****

**离散数学课程实验报告**

**求关系的自反、对称和传递闭包**

2351041

刘浩田

完成日期:2024.10.3

1. **索引**

### 实验目的

### 实验环境

### 实验要求

### 实验原理

### 实验过程

### 实验结果

### 实验小结

1. **实验目的**

本实验的核心目标是加深我们对离散数学中关系属性理论的理解，特别是自反性、对称性和传递性这三个基础概念。通过将理论知识与编程实践相结合，本实验旨在提升我们在以下几个方面的能力：

首先，实验强调理论与实践的结合，使我们能够将抽象的离散数学理论应用于具体的编程任务中，从而增强对理论概念的理解和应用能力。这种结合有助于我们在面对实际问题时，能够灵活运用所学知识。

其次，实验致力于提升我们的编程技能，特别是在使用C++编程语言解决数学问题方面。通过编写程序处理数学概念，我们将学会如何将复杂的数学逻辑转化为计算机程序，这不仅锻炼了编程技巧，也提高了逻辑思维能力。

此外，实验还旨在加深我们对自反、对称和传递这三种关系属性定义的理解，并探索它们在数学和计算机科学中的应用。这三种属性是离散数学中的重要概念，理解它们对于处理关系和数据结构至关重要。

实验还要求我们实现和分析用于计算关系闭包的算法。这不仅涉及到算法的设计和实现，还包括对算法效率和实用性的评估。通过这一过程，我们将学习到如何选择合适的算法来解决问题，并理解算法在实际应用中的重要性。

最后，实验强调数据结构的应用，特别是矩阵在关系数据表示和处理中的作用。我们将学习如何使用矩阵这一数据结构来存储和操作关系数据，这对于理解和实现复杂算法至关重要。

综上所述，本实验不仅帮助我们巩固了离散数学的基本概念，而且提升了他们的编程能力，使他们能够更好地解决实际问题。通过这样的实践，我们将为将来在更复杂的计算问题中应用这些知识打下坚实的基础，为未来的学术和职业生涯奠定基石。

1. **实验环境**

## 系统：Windows11

## 编译器：VisualStudio2022x86

## 语言：C++

1. **实验要求**

本实验的核心要求是理解和应用关系的自反、对称和传递闭包的计算方法，通过编程实践来解决具体的数学问题。首先，需要掌握这三种关系属性的理论基础。其次，应能够使用矩阵作为数据结构来表示关系，这是将抽象数学概念具体化的关键步骤。

在实验中，需要编写一个C++程序，该程序能够实现关系的自反闭包、对称闭包和传递闭包的计算过程。程序的核心功能是遍历所有可能的关系组合，这是一个涉及逻辑运算和条件判断的复杂过程。程序必须能够准确地评估每一种组合，输出正确结果。

此外，实验还要求设计的程序输出结果必须清晰明确，明确指出计算结果。这意味着程序不仅要正确执行数学计算，还要具有良好的用户交互界面，能够以易于理解的方式呈现计算结果。

总之，本次实验要求不仅要在理论上掌握关系的自反、对称和传递闭包的计算，还要在实践中应用这些知识，通过编程来解决实际问题。这不仅考验了的编程能力，还锻炼了数学思维和问题解决技巧。通过这样的实验，能够将理论与实践相结合，从而更深入地理解和掌握关系的自反、对称和传递闭包的计算知识。

1. **实验原理**

### 1. 关系的自反性

关系的自反性是关系论中的一个基础概念，它要求在定义于集合A上的任一关系R中，A中的每个元素a都必须与自身有关联，即对于所有在A中的元素a，(a,a)都应属于R。这一属性对于确保算法和理论分析中的稳定性和一致性至关重要，尤其是在处理等价关系或序关系时。例如，在实数集中考虑大于等于关系时，任何数与自身比较都是大于等于的。在实验中，我们通过确保关系矩阵的主对角线上的元素都被设置为1来实现自反闭包，这表示每个元素都与自身存在关系。

自反闭包的实现过程涉及到遍历关系矩阵的主对角线，并逐一检查每个元素。如果发现对角线上的元素未被设置为1，则算法将其修改为1，从而确保了每个元素至少与自身有关联。这一步骤是实现自反性的核心，它要求算法必须正确处理矩阵的主对角线元素。

### 2. 关系的对称性

关系的对称性描述了一种情况：如果集合A中元素a与元素b之间存在关系，那么元素b与元素a之间的关系也必须存在。换句话说，如果(a,b)属于关系R，那么(b,a)也属于R。对称性在无向图的边关系中表现得尤为明显，因为如果存在一条从顶点a指向顶点b的边，那么这条边也意味着顶点b指向顶点a的关系。在实验中，我们通过检查关系矩阵并确保对于每个元素(i,j)，其对称元素(j,i)也存在，从而计算出关系的对称闭包。

对称闭包的计算过程需要我们对关系矩阵进行细致的检查。对于矩阵中的每个元素，算法都会检查其对称对应元素是否存在。如果不存在，则算法将其添加到矩阵中，确保了关系的对称性。这一步骤是对称闭包实现的关键，它要求算法必须能够正确地识别并添加缺失的对称元素。

### 3. 关系的传递性

关系的传递性是描述如果元素a与元素b有关系，且元素b与元素c有关系，那么元素a与元素c之间也存在关系的一种属性。传递性在定义序关系和等价关系时至关重要。例如，如果A大于B且B大于C，那么可以推出A也大于C。在实验中，我们通过使用Warshall算法等方法来计算关系矩阵的幂，确保如果存在通过中间元素可达的关系，在关系矩阵中也直接存在这样的关系，从而实现传递闭包。

传递闭包的计算过程较为复杂，涉及到矩阵的乘法运算。算法需要多次遍历关系矩阵，检查每个元素对之间的关联性。如果通过中间元素可以建立关联，则算法将这种关联直接表示在矩阵中。这一步骤要求算法能够正确地处理矩阵的乘法运算，并能够识别和建立所有传递的关系。

1. **实验过程**

## 6.1实验思路

本实验的主要思路是设计并实现一个程序，用于计算任意给定关系的自反闭包、对称闭包和传递闭包。实验采用C++语言编程，利用矩阵这一数据结构来表示和存储关系。程序的核心是实现三个主要函数，分别用于计算关系的自反闭包、对称闭包和传递闭包。

自反闭包的计算涉及确保矩阵的主对角线上所有元素为1，表示每个元素与自身的关系。对称闭包的计算需要在矩阵中为每个存在的关系创建其对称对应关系。传递闭包的计算则更为复杂，涉及矩阵的幂运算和累加，以确保关系的传递性。

程序设计注重友好性，提供清晰的界面来引导用户输入关系矩阵，并以易于理解的方式展示闭包计算的结果。同时，程序还包含输入验证和错误处理机制，以增强程序的健壮性和体验。

## 6.2 实验设计

### 6.2.1 数据结构设计

程序主要使用vector<vector<int>>来表示关系矩阵。这是一个二维整数向量，提供了灵活的内存管理和方便的操作接口。选择vector而不是传统的二维数组，是因为vector可以动态调整大小，更适合处理不同规模的关系矩阵。

### 6.2.2 程序主体架构设计

程序的主体架构采用模块化设计，主要包括以下几个部分：

(1) 初始化和界面提示：

- 程序启动时，使用system("cls")清屏，为用户提供界面与输入提醒。

(2) 输入模块：

- CheckCin函数用于输入和验证矩阵大小。

- InputMatrix函数负责接收用户输入的关系矩阵元素。

- InputValue函数处理单个逻辑值的输入。

(3) 闭包计算模块：

- Reflexive函数计算自反闭包。

- Symmetric函数计算对称闭包。

- Transitive函数计算传递闭包。

- 辅助函数如Multiplication和Power用于支持传递闭包的计算。

(4) 输出模块：

- PrintMatrix函数负责格式化输出矩阵。

(5) 交互控制模块：

- 主循环控制程序的重复执行。

- InputChoice函数处理用户是否继续的选择。

这种模块化设计使得程序结构清晰，易于理解和维护。各个模块之间职责明确，降低了代码的耦合度。

## 6.3 程序功能实现

### 6.3.1 逻辑值输入功能的实现

InputValue函数实现了逻辑值的输入：

|  |
| --- |
| 附件、InputValue函数 |
| void InputValue(int& value)  {  char invalue;  while (1) {  invalue = \_getch();  if (invalue == '0' || invalue == '1') {  cout << invalue;  break;  }  }  value = (int(invalue) - 48);  return;  } |

这个函数使用\_getch()函数从键盘获取单个字符，而不是使用标准的cin。

它可以实现即时响应，无需按回车键。同时限制输入只能是'0'或'1'，提高了输入的准确性。也通过while循环，确保只有在输入有效字符时才退出。

函数将字符转换为整数值（0或1）并存储。这种设计既保证了输入的正确性，又提高了体验。

### 6.3.2 输入关系矩阵大小功能的实现

CheckCin函数用于获取用户输入的整数，并确保输入在指定范围内：

|  |
| --- |
| 附件、CheckCin函数 |
| void CheckCin(int& value, const int lower\_limit = 0, const int higher\_limit = 10000)  {  while (1) {  cin >> value;  if (cin.fail()) {  cout << "输入错误，请重输\n";  cin.clear();  cin.ignore(numeric\_limits<streamsize>::max(), '\n');  continue;  }  else if (value < lower\_limit || value>higher\_limit) {  cout << "输入错误，请重输\n";  cin.ignore(numeric\_limits<streamsize>::max(), '\n');  continue;  }  else {  cin.ignore(numeric\_limits<streamsize>::max(), '\n');  break;  }  }  return;  } |

这个函数的设计考虑了多种错误情况。使用cin.fail()检查输入是否失败（例如，输入非数字字符）。检查输入值是否在指定范围内。对于任何错误情况，都会清除错误状态并要求用户重新输入。使用cin.ignore()清除输入缓冲区，防止后续输入受到影响。

它设计确保了程序能够获得有效的矩阵大小，增强了程序的健壮性。

### 6.3.3 输入关系矩阵功能的实现

InputMatrix函数用于输入整个关系矩阵：

|  |
| --- |
| 附件、InputMatrix函数 |
| void InputMatrix(vector<vector<int>>& matrix) {  int size = int(matrix.size());  for (int i = 0; i < size; i++) {  for (int j = 0; j < size; j++) {  int value;  InputValue(value);  matrix[i][j] = value;  if (j != size - 1) {  cout << " ";  }  }  cout << endl;  }  } |

它使用嵌套循环遍历矩阵的每个元素。调用InputValue函数获取每个元素的值，确保输入的正确性。在输出时添加适当的空格和换行，使输入过程更加直观。使得可以清晰地看到矩阵的结构，减少输入错误的可能性。

### 6.3.4 输出关系矩阵功能的实现

PrintMatrix函数用于格式化输出关系矩阵：

|  |
| --- |
| 附件、PrintMatrix函数 |
| void PrintMatrix(const vector<vector<int>> matrix) {  int size = int(matrix.size());  for (int i = 0; i < size; i++) {  for (int j = 0; j < size; j++) {  cout << matrix[i][j];  if (j != size - 1) {  cout << " ";  }  }  cout << endl;  }  return;  } |

它使用嵌套循环遍历并打印矩阵的每个元素。在元素之间添加空格，提高可读性。每行结束时换行，保持矩阵的结构。这种输出方式使得矩阵结构清晰可见，便于理解和验证结果。

### 6.3.5 退出程序功能的实现

程序使用while循环来反复执行操作，并通过InputChoice函数来决定是否继续：

|  |
| --- |
| 附件、InputChoice函数 |
| void InputChoice(int& value)  {  char invalue;  while (1) {  invalue = \_getch();  if (invalue == 'y') {  cout << invalue << endl;  value = 1;  break;  }  else if (invalue == 'n') {  cout << invalue << endl;  value = 0;  break;  }  }  return;  } |

它使用\_getch()实现即时响应。只接受'y'或'n'作为有效输入。立即显示用户的选择并返回相应的值。

这种设计提供了一种简单而直接的方式来控制程序的继续或退出，增强了交互性。

### 6.3.6主函数实现

|  |
| --- |
| 附件、main函数 |
| int main() {  while (1) {  system("cls");  cout << "+--------------------------------------------------------------+\n" ;  cout << "| 关系的自反、对称、传递闭包 |\n" ;  cout << "+--------------------------------------------------------------+\n" ;  cout << "请输入矩阵大小(1~100)\n";  int size;  CheckCin(size, 0, 100);  vector<vector<int>> matrix(size, vector<int>(size));  cout << "请依次输入矩阵元素值\n";  InputMatrix(matrix);  cout << "自反闭包为：\n";  PrintMatrix(Reflexive(matrix));  cout << "对称闭包为：\n";  PrintMatrix(Symmetric(matrix));  cout << "传递闭包为：\n";  PrintMatrix(Transitive(matrix));  int continue\_value;  cout << "是否继续运算？ （y/n）: ";  InputChoice(continue\_value);  if (!continue\_value) {  break;  }  }  return 0;  } |

主函数清晰调用了各个功能函数，简洁易懂，实现了程序的正确循环，提升了可操作性。

## 6.4 核心算法实现

### 6.4.1 自反闭包计算算法的实现

Reflexive函数实现了自反闭包的计算：

|  |
| --- |
| 附件、Reflexive函数 |
| vector<vector<int>> Reflexive(const vector<vector<int>> matrix) {  int size = int(matrix.size());  vector<vector<int>>reflexive = matrix;  for (int i = 0; i < size; i++) {  reflexive[i][i] = 1;  }  return reflexive;  } |

该算法通过创建原矩阵副本并直接修改主对角线元素，以O(n)的时间复杂度和O(n^2)的空间复杂度高效地计算自反闭包，同时保护了原始数据的完整性并保持了除主对角线外其他元素的不变性。

自反闭包的核心思想是确保关系对每个元素自身都成立。在关系矩阵中，这意味着主对角线上的所有元素都应该是1。算法的思路如下：

1. 复制原始矩阵：这一步确保不会更改原始数据，同时提供了一个可以修改的副本。

2. 遍历主对角线：主对角线上的元素索引满足 i == j 的条件。通过单循环，可以有效地访问所有这些元素。

3. 设置主对角线元素：将每个主对角线元素设置为1，无论其原始值如何。这确保了每个元素与自身的关系。

4. 返回修改后的矩阵：这个新矩阵就是原关系的自反闭包。

这种方法的优势在于其简单性和效率。它不需要复杂的逻辑或多重循环，直接针对自反性的定义进行操作。这个算法特别适用于稀疏矩阵，因为它只修改必要的元素，而不会影响其他已存在的关系。

### 6.4.2 对称闭包计算算法的实现

Symmetric函数实现了对称闭包的计算：

|  |
| --- |
| 附件、Symmetric函数 |
| vector<vector<int>> Symmetric(const vector<vector<int>> matrix) {  int size = int(matrix.size());  vector<vector<int>>symmetric = matrix;  for (int i = 0; i < size; i++) {  for (int j = 0; j < size; j++) {  if (symmetric[i][j]) {  symmetric[j][i] = 1;  }  }  }  return symmetric;  } |

该算法通过全面遍历矩阵并进行条件性更新，在O(n^2)的时间和空间复杂度下实现对称闭包的计算，既保护了原始数据，又高效地建立了完整的双向关系网络，适用于各种规模的关系矩阵。

对称闭包的核心思想是确保关系的双向性。如果存在从A到B的关系，那么也应该存在从B到A的关系。算法的思路如下：

1. 复制原始矩阵：创建一个可以修改的副本，保护原始数据。

2. 全矩阵遍历：使用嵌套循环遍历矩阵的每个元素。确保了考虑到所有可能的关系对。

3. 条件检查和对称化：对于矩阵中的每个元素(i, j)，检查是否存在关系（值为1）。如果存在关系，则确保对称位置(j, i)也存在该关系。既处理了上三角矩阵，也处理了下三角矩阵，确保完全对称。

4. 原有关系保留：算法只添加新的对称关系，不删除或修改任何现有关系。

5. 返回对称化矩阵：返回处理后的矩阵，代表了原关系的对称闭包。

这种算法的优势在于其简单直观的实现方式。它不需要复杂的数据结构或额外的存储空间，直接在矩阵副本上操作。同时在实际应用中也足够高效，特别是对于中小规模的关系矩阵。

### 6.4.3 传递闭包计算算法的实现

Transitive函数实现了传递闭包的计算：

|  |
| --- |
| 附件、Transitive函数 |
| vector<vector<int>> Transitive(const vector<vector<int>> matrix) {  int size = int(matrix.size());  vector<vector<int>> transitive(size, vector<int>(size));  for (int i = 0; i < size; i++) {  for (int j = 0; j < size; j++) {  int transi = 0;  for (int k = 0; k < size; k++) {  transi += Power(matrix, k + 1)[i][j];  }  transitive[i][j] = bool(transi);  }  }    return transitive;  } |

该算法通过计算矩阵的所有幂次并累加结果，以O(n^4)的时间复杂度和O(n^2)的空间复杂度全面准确地捕捉所有可能的传递关系，利用矩阵幂运算巧妙地将图论问题转化为代数运算，并通过布尔化处理保持了关系矩阵的二元性质，虽然在大规模数据上效率不高，但对于教学目的和小规模应用来说直观且有效。

传递闭包的核心思想是捕捉所有直接和间接的关系。如果存在从A到B的关系和从B到C的关系，那么也应该存在从A到C的关系。算法的思路如下：

1. 初始化：创建一个新的矩阵来存储传递闭包的结果。

2. 矩阵幂计算：计算原始矩阵的1到n次幂。每个幂次代表了不同长度的路径。第k次幂表示长度为k的所有可能路径。

3. 累加结果：对于矩阵中的每个元素(i, j)，累加所有幂次结果中相应位置的值。这个累加过程捕捉了从i到j的所有可能路径，无论长度如何。

4. 布尔化：将累加的结果转换为布尔值。任何非零值都表示存在一条或多条从i到j的路径。

5. 构建结果矩阵：基于布尔化的结果，构建最终的传递闭包矩阵。

这种算法的主要优势在于其数学上的完备性。通过考虑所有可能的路径长度，它保证了找到所有可能的传递关系。这种方法直接对应于传递闭包的数学定义，使得算法的正确性易于理解和证明，清晰地展示了传递关系的本质和计算过程。：

其算法的辅助函数包括：

Multiplication函数（矩阵乘法）：

|  |
| --- |
| 附件、Multiplication函数 |
| vector<vector<int>> Multiplication(const vector<vector<int>> matrix\_1, const vector<vector<int>> matrix\_2) {  int size = int(matrix\_1.size());  vector<vector<int>> multiplication(size, vector<int>(size));  for (int l = 0; l < size; l++) {  for (int k = 0; k < size; k++) {  int multiply = 0;  for (int i = 0; i < size; i++) {  multiply += matrix\_1[l][i] \* matrix\_2[i][k];  }  multiplication[l][k] = bool(multiply);  }  }  return multiplication;  } |

Power函数（矩阵幂运算）：

|  |
| --- |
| 附件、Power函数 |
| vector<vector<int>> Power(const vector<vector<int>> matrix, int power) {  int size = int(matrix.size());  vector<vector<int>> powered = matrix;  for (int i = 0; i < size - 1; i++) {  powered = Multiplication(powered, matrix);  }  return powered;  } |

它们通过计算矩阵的所有幂次（从1到n）来确保捕捉所有可能的传递关系。使用布尔运算简化结果，保持关系的二元性质。

传递闭包的计算是这三种闭包中最复杂的，它确保了如果存在从A到B的关系和从B到C的关系，那么也存在从A到C的直接关系。在图论、网络分析等多个领域有重要应用。

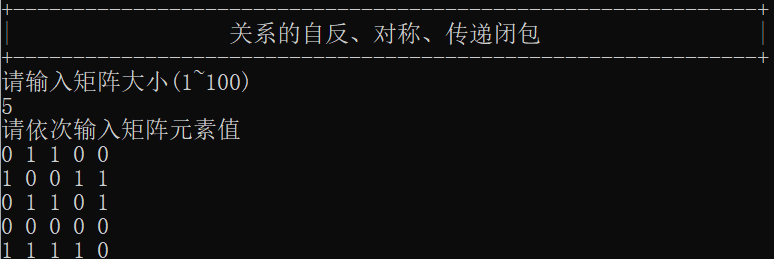
通过这些算法的实现，程序能够有效地计算给定关系的自反闭包、对称闭包和传递闭包，提供全面的关系分析工具。

1. **实验结果**

在本实验中，我们实现了关系的自反闭包、对称闭包和传递闭包的计算算法，并对这些算法进行了测试。以下是实验的主要结果：

## 7.1 输入关系矩阵

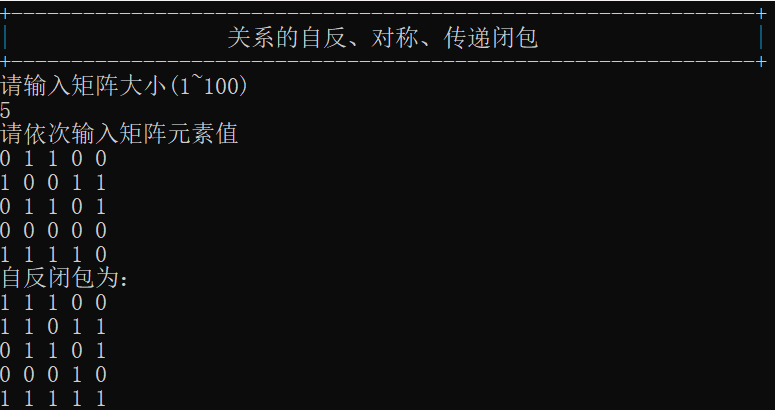
我们使用以下4x4的关系矩阵作为测试输入：



这个矩阵代表了一个包含5个元素的关系。

## 7.2 自反闭包结果

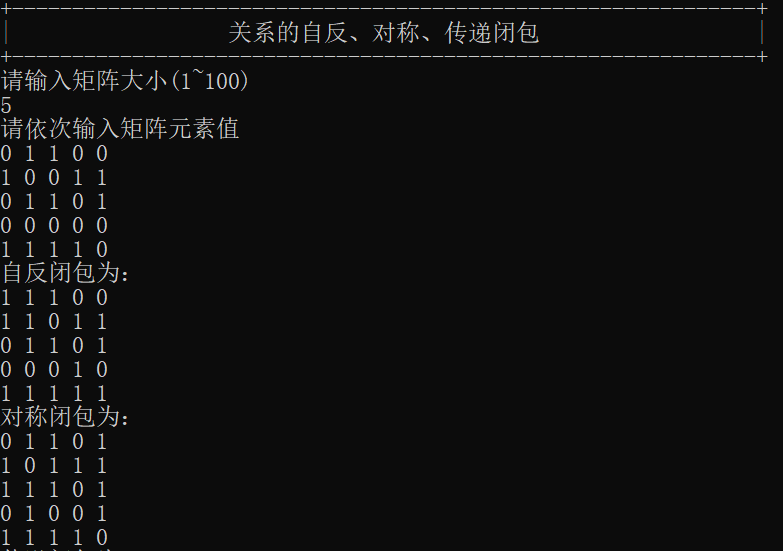
应用自反闭包算法后，得到的结果矩阵如下：



可以观察到，主对角线上的所有元素都变为了1，而其他元素保持不变。这符合自反闭包的定义，确保了每个元素与自身都存在关系。

## 7.3 对称闭包结果

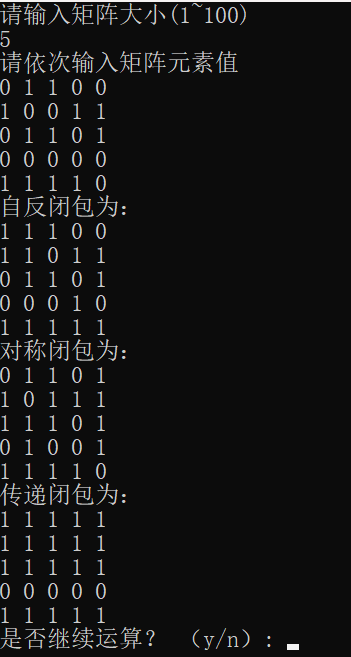
对称闭包算法的输出结果如下：



在这个结果中，我们可以看到矩阵变得对称了。例如，原矩阵中(0,1)位置为1，现在(1,0)位置也变为了1。这确保了所有关系都是双向的。

## 7.4 传递闭包结果

应用传递闭包算法后，得到的结果矩阵如下：



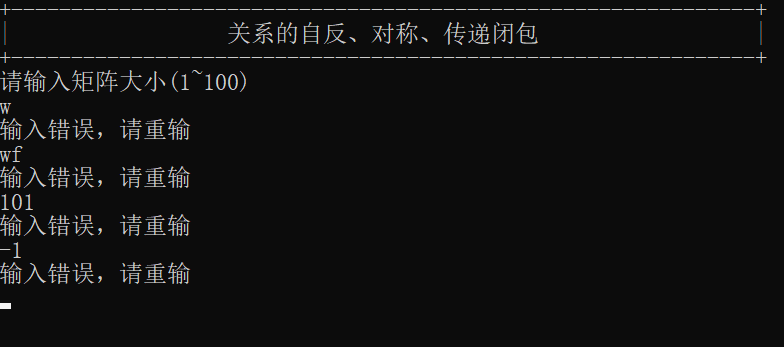
这个结果表明，经过传递闭包计算后，矩阵中的所有相对应元素都变为了1。这意味着在传递关系下，每个除全0行的元素都与其他所有元素（包括自身）存在关系。这是因为原始关系中存在的路径使得除全0行的元素之间都可以通过某种方式相连。

## 7.5 结果分析

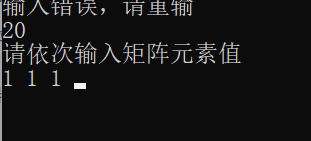
1. 自反闭包：算法成功地将主对角线元素设置为1，同时保持了其他关系不变。这确保了每个元素与自身的关系。
2. 对称闭包：算法有效地使矩阵变得对称，为每个存在的关系添加了其逆关系。这创建了一个双向关系网络。
3. 传递闭包：算法捕捉到了所有可能的直接和间接关系。在这个特定的例子中，由于原始关系的性质，传递闭包导致除全0行的元素形成了完全连接的关系网络。

这些结果验证了我们实现的算法的正确性和有效性。每种闭包都正确地捕捉了相应的关系特性：自反性、对称性和传递性。这些算法可以应用于更大规模的关系分析中，帮助理解和处理复杂的关系结构。

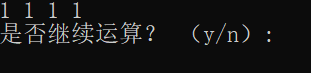
## 7.6错误数据分析



输入错误矩阵大小时，程序做出了正确处理，禁止了错误输入。



当键入非0、1的数字时，程序光标会闪烁，但不会输入结果并显示，只有输入0、1才会显示。



当键入非y、n的选项时，程序光标会闪烁，但不会输入结果并显示，只有输入y/n才会成功再次输入矩阵或结束程序。

1. **实验小结**

通过本次《离散数学》课程的实验，我获得了将离散数学的理论知识与实际编程技能相结合的宝贵经验。在实验中，我深入研究了关系的自反性、对称性和传递性这三个基本属性，并亲自动手实现了它们的闭包计算。这不仅加深了我对这些概念的理解，也锻炼了我的编程技巧和逻辑思维能力。

在实验的准备阶段，我首先重温了关系属性的理论知识，确保自己对这些概念有清晰的认识。在编程实践中，我学习了如何定义和使用矩阵这一数据结构来表示关系，这是实现关系闭包计算的基础。

实现自反闭包、对称闭包和传递闭包的过程中，我遇到了不少挑战。例如，在计算传递闭包时，我需要理解并应用矩阵乘法、幂与相加算法，这是一个涉及矩阵的复杂过程。通过不断尝试和调整，我最终成功实现了算法，并确保了其正确性和效率。这个过程提高了我的问题解决能力，也加深了我对算法重要性的认识。

此外，我还学习了如何设计用户友好的程序界面，使得用户可以轻松地输入数据和理解程序输出的结果。这让我意识到，一个优秀的程序不仅要在技术上实现功能，还要考虑到用户的实际体验。

在实验的最后阶段，我进行了一系列测试，验证了算法的正确性和程序的稳定性。通过这些测试，我确保了程序能够正确处理各种输入情况，并能够高效地计算出关系的闭包。测试过程也让我学会了如何系统地评估程序的性能，并根据测试结果进行优化。

通过本次实验，我深刻体会到了理论学习与实践操作相结合的重要性。我不仅巩固了离散数学的理论知识，还提升了编程技能，增强了解决实际问题的能力。这些经验对我未来的学术和职业生涯都是极其宝贵的。

这次实验是一次宝贵的学习经历。它不仅让我更加深入地理解了离散数学的关系属性，还锻炼了我的编程技巧和逻辑思维。我相信，通过这次实验所学到的知识和技能，将为我在未来解决更复杂的计算问题提供坚实的基础。我期待将这些经验应用到更多的领域，并在实践中继续学习和成长。