《离散数学》课程实验报告文档

题目：求关系的自反、对称和传递闭包

姓名：赵卓冰

学号：2252750

专业：软件工程

年级：2023 级

指导教师：唐剑锋

2024 年 11 月 9 日

1. 实验简介

本实验主要计算图论中的三种闭包：自反闭包、传递闭包和对称闭包。通过矩阵的运算（加法、乘法及转置等），实现了对输入的关系矩阵进行闭包运算。程序通过用户输入矩阵数据并选择相应操作，能够展示关系矩阵的各种闭包形式。

2. 实验目的

本实验的主要目的是：

1. 理解和掌握关系矩阵的基本操作（如加法、乘法、转置等）。
2. 熟悉自反闭包、传递闭包和对称闭包的计算方法。

3. 运行环境

本项目可以在不同的开发环境和编译运行环境上运行。

- **Windows 操作系统：**
  - 版本：Windows 10 x64
  - IDE: Visual Studio 2022 (Debug模式)
  - 编译器：MSVC 14.39.33519
- **Linux 操作系统：**
  - 版本：Ubuntu 20.04.6 LTS
  - IDE：VS Code
  - 编译器：gcc version 9.4.0 (Ubuntu 9.4.0-1ubuntu1~20.04.2)

4. 实验原理

4.1. 关系矩阵（邻接矩阵）

在图论中，关系矩阵用于表示图的结构。一个大小为 $n \times n$ 的矩阵用于表示一个有 n 个节点的图，其中矩阵的元素表示节点之间的关系或连接。常见的关系包括：

- **自反关系**：每个元素和它自己之间都有关系。例如，对于每个节点 i，矩阵中的 $M[i][i] = 1$。
- **传递关系**：如果存在 $M[i][j] = 1$ 且 $M[j][k] = 1$，则 $M[i][k]$ 也应为 1。
- **对称关系**：如果 $M[i][j] = 1$，则 $M[j][i] = 1$。

4.2. 闭包运算

- **自反闭包 (Reflexive Closure)**：
  自反闭包是指在原关系矩阵的基础上，确保所有节点与自己之间有关系，即确保所有主对角线元素为 1。
  - **计算方法**：将关系矩阵与单位矩阵（主对角线为1，其他为0）相加。
- **传递闭包 (Transitive Closure)**：
  传递闭包是指如果节点 i 与节点 j 有路径，且节点 j 与节点 k 有路径，则节点 i 与节点 k 也应有路径。
  - **计算方法**：通过反复对关系矩阵进行乘法运算并累加，直至收敛。即计算 $R + R^2 + R^3 + \ldots + R^n$
- **对称闭包 (Symmetric Closure)**：
  对称闭包是指在原关系矩阵的基础上，保证矩阵是对称的，即对于任意 i 和 j，如果 $M[i][j] = 1$，则 $M[j][i] = 1$。
  - **计算方法**：将关系矩阵与其转置矩阵相加。

4.3. 矩阵操作的实现

- **矩阵加法**：使用逻辑“或”运算符实现矩阵加法，表示两个关系的并集。
- **矩阵乘法**：使用逻辑“与”运算符进行矩阵的乘法运算，并在结果中使用逻辑“或”进行累加，表示关系的连接。
- **矩阵转置**：通过交换矩阵的行和列来得到转置矩阵。

5. 实验结果

在程序运行过程中，用户首先需要输入矩阵的大小以及关系矩阵的数据。根据选择的操作（自反闭包、传递闭包、对称闭包），程序会进行相应的闭包运算，并输出结果矩阵。

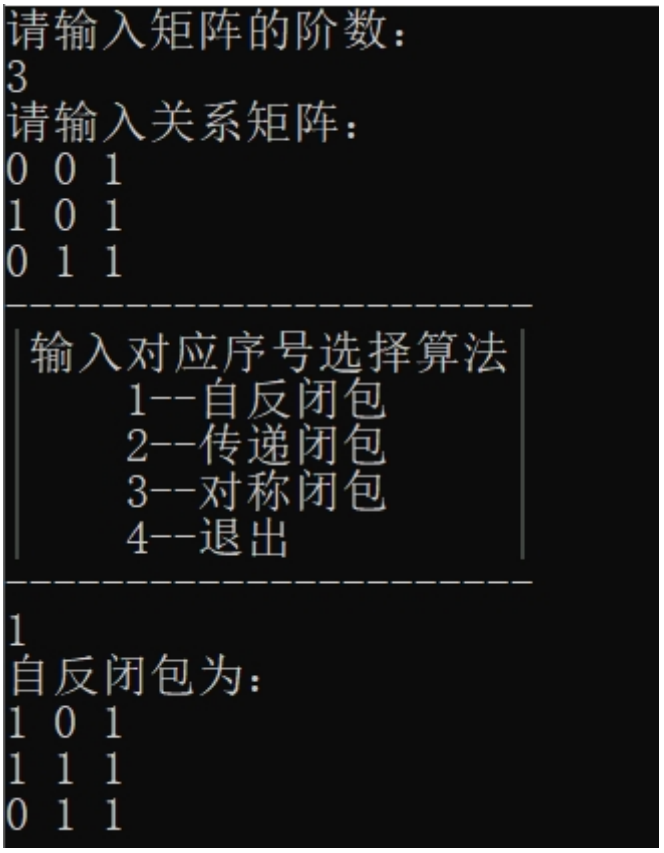
例如，假设输入一个 3×3 的关系矩阵：

```
0 0 1
1 0 1
0 1 1
```

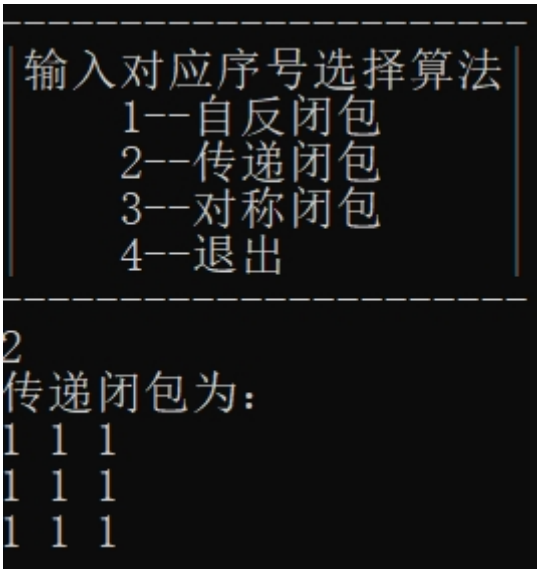
通过执行自反闭包、传递闭包和对称闭包操作，程序分别输出自反闭包、传递闭包和对称闭包的结果矩阵。

5.1. Window系统实验结果

自反闭包：

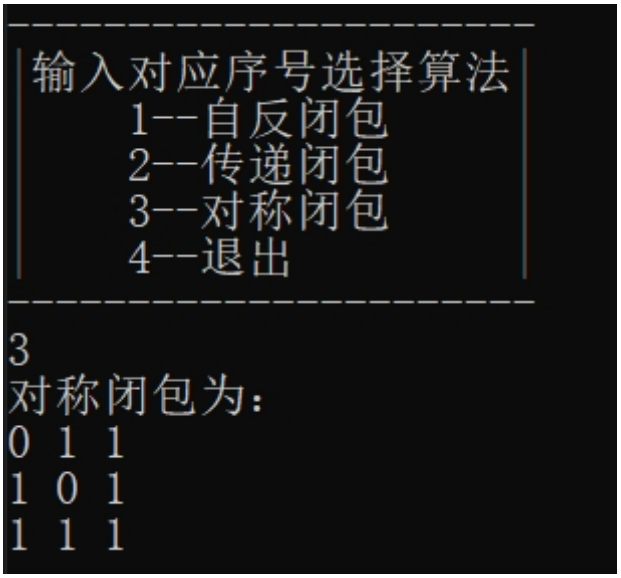


传递闭包：



（传递闭包保证了任何有路径的节点对都会互通）

对称闭包：

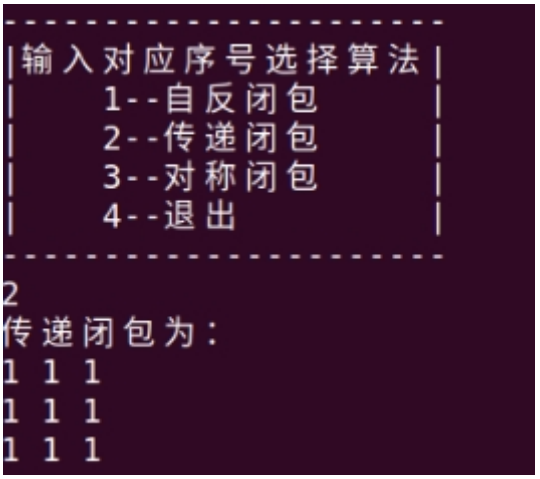


（对称闭包保证了关系的对称性，即 $M[i][j] = M[j][i]$

5.2. linux系统实验结果

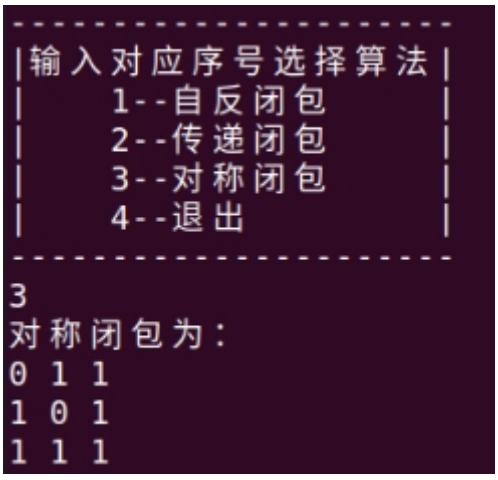
自反闭包：

传递闭包：



（传递闭包保证了任何有路径的节点对都会互通）

对称闭包：



（对称闭包保证了关系的对称性，即 $M[i][j] = M[j][i]$

6. 实验代码

```cpp
#include <iostream>
#include <vector>
using namespace std;

// 关系矩阵类
class Matrix {
private:
    vector<vector<int>> data; // 存储矩阵数据的二维vector
    size_t size;              // 矩阵的大小（行数和列数）
public:
    // 构造函数：初始化大小为input_size的矩阵，并将所有元素设置为0
    Matrix(const size_t input_size) {
        size = input_size;
        // 给矩阵分配内存，并将所有元素初始化为0
        data.assign(size, vector<int>(size, 0));
    }

    // 用另一个Matrix对象来初始化
    Matrix(const Matrix& other) {
        size = other.size;
```

```
        data = other.data;
    }

    // 析构函数：因为使用的是vector容器，内存会自动释放，什么也不做
    ~Matrix() {}

    // 返回矩阵的大小
    size_t Size() {
        return size;
    }

    // 重载[]运算符，用于返回第i行的引用
    vector<int>& operator[](const size_t i) {
        if (i > size - 1) {
            throw::out_of_range("Matrix index out of range"); // 超出范围异常
        }
        return data[i];
    }

    // const引用，适用于const对象
    const vector<int>& operator[](const size_t i) const {
        return data[i];
    }

    // 矩阵的转置
    Matrix Trans() {
        Matrix result(size); // 创建一个与原矩阵同样大小的矩阵
        for (int i = 0; i < size; ++i) {
            for (int j = 0; j < size; ++j) {
                result[i][j] = data[j][i]; // 将原矩阵的(i, j)元素放置到转置矩阵的(j, i)位置
            }
        }
        return result;
    }

    // 重载+运算符，表示矩阵的逻辑加法（并集）
    Matrix operator+(const Matrix& other) {
        if (other.size != size) {
            throw::invalid_argument("矩阵的维度不同"); // 矩阵维度不同，不能进行加法
        }
        Matrix result(size); // 结果矩阵
        for (size_t i = 0; i < size; ++i) {
            for (size_t j = 0; j < size; ++j) {
                result[i][j] = data[i][j] || other[i][j]; // 逻辑“或”操作（并集）
            }
        }
        return result;
    }
```

```
69
70      // 重载*运算符，表示矩阵的乘法（逻辑与运算）
71      Matrix operator*(const Matrix& other) {
72          if (other.size != size) {
73              throw::invalid_argument("矩阵的维度不同"); // 矩阵维度不同，
    不能进行乘法
74          }
75          Matrix result(size); // 结果矩阵
76          for (size_t i = 0; i < size; ++i) {
77              for (size_t j = 0; j < size; ++j) {
78                  for (size_t k = 0; k < size; ++k) {
79                      result[i][j] = result[i][j] || (data[i][k] &&
    other[k][j]); // 逻辑与操作，然后逻辑“或”累加
80                  }
81              }
82          }
83          return result;
84      }
85
86      // 输入矩阵数据
87      void Input() {
88          for (int i = 0; i < size; ++i) {
89              for (int j = 0; j < size; ++j) {
90                  cin >> data[i][j]; // 从输入读取数据
91              }
92          }
93      }
94
95      // 打印矩阵
96      void Disp() {
97          for (int i = 0; i < size; ++i) {
98              for (int j = 0; j < size; ++j) {
99                  cout << data[i][j] << ' '; // 输出矩阵元素
100             }
101             cout << endl;
102         }
103     }
104 };
105
106 // 求自反闭包
107 Matrix ReflexiveClosure(Matrix& matrix) {
108     // 创建一个单位矩阵
109     Matrix identity_matrix(matrix.Size());
110     // 主对角线元素全部设为1，表示自反关系
111     for (int i = 0; i < matrix.Size(); ++i) {
112         identity_matrix[i][i] = 1;
113     }
114
115     // 自反闭包=单位矩阵+关系矩阵
116     return identity_matrix + matrix;
117 }
118
```

```cpp
// 求传递闭包
Matrix TransitiveClosure(Matrix& matrix) {
    // 创建结果矩阵，初始化为关系矩阵
    Matrix result(matrix);
    // 通过迭代计算，累积R + R^2 + ... + R^n，直到收敛
    for (int i = 0; i < matrix.Size() - 1; ++i) {
        result = result * matrix + matrix; // 每次更新矩阵的值
    }
    return result;
}

// 求对称闭包
Matrix SymmetricClosure(Matrix& matrix) {
    // 创建结果矩阵，初始化为关系矩阵
    Matrix result(matrix);
    // 对称闭包=关系矩阵+关系矩阵的转置
    result = result + matrix.Trans();
    return result;
}

int main() {
    size_t size;
    cout << "请输入矩阵的阶数：" << endl;
    cin >> size; // 输入矩阵的阶数（大小）
    cout << "请输入关系矩阵：" << endl;
    Matrix matrix(size); // 创建矩阵对象
    matrix.Input(); // 输入矩阵数据

    // 提供菜单供用户选择
    while (1) {
        cout << "----------------------" << endl;
        cout << "|输入对应序号选择算法|" << endl;
        cout << "|    1--自反闭包      |" << endl;
        cout << "|    2--传递闭包      |" << endl;
        cout << "|    3--对称闭包      |" << endl;
        cout << "|    4--退出          |" << endl;
        cout << "----------------------" << endl;
        int choice;
        cin >> choice; // 用户输入选择的操作

        // 清除输入错误的状态
        if (cin.fail()) {
            cin.clear();
            cin.ignore(65536, '\n');
        }

        // 根据用户的选择执行不同的操作
        if (1 == choice) {
            // 计算自反闭包
            Matrix reflexive_closure = ReflexiveClosure(matrix);
            cout << "自反闭包为：" << endl;
            reflexive_closure.Disp(); // 输出自反闭包
```

```
        }
        else if (2 == choice) {
            // 计算传递闭包
            Matrix transitive_closure = TransitiveClosure(matrix);
            cout << "传递闭包为：" << endl;
            transitive_closure.Disp(); // 输出传递闭包
        }
        else if (3 == choice) {
            // 计算对称闭包
            Matrix symmetric_closure = SymmetricClosure(matrix);
            cout << "对称闭包为：" << endl;
            symmetric_closure.Disp(); // 输出对称闭包
        }
        if (4 == choice) {
            // 用户选择退出
            break;
        }
    }
    return 0;
}

```

7. 实验总结与心得体会

本实验通过实现关系矩阵的闭包运算，深入理解了矩阵运算在图论中的应用，并加深了对矩阵操作的掌握。通过编写和测试自反闭包、传递闭包和对称闭包的计算方法，体会到了闭包运算在图中连接和关系传播中的重要作用。

在实现过程中，重载运算符（如 `+`, `*`, `[]`）为代码的简洁性和可读性提供了极大的帮助。通过自定义类 `Matrix`，能够方便地实现矩阵操作，避免了直接使用二维数组带来的复杂性。

心得体会：

1. **矩阵的抽象和封装**：通过面向对象的设计思想，将矩阵和相关操作封装成一个类，使得矩阵操作更加简洁和易于管理。
2. **图论的应用**：通过实现自反闭包、传递闭包和对称闭包，进一步理解了图中节点之间关系的传递性和对称性。
3. **错误处理**：程序中加入了合理的错误处理机制（如越界检查和输入验证），提高了程序的健壮性。
4. **矩阵运算的效率**：通过反复运算矩阵的乘法和加法，实验加深了对矩阵运算性能的关注，尤其是大型矩阵运算时可能带来的性能瓶颈。