....，x样的一条路径，除去a和x之外的所有端点称为这条路径的内点。具体的操作方法是以R为开头构造一系列矩阵，分别是W0,W1,W2,W3,W4 .....，其中W0= MR。和传统算法不同，warshall算法的高阶矩阵的值并不是由前一阶的布尔幂运算得来。而是判断如果存在一条从vi到vj的路径且这条路径的所有内点都在集合{v1，v2，....，vk}内，那么其值为1，否则为0。

可以看到，因为Wk-1中的“1”也必然满足于Wk，所以Wk-1中的“1”可以直接继承到Wk之中。而对于Wk-1中的0，从前段的定义中我们可以看到，对于wij[k-1]为0，而wij[k]为1的情况，当且仅当wik[k-1]=1且wkj[k-1]=1。使用路径的说法，即存在一条从vi到vk和一条从vk到vj的路径，使得当内点集合扩大到vk时才满足了前段定义的第二条条件。这里的下标k不再泛指k≤n的正整数，而是实指当前正在运算的Wk的k。

总结来说，传统算法是通过n阶布尔幂来化简多重传递，并在过程中进行了矩阵合并的运算以保证所有路径都被保留。而warshall算法在每阶矩阵的每个元素上，最多只进行2次布尔运算就可以取代标准算法的一个复杂度为的矩阵运算，因此效率得以提升。warshall算法的算法描述及步骤如下：



(1)置新矩阵A＝M；

(2)i＝1；

(3)对所有j如果A[j，i]＝1，则对k＝1，2，…，n，A[j，k]＝A[j，k]∨A[i，k]；

(4)i加1；（i是行，j是列）

(5)如果i≤n，则转到步骤3)，否则停止。

1. **数据结构**

计算机存储二元关系即为存储关系图，有两种主要方法，分别是邻接矩阵与邻接表。邻接矩阵通过建立一个行列数为顶点个数的二维数组来存储图的相关信息，邻接表则采用链表的方式。在一个顶点数为n,边数为e的图中，二者的时间复杂度比较如下：

|  |  |  |
| --- | --- | --- |
|  | 邻接矩阵 | 邻接表 |
| 构造功能 |  |  |
| 查找邻接对象功能 |  |  |
| 判断是否邻接功能 |  |  |
| 占用空间 |  |  |

结合多方面因素考虑，本项目采用邻接矩阵的形式存储。

为了方便操作，程序定义了一个关系类Relation，用于存放二元关系以及操作方法。Relation类的private属性包括关系矩阵的行列数row、cal，以及指向一个二维数组的指针relation。Relation的public属性主要是对于关系矩阵的操作，包括构造函数、析构函数、求自反闭包函数、求对称闭包函数、求传递闭包函数、打印矩阵函数。程序通过Relation类的实例化与调用实现对关系矩阵的闭包运算。

//关系类

**class** Relation {

**private**:

**int** row, cal;                       //行列数

**int**\*\* relation;                     //关系矩阵

**public**:

    Relation();                         //构造函数

    ~Relation();                        //析构函数

**void** transitive();                  //求传递闭包

**void** printMatrix(**int**\*\* matrix);     //打印矩阵

};

1. **核心算法**

**4.1 关系矩阵的输入：**

关系矩阵的输入在Relation类对象的构造函数中实现，用户输入所期望矩阵的行列数，赋值给Relation类的row、cal属性。程序根据所输入的row与cal值动态开辟关系矩阵的大小并进行矩阵元素的输入。此时矩阵元素应该为布尔型，当程序检测到用户不合法的输入时应给出提示信息并让用户再次输入。

//构造函数

Relation::Relation() {

    //输入行数

    cout << "请输入矩阵的行数:";

**while** (1) {

        cin >> row;

**if** (cin.fail()) {

            cout << "输入错误，请重新输入:" << endl;

            cin.clear();

**char** t;

**while** ((t = cin.get()) != '\n');

        }

**else** **if** (row < 0 || row>100) {

            cout << "函数应该在0-100之间，请重新输入:" << endl;

**char** t;

**while** ((t = cin.get()) != '\n');

        }

**else**

**break**;

    }

    //输入列数

    cout << "请输入矩阵的列数:";

**while** (1) {

        cin >> cal;

**if** (cin.fail()) {

            cout << "输入错误，请重新输入:" << endl;

            cin.clear();

**char** t;

**while** ((t = cin.get()) != '\n');

        }

**else** **if** (cal < 0 || cal>100) {

            cout << "函数应该在0-100之间，请重新输入:" << endl;

**char** t;

**while** ((t = cin.get()) != '\n');

        }

**else**

**break**;

    }

    //输入关系矩阵

    relation = **new** **int**\*[row];

**for** (**int** i = 0; i < row; i++) {

        relation[i] = **new** **int**[cal];

    }

    cout << "请输入关系矩阵" << endl;

**for** (**int** i = 0; i < row; i++) {

        cout << "请输入矩阵的第" << i + 1 << "行元素(元素以空格分隔):";

**for** (**int** j = 0; j < cal; j++) {

            cin >> relation[i][j];

**if** (cin.fail() || (relation[i][j] != 0 && relation[i][j] != 1)) {

                cin.clear();

                cout << "输入错误，请重新输入第" << i + 1 << "行元素:";

**char** t;

**while** ((t = cin.get()) != '\n');

                j = -1;

            }

        }

    }

}

**4.2 warshall算法的实现：**

传递闭包运算是将原关系矩阵所有可达的节点之间置1，若采用传统方法的话共需要四重循环。每一次扫描关系矩阵所有的元素，若第i行第j列元素为1，则遍历扫描第j行的元素，查找是否有第j行第k列的元素为1，若存在的话则将第i行第k列元素置1。可以看出，该操作每执行一次是求路径长度不大于n的可达矩阵，由于路径长度最长为n-1，因此该循环需要执行n-1次，以求得原关系的传递闭包。

相较于传统算法，warshall算法只需三重循环，因为warshall算法在每阶矩阵的每个元素上，最多只进行2次布尔运算就可以取代标准算法的一个复杂度为的矩阵运算，因此可以减少一重循环。程序首先遍历矩阵元素，之后对每一个第j行第i列的元素去遍历第i行，若第j行第i列的元素与第i行第k列的元素都为1，则将第j行第k列元素置1，相当于每次以一列为模板去更新每一行，如此循环便可得到原先矩阵的传递闭包，可以证得此时s满足传递闭包的三个性质，即包含A、满足传递性、包含于任何一个满足上述两条性质的集合。传递闭包运算的实现代码如下：

//求关系矩阵的传递闭包

**void** Relation::transitive() {

    //初始化传递闭包矩阵

**int**\*\* transitive = **new** **int**\* [row];

**for** (**int** i = 0; i < row; i++) {

        transitive[i] = **new** **int**[cal];

    }

**for** (**int** i = 0; i < row; i++) {

**for** (**int** j = 0; j < cal; j++) {

            transitive[i][j] = relation[i][j];

        }

    }

    //求传递闭包

**for** (**int** i = 0; i < cal; i++) {

**for** (**int** j = 0; j < row; j++) {

**for** (**int** k = 0; k < cal; k++) {

**if** (transitive[i][k] && transitive[j][i] == 1)

                    transitive[j][k] = 1;

            }

        }

    }

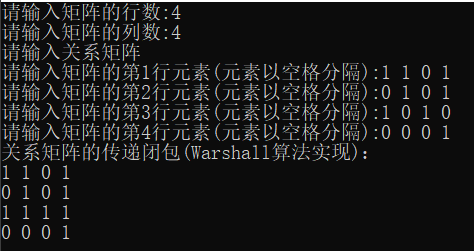
    //打印传递闭包

    printMatrix(transitive);

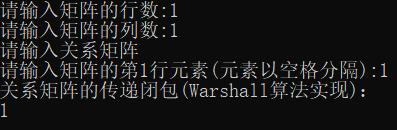
}

1. **实验测试**

**5.1 常规测试：**

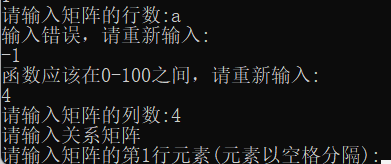


**5.2 边界测试：**

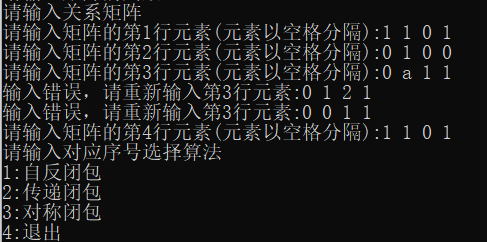


**5.3 错误测试：**

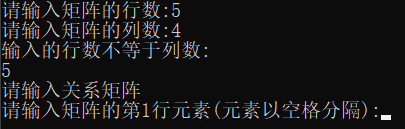
输入行列数错误：



输入矩阵元素错误：



输入行数不等于列数：



1. **心得体会**

本实验中，我们学会了如何使用warshall算法代替传统算法求解关系的传递闭包。warshall算法是一种求解可达问题的高效算法，在传统算法的基础上减少了一重循环，主要思想是通过布尔运算取代传统算法的一个复杂度为的矩阵运算，对于关系运算中传递闭包的求解具有重要意义。从传统方法与warshall算法的对比使用中我们可以看出，数据结构和算法的优化是一个探索的过程，随着时间的推移算法会一步步被优化。优化的想法有很多，虽然成功的方向只有几个，但也值得我们不断尝试。

其实对于传递闭包的求解，warshall算法仍有可改进之处。比如对于关系矩阵的运算，我们完全可以用位运算代替矩阵运算。常见的位运算有按位与、按位或、按位取反等。因为程序中的所有数在[计算机内存](https://baike.baidu.com/item/%E8%AE%A1%E7%AE%97%E6%9C%BA%E5%86%85%E5%AD%98/9021807?fromModule=lemma_inlink" \t "https://baike.baidu.com/item/%E4%BD%8D%E8%BF%90%E7%AE%97/_blank)中都是以二进制的形式[储存](https://baike.baidu.com/item/%E5%82%A8%E5%AD%98/2446499?fromModule=lemma_inlink" \t "https://baike.baidu.com/item/%E4%BD%8D%E8%BF%90%E7%AE%97/_blank)的，位运算就是直接对整数在内存中的二进制位进行[操作](https://baike.baidu.com/item/%E6%93%8D%E4%BD%9C/33052?fromModule=lemma_inlink" \t "https://baike.baidu.com/item/%E4%BD%8D%E8%BF%90%E7%AE%97/_blank)，这种高效并行的处理模式效率远远高于循环遍历矩阵运算的模式。因此，在实际工程中涉及传递闭包我们通常是直接用二进制数位运算求解，可以大大提升算法的性能。