**《离散数学》课程实验报告**

# 《离散数学》课程实验报告3

求关系的自反、对称、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关系[矩阵](https://so.csdn.net/so/search?q=%E7%9F%A9%E9%98%B5&spm=1001.2101.3001.7020" \t "https://blog.csdn.net/bigotry__/article/details/_blank)求解有限集上给定关系的自反、对称和传递闭包，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，否则再次输入操作符，通过输入不同的操作符分别测试关系矩阵的自反、对称和传递闭包运算，其中传递闭包的运算采用传统方法即复杂度为的遍历方法，Warshall算法的实现会在实验6中实现并说明。

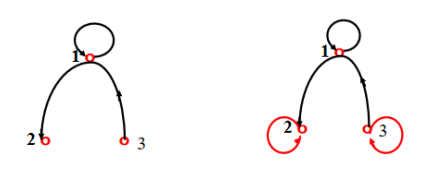
**1.2实验环境：**

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

1. **解题思路**

**2.1 自反闭包的概念与求解：**

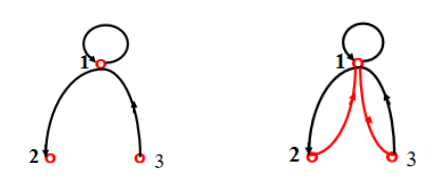
自反闭包是指包含原关系且满足自反性的最小关系。对于既不是[自反](https://baike.baidu.com/item/%E8%87%AA%E5%8F%8D/7468574?fromModule=lemma_inlink" \t "https://baike.baidu.com/item/%E8%87%AA%E5%8F%8D%E9%97%AD%E5%8C%85/_blank)也不是反自反的关系，若要构造自反闭包的话，需要适当的添加一些[序偶](https://baike.baidu.com/item/%E5%BA%8F%E5%81%B6?fromModule=lemma_inlink" \t "https://baike.baidu.com/item/%E8%87%AA%E5%8F%8D%E9%97%AD%E5%8C%85/_blank)使之变成自反关系，同时要求添加的序偶尽可能的少。使用关系图法求解自反闭包，可直接在每个节点上加环，这样得到的关系必定包含恒等关系，可证得是自反的。



而使用关系矩阵法求解，其核心思路与关系图思路类似。对于一个给定的关系矩阵，我们只要将矩阵主对角线上的元素全部置1便可得到原关系的自反闭包，具体可通过一次遍历实现。

**2.2 对称闭包的概念与求解：**

对称闭包是指包含原关系且满足自反性的最小关系，即希望在原关系上添加尽可能少的二元组，使得原关系具有对称性。使用关系图法求解对称闭包，可使用加边的方式，遍历图中所有的边，若某两个节点之间仅存在一条单向边，则添加另一条反方向的边，使其双向可达，反复操作。可证明最后得到的关系是对称的。

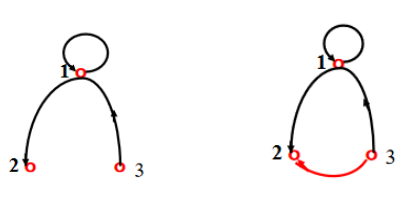


而如若使用关系矩阵求解，则应该遍历矩阵进行修改，使其成为一个对称阵。若遍历过程中发现关系矩阵第i行第j列的逻辑值为1，则修改关系矩阵第j行第i列的值也为1，这样操作得到的矩阵具有对称性。

**2.3 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，对于warshall算法的尝试与研究会在实验报告6中详细说明，本实验中选择采用传统方法求解关系矩阵的传递闭包。



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** reflexive();                   //求自反闭包

**void** symmetric();                   //求对称闭包

**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 自反闭包运算的实现：**

自反闭包运算是将原先关系矩阵主对角线元素置1，因此可进行row次循环，对关系矩阵第i行第i列的元素进行赋值操作。此时体现的是将原先二元关系A与恒等关系做并集，得到自反闭包，可以证得此时r满足自反闭包的三个性质，即包含A、满足自反性、包含于任何一个满足上述两条性质的集合。自反闭包运算的实现代码如下：

//求关系矩阵的自反闭包

**void** Relation::reflexive() {

    //初始化自反闭包矩阵

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

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

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

    }

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

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

            reflexive[i][j] = relation[i][j];

        }

    }

    //求自反闭包

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

        reflexive[i][i] = 1;

    }

    //打印自反闭包

    printMatrix(reflexive);

}

**4.3对称闭包运算的实现：**

对称闭包运算是将原关系矩阵变成对称阵，且原先为1的元素保持不变。为进行更新操作，程序需要进行一个循环次数为row和cal的二重循环，当relation[i][j]值为1时，将对应位置relation[j][i]的元素数值也置为1。此时体现的是将原先二元关系A与A的转置矩阵做并集，得到对称闭包，可以证得此时s满足对称闭包的三个性质，即包含A、满足对称性、包含于任何一个满足上述两条性质的集合。对称闭包运算的实现代码如下：

//求关系矩阵的对称闭包

**void** Relation::symmetric() {

    //初始化对称闭包矩阵

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

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

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

    }

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

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

            symmetric[i][j] = relation[i][j];

        }

    }

    //求对称闭包

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

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

**if** (j < row && symmetric[i][j] == 1)

                symmetric[j][i] = 1;

        }

    }

    //打印对称闭包

    printMatrix(symmetric);

}

**4.4传递闭包运算的实现：**

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。此时体现的是将原先二元关系A与做并集，得到传递闭包，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(传统算法)

**for** (**int** m = 0; m < row - 1; m++) {

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

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

**if** (transitive[i][j] == 1) {

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

**if** (transitive[j][k] == 1)

                            transitive[i][k] = 1;

                    }

                }

            }

        }

    }

    //打印传递闭包

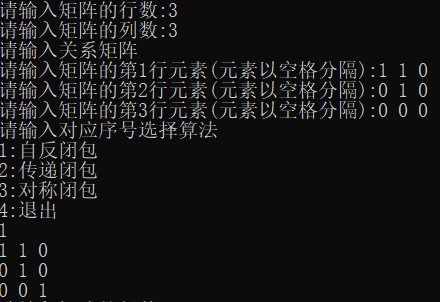
    printMatrix(transitive);

}

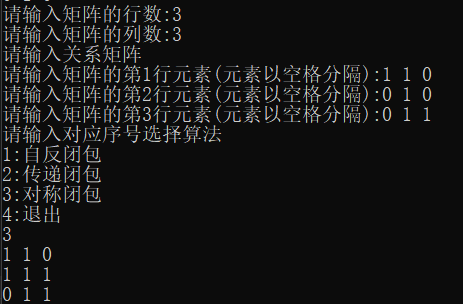
1. **实验测试**

**5.1 常规测试：**

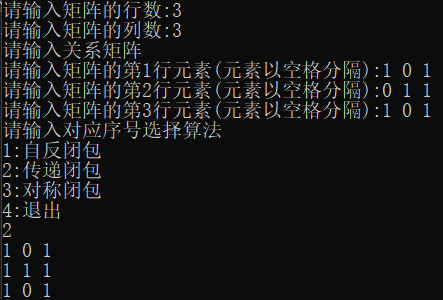
求自反闭包：



求对称闭包：

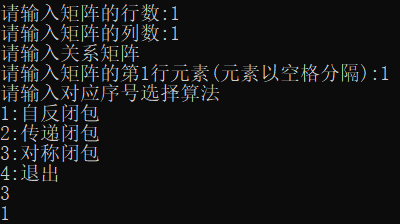


求传递闭包：



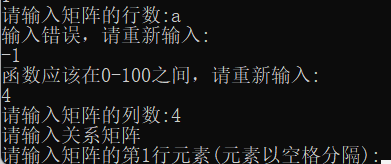
**5.2 边界测试：**

一阶矩阵的情况：

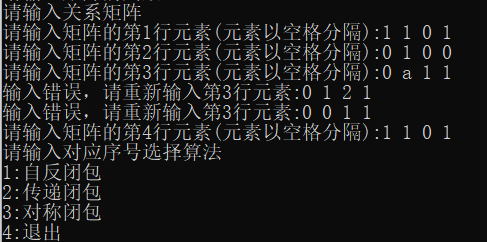


**5.3 错误测试：**

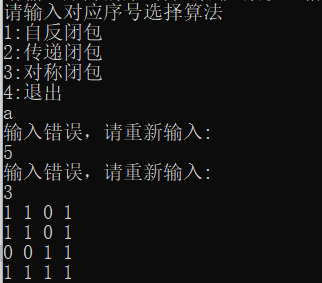
输入行列数错误：



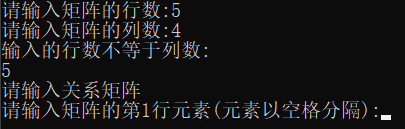
输入矩阵元素错误：



输入操作码错误：



输入行数不等于列数：



1. **心得体会**

本实验中，我们学会了如何利用关系[矩阵](https://so.csdn.net/so/search?q=%E7%9F%A9%E9%98%B5&spm=1001.2101.3001.7020" \t "https://blog.csdn.net/bigotry__/article/details/_blank)求解有限集上给定关系的自反、对称和传递闭包，熟悉关系的闭包运算。其中包括三种闭包运算在集合层面上的运算、在关系图层面上的运算、在关系矩阵层面上的运算。计算机能够直接操作的是关系矩阵，因此本程序设计了一个类，该类的数据成员包含关系矩阵，程序通过调用该类的成员函数执行对关系矩阵的各项操作。通过本实验，我不仅回顾了离散数学课堂上二元关系的相关知识，也对计算机矩阵运算有了更加深入的了解。

在实验6中，我们将会继续探讨关系闭包运算的相关问题，但会侧重与矩阵传递闭包运算的第二种方法Warshall算法，该算法通过巧妙的设计使得整个程序的时间复杂度由降到了，具有非常重要的意义。