**实验报告3:** **求关系的自反、对称和传递闭包**

# 实验内容

求关系的自反、对称和传递闭包。

# 实验环境

采用C＋＋编程语言，MS Visual Studio 2012实验环境实现。

# 核心算法及解题思路

## 解题思路

要求求关系的自反、对称和传递闭包，因此为了要能够求解，该关系只能是某一非空集合上的关系，所以该关系矩阵要是方阵。只需知道矩阵行数或列数（即阶数）即可。

为了方便逻辑加的操作，将相关变量设置为bool类型，将关系矩阵存储于二维数组中，方便求解。①求关系的自反闭包，只要将矩阵的主对角线全部置为1；②求关系的对称闭包，则将关系矩阵加上（逻辑加）其转置矩阵即可得到；③求关系的传递矩阵，则需求得关系矩阵的不同的幂并累加，直至将有限种全部包含。输出结果。

为使程序更用户友好，在最后加上“是否继续运行该程序”的判断。

## 核心算法

* + 1. select函数

存储输入的关系矩阵，选择算法。

* 函数中部分变量解释
  + n：关系矩阵行数或列数（阶数）
  + s[100][100]：存放关系矩阵的二维数组
* 函数分析

输入关系矩阵行数后，根据行数引导输入关系矩阵各元素，存储于二维数组s中。之后通过switch选择语句，根据用户需要的算法跳转至相应函数。

* + 1. duichen函数

求对称闭包。

* 函数中部分变量解释
  + s1[100][100]：存放关系矩阵的转置矩阵的二维数组
  + s2[100][100]：存放关系矩阵的二维数组
* 函数分析

求关系的对称闭包，则将关系矩阵加上（逻辑加）其转置矩阵即可得到。因此在处理中，先通过循环求得转置矩阵，并存入二维数组s1中。直接利用s2数组作为最终计算结果，即s2[i][j]=s1[i][j]+s2[i][j]。由于为bool类型数组，所以相加即为逻辑加。输出结果。

* + 1. chuandi函数

求传递闭包。

* 函数中部分变量解释
  + power[100][100]：存放M^n结果的数组
  + t[100][100]：存放每次加上去的结果的数组，最终作为传递闭包输出
  + compare[100][100]：存放M+M^2+…+M^(n-1)的结果，用于确认是否需要继续循环
  + temp[100][100]：存储M^(n-1)的数组
  + flag：判断是否需要继续循环的标志（0则表示还需要循环，1则表示已将所有不同的幂包括且不需要继续循环）
* 函数分析

求关系的传递矩阵，则需求得关系矩阵的不同的幂并累加（逻辑加），直至将有限种全部包含。因此需要通过循环求M^2、M^3、……

在处理中，初始化部分使power所有元素为0，t、compare、temp与关系矩阵相同。

接下来只要flag=0则进行累加操作。首先根据矩阵相乘的原理，利用temp（即M^n）和s2（即M）相乘得到power（即M^n）；接着将power复制给temp（变为M^n），t为加上M^n的累加结果；然后，先使flag=1，在通过compare（t-M^n）与t的比较判断是否将所有幂包含在内，只要有一个元素不同则flag=0（还没全包含）；最后，将t的结果赋值给compare，以便下一轮比较。

输出结果。

# C++语言源代码

#include <iostream>

using namespace std;

void output(bool s[][100]);

void zifan(bool s2[][100]);

void duichen(bool s2[][100]);

void chuandi(bool s2[][100]);

void select();

void exit();

bool s[100][100]; //s：存放关系矩阵的二维数组

int z,n,i,j; //z：算法方式编号；n：行数（阶数）

int main()

{

//标语

cout<<"\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n"; //标语

cout<<"\*\* \*\*\n";

cout<<"\*\* 欢迎进入关系的自反、对称 \*\*\n";

cout<<"\*\* 和传递闭包求解程序 \*\*\n";

cout<<"\*\* \*\*\n";

cout<<"\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n\n\n";

select();

return 0;

}

//存储关系矩阵，选择求解的闭包

void select()

{

//求解关系的自反、对称和传递闭包，则意味着该关系是某个非空集合上的关系，因此关系矩阵为方阵，只需知道行数（阶数）即可

//n：关系矩阵行数（阶数）

bb:cout<<"请输入关系矩阵的行数（阶数）（<=100）:";

cin>>n;

if(n<0||n>100)

{

cout<<"输入有误，请重新输入！\n"<<endl;

goto bb; //返回，重新输入

}

//输入关系矩阵

cout<<"\n请输入关系矩阵:"<<endl;

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

{

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

for(j = 0;j < n;j++)

cin>>s[i][j]; //s[100][100]：存放关系矩阵的二维数组

}

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

cin>>z; //z：算法编号

switch(z)

{

case 1:zifan(s); break; //l:自反闭包

case 2:chuandi(s);break; //2:传递闭包

case 3:duichen(s);break; //3:对称闭包

case 4:exit();break; //4:退出

}

}

//在一次求解结束后询问是否继续求解

void re()

{

char x; //存储y或n以表示继续或不继续

cout<<"\n是否继续运行该程序？（y/n）：";

cin>>x;

if(x=='y')

{

cout<<endl;

select();

}

else if(x=='n')

exit(1);

else

{

cout<<"输入错误，请重新输入。"<<endl;

re();

}

}

//显示算法结果

//传入的参数s[][100]为关系矩阵

void output(bool s[][100])

{

cout<<"\n所求关系矩阵为:"<<endl;

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

{

for(j = 0;j < n;j++)

cout<<s[i][j]<<" ";

cout<<endl;;

}

}

//求自反闭包

//传入的参数s2[][100]为关系矩阵

void zifan(bool s2[100][100])

{

//M=M+E（+为逻辑加，因此0+1=1，1+1=1，使关系矩阵对角线元素均为1即得其自反闭包）

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

s2[i][i] = 1;

output(s2);

re();

}

//求对称闭包

//传入的参数s2[][100]为关系矩阵

void duichen(bool s2[][100])

{

bool s1[100][100]; //s1：关系矩阵s2的转置矩阵

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

for(j = 0;j < n;j++)

s1[j][i] = s2[i][j];

//M=M+M'（求关系矩阵的对称闭包，即求关系矩阵与其转置矩阵逻辑加的结果）

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

for(j = 0;j < n;j++)

s2[i][j] = s1[i][j] + s2[i][j];

output(s2);

re();

}

//求传递闭包

//传入的参数s2[][100]为关系矩阵

void chuandi(bool s2[][100])

{

bool flag=0,t[100][100],temp[100][100],power[100][100],compare[100][100];

int h;

//对于各个变量初始化

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

for(j = 0;j < n;j++)

{

power[i][j] = 0; //power：存放M^n结果的数组

t[i][j] = s2[i][j]; //t：存放每次加上去的结果的数组

compare[i][j]=s2[i][j]; //compare：存放前一次+M^n的结果，用于确认是否需要继续循环

temp[i][j]=s2[i][j]; //temp：存储上一回power，即M^(n-1)

}

//进行Mt=M+M^2+M^3+…的计算

while(flag==0)

{

//计算M^n

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

for(j=0;j<n;j++)

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

//power为这一轮的M^n

//temp实际等于上一回power，即M^(n-1)；s2为M

power[i][j]=power[i][j]+temp[i][h]\*s2[h][j]; //compare实际等于上一回power，即M^(n-1)；s2为M

//将此轮M^n存入compare，计算累加和t

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

for(j = 0;j < n;j++)

{

temp[i][j]=power[i][j]; //temp完全复制此轮power（即M^n）

t[i][j]=t[i][j]+power[i][j]; //t为累加结果，Mt=M+M^2+M^3+……+M^n

}

flag=1; //将flag改为1，在接下来判断中只要发现t和compare有一个元素不同则flag=0，即需要继续循环

//判断是否需要继续循环

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

for(j = 0;j < n;j++)

//通过t与compare是否相等的判断，判断M^n是否达到了有限种

if(t[i][j]!=compare[i][j]) //compare为M+M^2+……+M^(n-1)

{

flag=0; //flag=1表示有限项M的幂均已涵盖，不必要继续循环

break;

}

//将本轮累加结果值t赋值给compare，在下一轮中可以对比

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

for(j = 0;j < n;j++)

compare[i][j]=t[i][j]; //compare为上一轮加完后t的值

}

output(t);

re();

}

void exit()

{

exit(1);

}

# 运行结果



图1欢迎界面

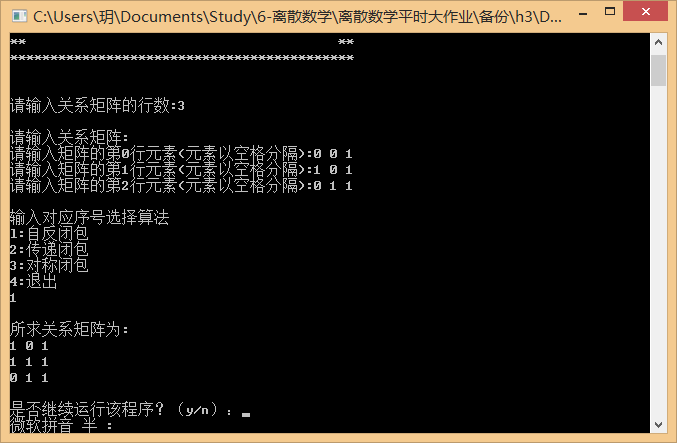


图2求关系的自反闭包

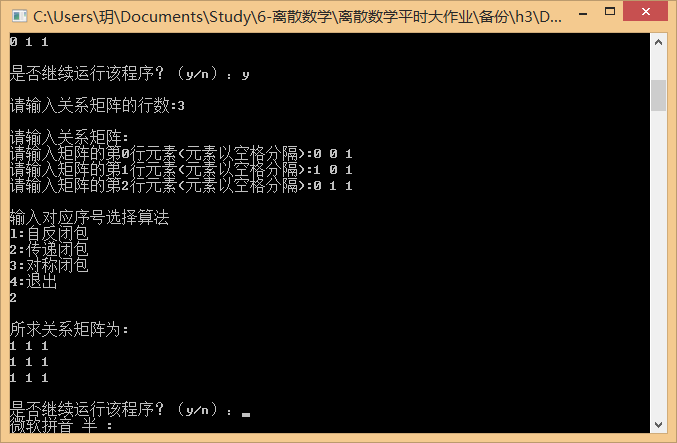


图3求关系的传递闭包

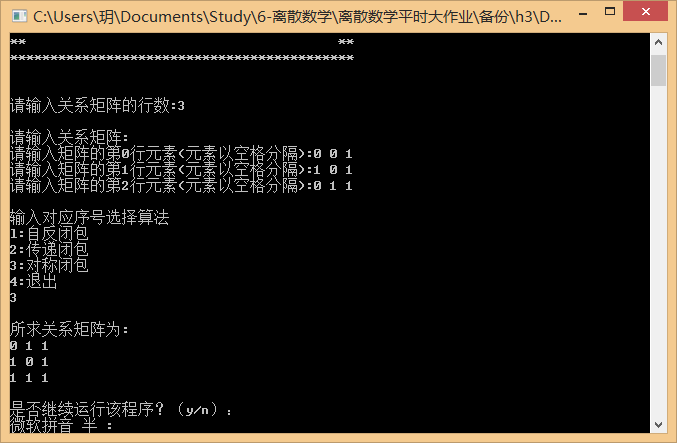


图4求关系的对称闭包

# 心得体会

实验3本身难度不算大，但为了使程序更加完善及便于理解，我做了不少改动，有些用了另一种思路考虑这些算法，但最终没写到代码中从而也认识到逻辑加在代码中处理的方式有很多。

同时，一个程序需要考虑方方面面，为了更用户友好，还增加了一些对于输入值的判断。

通过关系矩阵方法求解自反、对称、传递闭包的实验，我复习了相应公式形式的证明过程，对于这些关系的性质有了更深一步的理解。看到参考代码需要输入行数并且输入列数，也让我发现参考代码存在的一些可以改进的地方。实验3完成后，关系的性质与闭包相关知识掌握得更好了。