项目说明文档

离散数学课程项目

——求关系的自反、对称和传递闭包

作 者 姓 名： 陈奕名

学 号： 2351883

指 导 教 师： 李冰

学院、 专业： 软件学院 软件工程

同济大学

Tongji University

# 1 项目分析

## 1.1 项目要求

手动输入矩阵阶数，然后手动输入关系矩阵，程序需要能求解该关系的自反、对称和传递闭包。

# 2 项目设计

## 2.1 数据结构设计

本程序的核心数据结构是矩阵，用于存储关系矩阵。由于矩阵的大小和元素在运行时是动态变化的，我们选择使用 C++ 标准库中的 std::vector 来动态管理矩阵。std::vector 是一个可变大小的容器，可以方便地扩展和缩小其大小。

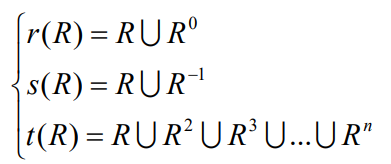
矩阵类型：二维整数矩阵，表示关系矩阵。

数据类型：int，表示矩阵中的每个元素是一个整数（可以是0或1，表示关系的存在与否）。

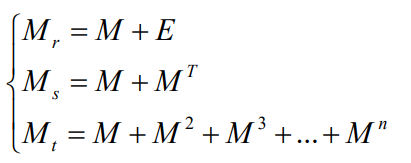
存储方式：std::vector<std::vector<int>>，即外层 vector 用于存储行，内层 vector 用于存储列。

## 2.2 算法设计

由各类闭包的性质有：



转换成关系矩阵，求解方法如下：



# 3 代码实现

#include <iostream>

#include <vector>

using namespace std;

class MatrixOperations {

private:

vector<vector<int>> matrix; // 矩阵

int n, d, z; // n为行数，d为列数，z为用户选择的操作序号

// 输出矩阵

void output(const vector<vector<int>>& mat) {

std::cout << "所求关系矩阵为:"<<endl;

for (int i = 0; i < n; ++i) {

for (int j = 0; j < d; ++j) {

std::cout << mat[i][j] << " "; // 输出每个元素

}

cout << endl;

}

}

// 自反闭包

void zifan() {

for (int i = 0; i < n; ++i) {

matrix[i][i] = 1; // 对角线元素置为1

}

output(matrix); // 输出结果

select(); // 重新选择操作

}

// 对称闭包

void duichen() {

// 创建一个新的矩阵 s1 用来存储转置后的矩阵

vector<vector<int>> s1(n, vector<int>(d));

// 矩阵转置

for (int i = 0; i < n; ++i) {

for (int j = 0; j < d; ++j) {

s1[j][i] = matrix[i][j];

}

}

// 对称闭包运算：矩阵和转置矩阵的元素相加，结果大于1则置为1

for (int i = 0; i < n; ++i) {

for (int j = 0; j < d; ++j) {

matrix[i][j] = matrix[i][j] + s1[i][j];

if (matrix[i][j] > 1) {

matrix[i][j] = 1;

}

}

}

output(matrix); // 输出结果

select(); // 重新选择操作

}

// 传递闭包

void chuandi() {

// 创建矩阵m，a和t来进行操作

vector<vector<int>> m(n, vector<int>(d));

vector<vector<int>> a(n, vector<int>(d));

vector<vector<int>> t(n, vector<int>(d));

// 初始化矩阵m、a和t

for (int i = 0; i < n; ++i) {

for (int j = 0; j < d; ++j) {

a[i][j] = 0;

t[i][j] = matrix[i][j];

m[i][j] = matrix[i][j];

}

}

// 进行传递闭包操作

for (int h = 0; h < n; ++h) {

// 遍历每个元素，进行传递计算

for (int i = 0; i < n; ++i) {

for (int j = 0; j < d; ++j) {

if (m[i][j] == 1) {

for (int k = 0; k < n; ++k) {

if (matrix[j][k] == 1) {

a[i][k] = 1;

}

}

}

}

}

// 更新矩阵 m 和 t

for (int i = 0; i < n; ++i) {

for (int j = 0; j < d; ++j) {

m[i][j] = a[i][j];

t[i][j] += a[i][j];

a[i][j] = 0;

if (t[i][j] > 1) {

t[i][j] = 1;

}

}

}

}

output(t); // 输出结果

select(); // 重新选择操作

}

// 选择操作

void select() {

cout << "请输入矩阵的行数: ";

cin >> n;

cout << "请输入矩阵的列数: ";

cin >> d;

matrix.resize(n, vector<int>(d)); // 调整矩阵大小

cout << "请输入关系矩阵:\n";

for (int i = 0; i < n; ++i) {

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

for (int j = 0; j < d; ++j) {

cin >> matrix[i][j]; // 输入矩阵元素

}

}

// 提供选择算法

cout << "输入对应序号选择算法\n1: 自反闭包\n2: 传递闭包\n3: 对称闭包\n4: 退出\n";

cin >> z;

switch (z) {

case 1: zifan(); break; // 自反闭包

case 2: chuandi(); break; // 传递闭包

case 3: duichen(); break; // 对称闭包

case 4: exit(0); break; // 退出

default:

cout << "无效的选择，请重新输入。"<<endl;

select();

break; // 无效输入

}

}

public:

// 构造函数初始化

MatrixOperations() : n(0), d(0), z(0) {}

// 运行程序

void run() {

select();

}

};

int main() {

MatrixOperations m; // 创建 MatrixOperations 对象

m.run(); // 启动程序

return 0;

}

# 4 测试

## 4.1 自反闭包

**测试用例**：

3 3

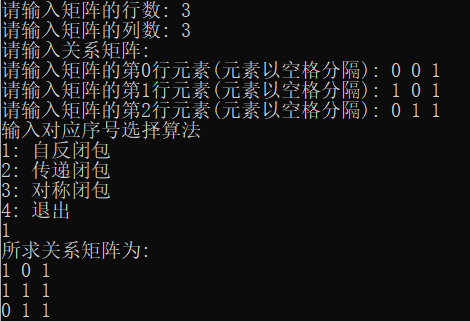
0 0 1

1 0 1

0 1 1

1

**实验结果:**



## 4.2 传递闭包

**测试用例**：

3 3

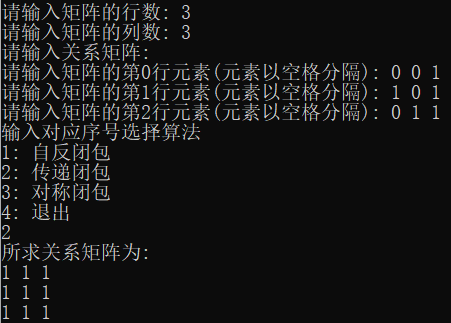
0 0 1

1 0 1

0 1 1

2

**实验结果:**



## 4.3 对称闭包

**测试用例**：

3 3

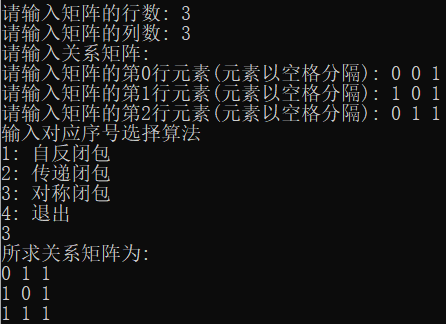
0 0 1

1 0 1

0 1 1

3

**实验结果:**



# 5 心得体会

这段程序实现了关系矩阵的自反闭包、对称闭包和传递闭包操作。在编写过程中，我深入理解了矩阵操作的数学原理，并学习了如何将这些操作转化为程序实现。通过使用 std::vector 动态管理矩阵，我避免了手动管理内存的问题，提升了代码的灵活性和可读性。

程序采用了面向对象的设计，所有矩阵相关的操作都封装在 MatrixOperations 类中。这样不仅使得代码结构更加清晰，也提高了程序的可扩展性。例如，若要增加新的矩阵操作，只需在类中添加相应的函数，而无需大幅修改现有代码。

通过实现传递闭包，我意识到算法的时间复杂度对于程序性能的重要性。虽然程序能够处理小规模的矩阵，但传递闭包的 O(n³) 时间复杂度在处理大矩阵时可能成为瓶颈。未来可以考虑优化算法以提高效率。

总体而言，这个项目增强了我对算法设计、面向对象编程和 C++ 编程技巧的理解，尤其是如何在实际问题中灵活运用这些技术来解决问题。

通过这次实践，我不仅增强了对图论和算法的理论知识，还提高了自己处理复杂数据结构和算法的能力。编写代码的过程中，我更加注重了算法的时间复杂度和空间复杂度，力求在保证正确性的前提下，优化代码的执行效率。

总的来说，这次实现不仅锻炼了我的编程技巧，尤其是数据结构和算法的实现能力，也让我在解决实际问题时更加注重算法的选择和优化，进一步提升了我的问题分析和解决能力。