《离散数学》课程实验报告

6-Warshall算法求解传递闭包

一、题目背景简介

定义：传递闭包t(R)=R∪R2∪R3∪…

传统求传递闭包：考虑n+1个矩阵的序列M0，M1，…，Mn，将矩阵Mk第i行第j列的元素记作Mk[i,j]。对于k=0，1，…，n，Mk[i,j]=1当且仅当在R的关系图中存在一条从xi到xj的路径，并且这条路径除端点外中间只经过{x1，x2，…，xk}中的顶点。不难证明M0就是R的关系矩阵，而Mn就对应了R的传递闭包。

二、核心算法及依据

传统的求传递闭包的算法的时间复杂度是O(n4)，当n足够大时，程序跑起来比较慢。

有没有一种时间复杂度更低的算法呢？Warshall在1962年提出了一种求传递闭包的复杂度更低的Warshall算法。Warshall算法时间复杂度从传统的求传递闭包的算法的O(n4)降到了O(n3)。

Warshall算法的依据是：从M0开始，顺序计算M1，M2，…，直到Mn为止。从Mk[i,j]计算Mk+1[i,j]：i，j∈V的方法为顶点集V1={1，2，…，k}，V2={k+2，…，n}，V=V1∪{k+1}∪V2，那么Mk+1[i,j]=1当且仅当存在从i到j中间只经过V1∪{k+1}中点的路径。

这些路径分为两类：第1类是只经过V1中点；第2类是经过k+1点。若存在第1类路径，则Mk[i,j]=1；若存在第2类路径，则Mk[i,k+1]=1Mk[k+1,j]=1。

用伪代码表示Warshall算法为：

（1）置新矩阵A=M；

（2）置i=1；

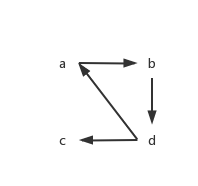
（3）对于所有j，如果A[j,i]=1，则对k=1，2，…，n，A[j,k]∨A[i,k]；

（4）i++；

（5）如果i≤n，则跳转到步骤（3），否则停止。

三、解题思路

这里我想用一个例题来进行解题思路的讲解。



上图对应的R={<a,b><b,d><d,a><d,c>}，根据二元关系易得R的关系矩阵：

M0=

先考察第1列元素，其中只有M0[4,1]等于1，于是应将第1行与第4行对应元素做布尔加，结果仍记在第4行上，得到：

M1=

再考察第2列元素，有M1[1,2]和M1[4,2]等于1，于是将第2行元素分别加到第1行和第4行，得到：

M2=

再考察第3列元素，仅有M2[4,3]等于1，于是将第3行元素加到第4行，得到的M3=M2。

最后考察第4列元素，有M3[1,4]、M3[2,4]和M3[4,4]等于1，按照要求分别进行布尔加，得到最终矩阵即为R的传递闭包t(R)的关系矩阵：

M4=

根据上述操作可以总结出状态转移方程为Mk[i,j]=Mk-1[i,j] or（Mk-1[i,k] and Mk-1[k,j]）。

于是，进行代码的框架构造。我打算用一个get\_matrix函数来从键盘输入获得R的关系矩阵、用一个output\_matrix函数对形参传递的二维数组进行元素的输出显示、用一个Warshall函数重点进行传递闭包的求解。

四、代码与运行结果

#include <iostream>

#include<cstring>

using namespace std;

#define R\_num 4//宏定义R中元素个数为4

bool get\_matrix(int M[R\_num][R\_num])

{

int i = 0, j = 0;

for (i = 0; i < R\_num; i++) {

for (j = 0; j < R\_num; j++) {

cin >> M[i][j];

if (M[i][j] != 0 && M[i][j] != 1)

return false;//若输入非0或1的数，则输入错误，返回false

}

}

return true;

}

void output\_matrix(int M[R\_num][R\_num])

{

int i = 0, j = 0;

for (i = 0; i < R\_num; i++) {

for (j = 0; j < R\_num; j++) {

cout << M[i][j] << ' ';

}

cout << endl;

}

}

void Warshall(int R[][4]) {

/\*(1)i＝1；

(2)对所有j如果a[j，i]＝1，则对k＝0，1，…，n-1，a[j，k]＝a[j，k]∨a[i，k]；

(3)i加1；

(4)如果i<n，则转到步骤2，否则停止\*/

int i = 0, j = 0, k = 0;

for (k = 0; k < R\_num; k++) {//进行R\_num-1次R矩阵的变换，得到R的R\_num次方

for (i = 0; i < R\_num; i++) {

for (j = 0; j < R\_num; j++) {

R[i][j] = R[i][j] || (R[i][k] && R[k][j]);

}

}

}

output\_matrix(R);//输出t(R)的关系矩阵

}

int main()

{

int M[R\_num][R\_num] = { 0 };

PART:

cout << "请输入一个" << R\_num << "×" << R\_num << "的关系矩阵M" << endl;

if (!get\_matrix(M)) {

cout << "输入错误，矩阵元素只能为0或1！" << endl << "请重新输入：" << endl;

goto PART;

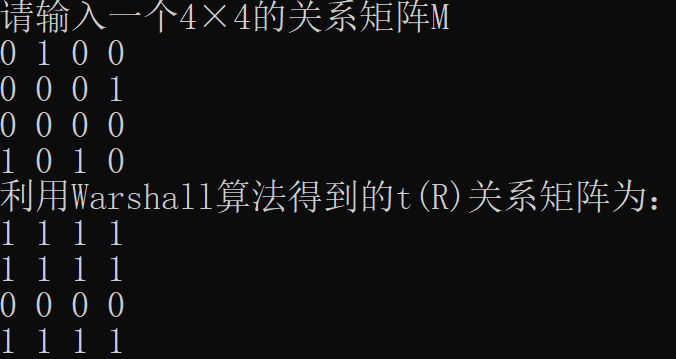
}

cout << "利用Warshall算法得到的t(R)关系矩阵为：" << endl;

Warshall(M);

return 0;

}

运行结果：

五、体会与心得

由于先前在大作业中看到需要实现Warshall算法，于是在老师讲课的时候仔细听讲并揣摩了一番。不同于考察每一列后进行布尔加，老师讲到的另一种理解思路是：Mk-1矩阵的第k行第k列的元素不动，对于其余的元素Mk-1[i,j]（i≠k 且j≠k），若为1，则依然保持为1,；若为0，则Mk-1[i,j]由其在第k行第k列的元素的与值决定。我跟着老师对照例题过了一遍，自以为深刻明白了Warshall算法的要义。

然而本次在做大作业，根据网上相关文档的解释，我发现与先前的了解又有些无法联系。但是经过一番阅读之后，我发现了：无论是我课上弄明白的思路，还是今天在项目中讨论的思路，都围绕着一个要义，就是我在前文提到的总结状态转移方程：Mk[i,j]=Mk-1[i,j] or（Mk-1[i,k] and Mk-1[k,j]）。

基于对Warshall算法双重理解的基础再进行代码的编写就变得轻松许多。

当然，在网上浏览了一些其他人的代码之后，我发现我的代码还有可修改或完善的地方：可以将输入关系矩阵变为输入R中的所有二元关系；关系矩阵的输入可以不固定矩阵的大小，从而减少了输入的限制，增强代码的应用全面性。然而，我觉得如果要更改为输入R中的所有二元关系，那么输入的工作量就更大，也更容易从输入端产生错误，所以还是pass了这种想法。