|  |
| --- |
| 离散数学实验报告 |
| 学号：\_\_\_\_1120220715\_\_\_\_\_\_\_ 姓名：\_\_\_\_刘秉致\_\_\_\_\_\_\_\_\_\_\_ |

**目 录**

[1 求命题的主范式 1](#_Toc154925882)

[1.1 **概述** 1](#_Toc154925883)

[1.2 **步骤流程** 1](#_Toc154925884)

[1.3 **程序实现** 1](#_Toc154925885)

[1.3.1 **PriorityofOperator** 2](#_Toc154925886)

[1.3.2 **evalRPN** 2](#_Toc154925887)

[1.3.3 **dual\_cal** 3](#_Toc154925888)

[1.3.4 **sing\_cal** 3](#_Toc154925889)

[1.3.5 **dfs** 3](#_Toc154925890)

[1.4 **最终代码** 4](#_Toc154925891)

[2 消解算法 35](#_Toc154925892)

[2.1 **概述** 35](#_Toc154925893)

[2.2 **步骤流程** 35](#_Toc154925894)

[2.3 **程序实现** 35](#_Toc154925895)

[2.3.1 **Init函数** 36](#_Toc154925896)

[2.3.2 **Same函数** 36](#_Toc154925897)

[2.3.3 **Cal函数** 36](#_Toc154925898)

[2.4 **最终代码** 36](#_Toc154925899)

[3 求关系的传递闭包 48](#_Toc154925900)

[3.1 **概述** 48](#_Toc154925901)

[3.2 **步骤流程** 48](#_Toc154925902)

[3.3 **程序实现** 49](#_Toc154925903)

[3.3.1 **Logic\_add函数** 49](#_Toc154925904)

[3.3.2 **Get\_point函数** 49](#_Toc154925905)

[3.4 **最终代码** 49](#_Toc154925906)

[4 求偏序关系的极小元和极大元 54](#_Toc154925907)

[4.1 **概述** 54](#_Toc154925908)

[4.2 **步骤梳理** 54](#_Toc154925909)

[4.3 **程序实现** 54](#_Toc154925910)

[4.3.1 **Number函数** 55](#_Toc154925911)

[4.3.2 **Deal函数** 55](#_Toc154925912)

[4.4 **最终代码** 55](#_Toc154925913)

[5 代数系统算律的判断 61](#_Toc154925914)

[5.1 **概述** 61](#_Toc154925915)

[5.2 步骤流程 61](#_Toc154925916)

[5.3 **程序实现** 61](#_Toc154925917)

[5.3.1 **isCommutative函数** 62](#_Toc154925918)

[5.3.2 **isAssociative函数** 62](#_Toc154925919)

[5.3.3 **isIdempotent函数** 62](#_Toc154925920)

[5.3.4 **identify函数** 62](#_Toc154925921)

[5.3.5 **zero函数** 62](#_Toc154925922)

[5.4 最终代码 62](#_Toc154925923)

[6 模n加群的元素的阶 70](#_Toc154925924)

[6.1 **概述** 70](#_Toc154925925)

[6.2 **步骤流程** 70](#_Toc154925926)

[6.3 程序实现 70](#_Toc154925927)

[6.3.1 mc函数 70](#_Toc154925928)

[6.4 最终代码 70](#_Toc154925929)

[7 二部图的判定 73](#_Toc154925930)

[7.1 **概述** 73](#_Toc154925931)

[7.2 **步骤流程** 73](#_Toc154925932)

[7.3 **程序实现** 73](#_Toc154925933)

[7.3.1 BFS函数 74](#_Toc154925934)

[7.4 **最终代码** 74](#_Toc154925935)

[8 有向图的判定 78](#_Toc154925936)

[8.1 **概述** 78](#_Toc154925937)

[8.2 **步骤流程** 78](#_Toc154925938)

[8.3 **程序实现** 79](#_Toc154925939)

[8.3.1 Logicadd函数 79](#_Toc154925940)

[8.3.2 Washell函数 79](#_Toc154925941)

[8.4 **最终代码** 79](#_Toc154925942)

**表目录**

[表 1求命题的主范式函数说明表 2](#_Toc154925943)

[表 2 消解算法主程序函数说明表 36](#_Toc154925944)

[表 3求关系的传递闭包的函数表 49](#_Toc154925945)

[表 4 求极大元极小元函数说明表 54](#_Toc154925946)

[表 5 代数系统的算律判断函数表 61](#_Toc154925947)

[表 6 求Zn中元素的阶的函数说明表 70](#_Toc154925948)

[表 7 二部图判定函数说明表 73](#_Toc154925949)

[表 8 有向图的判定函数说明表 79](#_Toc154925950)

**图目录**

[图 1 求命题的主范式流程图 1](#_Toc154925951)

[图 2 evalRPN 函数执行流程 3](#_Toc154925952)

[图 3 消解算法主程序流程图 35](#_Toc154925953)

[图 4 求关系的传递闭包 48](#_Toc154925954)

[图 5 求极大元极小元的步骤流程图 54](#_Toc154925955)

[图 6 代数系统算律的判断流程 61](#_Toc154925956)

[图 7 求Zn中元素的阶的主要流程图 70](#_Toc154925957)

[图 8 二部图判定的主要流程图 73](#_Toc154925958)

[图 9 有向图的判定的主要流程图 78](#_Toc154925959)

# 求命题的主范式

## **概述**

输入命题公式的合式公式，求出公式的真值表，并输出该公式的主合取范式和主析取范式。

## **步骤流程**

程序运行主逻辑如图1所示。

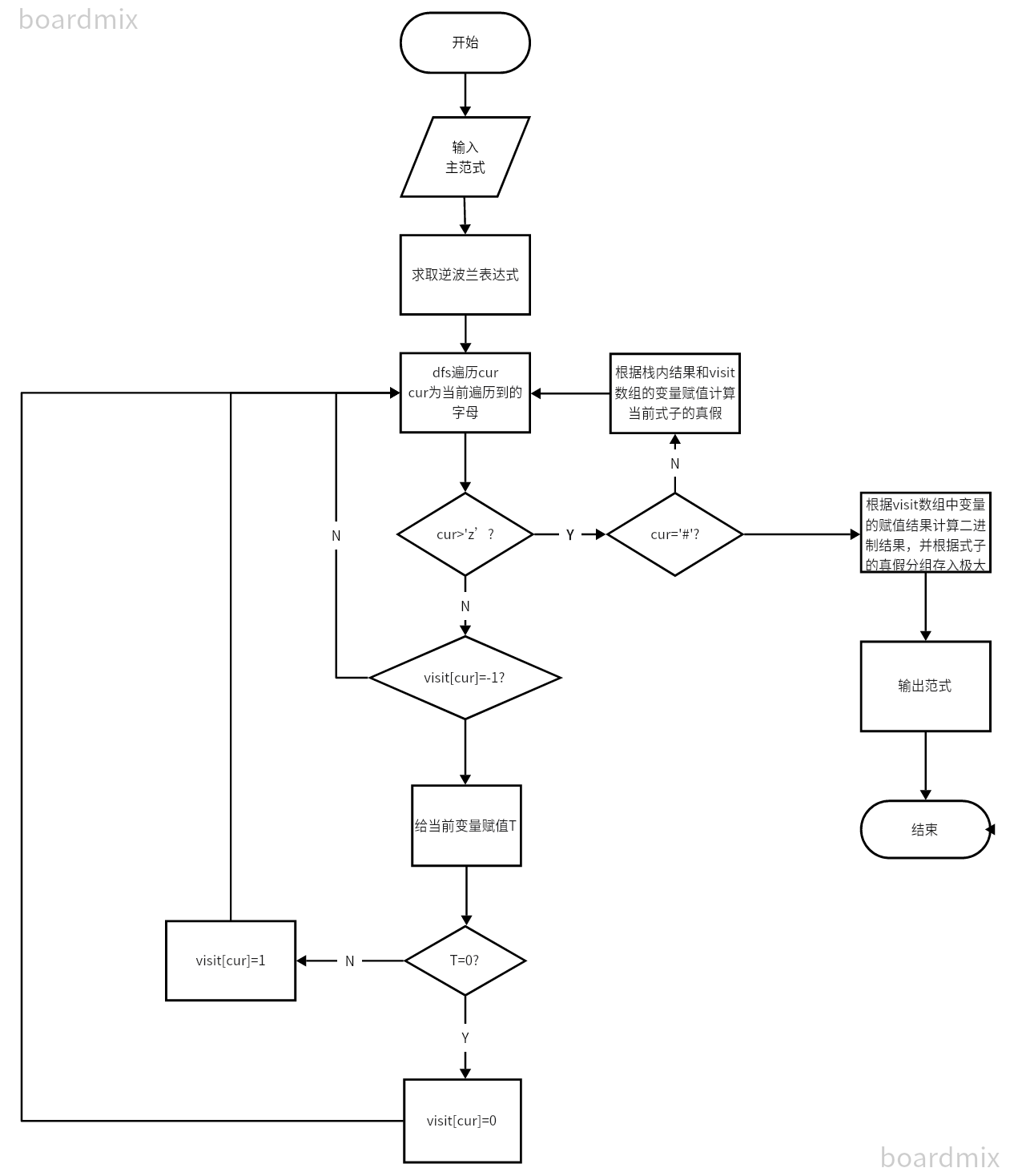


图 1 求命题的主范式流程图

## **程序实现**

表 1求命题的主范式函数说明表

| 序号 | 名称 | 说明 |
| --- | --- | --- |
|  | PriorityofOperator | 用于归定和处理运算符的优先级 |
|  | evalRPN | 用于将输入的字符串转成逆波兰表达式 |
|  | Dual\_cal | 进行双目运算符的运算 |
|  | Sing\_cal | 进行单目运算符的运算 |
|  | Dfs | 对储存的逆波兰式进行深度遍历 |

### **PriorityofOperator**

用于归定和处理运算符的优先级，其中status参数记录当前运算处在第status层括号中，status=0时表示处于括号外；

括号内运算优先级：否定>析取、合取>蕴含>等价>左括号>右括号

括号外运算优先级：左括号>右括号>否定>析取、合取>蕴含>等价

### **evalRPN**

用于将输入的字符串转成逆波兰表达式，其中就要使用函数一作为辅助。 依次遍历整个字符串，如果为字母，则直接输出到存储结果的数组中，如果为运算符&、|、+、-，则需要与栈顶运算符的优先级进行比较，处理类似 ！插队（优先级更高）的现象，如果为（ 则直接入栈，如果为），则一直弹出栈顶元素直到找到与之匹配的左括号。最终原表达式遍历完成，将存在栈内未弹出的所有字符弹出到储存结果的数组末尾并追加‘#’符号表示终止，过程中记录逆波兰式的字符串长度，便于之后利用。

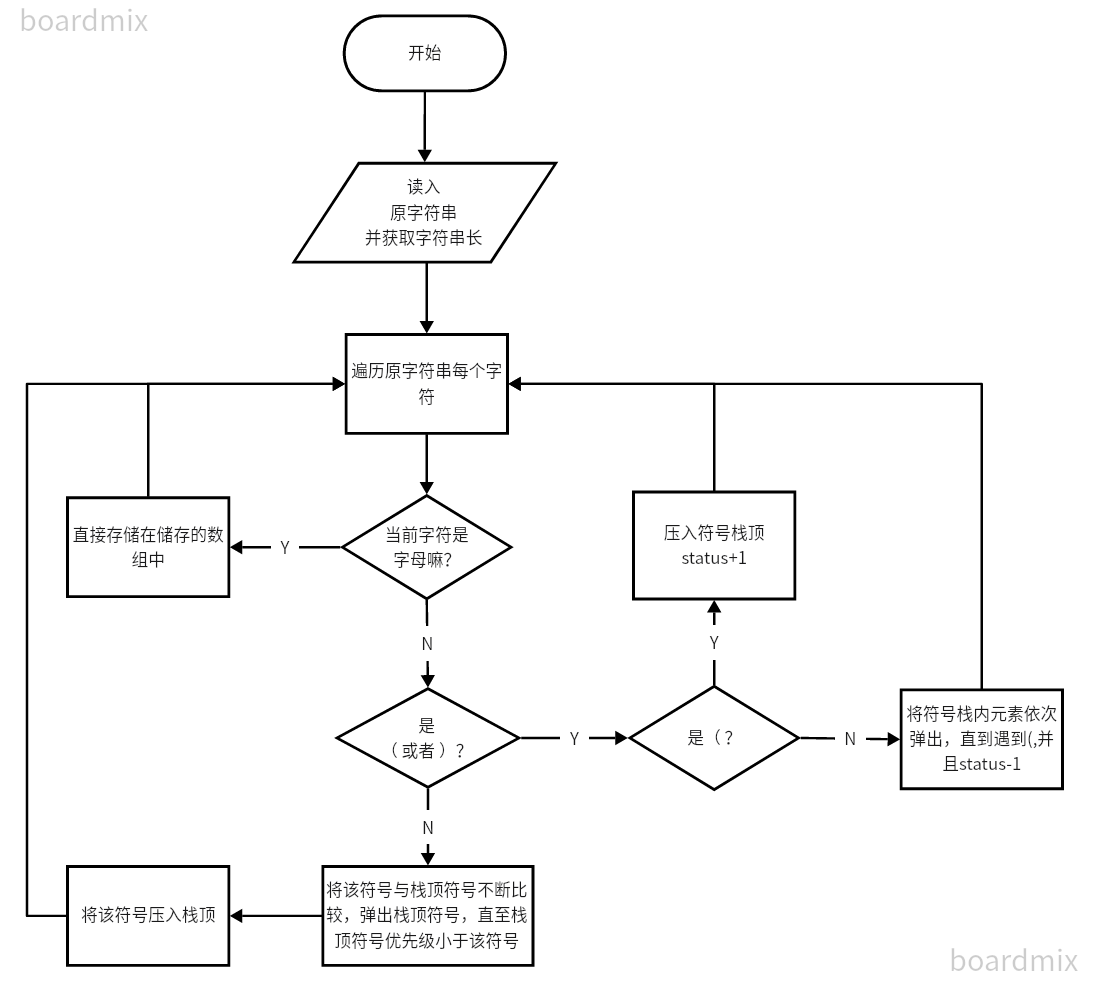


图 2 evalRPN 函数执行流程

### **dual\_cal**

用于计算得出两个变量间的双目运算结果，可进行的运算包括：合取、析取、蕴含、等价。

### **sing\_cal**

用于计算得出单个变量的逆的运算结果。

### **dfs**

用于对储存好的逆波兰表达式中的每个字符进行深度优先遍历。

Cur指针指向当前字符，若当前字符为变量，则先根据visit数组判断该变量是否已经被赋值。如果是，则visit[cur]!=1,忽略对该变量的赋值操作，压栈并继续深度遍历；若果否，则分别对当前变量赋值为0或1，将结果存入visit[cur]中，压栈并继续深度遍历。

若该字符为运算符，则先判断该字符是不是结束符‘#’。如果否，则取出栈顶变量进行运算并将运算结果压栈，继续深度遍历。如果是，则根据visit数组中储存的变量的赋值情况，计算二进制对应的十进制结果。并根据式子的真假区分极大项和极小项，分别储存。

## **最终代码**

#include <bits/stdc++.h>

using namespace std;

#define max\_size 2048

/\*基本思路：

  1.(core)先将合式读入，将其转化为逆波兰式储存至新的数组内，

    并开数组记录变量是否被访问过

  2.(core)将指针指向最开头第一个变量，从此开始dfs遍历每一种可能，

    记录极大项和极小项出现的次数和赋值的内容（利用vector数组）

  3.输出结果

\*/

//全局变量域

char origin[max\_size];//用来储存原串

int visit[max\_size];//一个全局数组，用来记录某一个变量是否被访问(未访问未-1，访问后与赋值相同）

char RPN[max\_size];//用来储存逆波兰式的数组

//记录极大项和极小项的信息的结构体

typedef struct{

    int array[max\_size];

    int lenth;

}LOG;

LOG bigone;

LOG smallone;

int var\_value[max\_size];//记录过程中变量赋值的数组

stack<int>num\_stack;//dfs中记录数值内容的栈

//函数域

bool PriorityofOperator(char op1,char op2,int status){

/\*处理运算符的函数--比较两运算符大小，

  前两个参数为两运算符，最后参数代表状态：置于status层括号内：括号内>0还是括号外=0

  如果op1比op2的运算级高则返回true,否则返回false；

  tips:除去括号的优先级：否定>合取析取>蕴含>等价

\*/

    if(status>0){//此时在括号内->  !>&>|>->+>(>)

        if(op1=='('){//栈顶运算符为（

            if(op2=='!'){

                return false;

            }

            else if(op2=='&'){

                return false;

            }

            else if(op2=='|'){

                return false;

            }

            else if(op2=='-'){

                return false;

            }

            else if(op2=='+'){

                return false;

            }

            else if(op2=='('){

                return false;

            }

            else if(op2==')'){

                return true;

            }

        }

        else if(op1==')'){//栈顶运算符为）

            if(op2=='!'){

                return false;

            }

            else if(op2=='&'){

                return false;

            }

            else if(op2=='|'){

                return false;

            }

            else if(op2=='-'){

                return false;

            }

            else if(op2=='+'){

                return false;

            }

            else if(op2=='('){

                return false;

            }

            else if(op2==')'){

                return false;

            }

        }

        else if(op1=='!'){//栈顶运算符为！

            if(op2=='!'){

                return true;

            }

            else if(op2=='&'){

                return true;

            }

            else if(op2=='|'){

                return true;

            }

            else if(op2=='-'){

                return true;

            }

            else if(op2=='+'){

                return true;

            }

            else if(op2=='('){

                return true;

            }

            else if(op2==')'){

                return true;

            }

        }

        else if(op1=='&'){//栈顶运算符为&

            if(op2=='!'){

                return false;

            }

            else if(op2=='&'){

                return true;

            }

            else if(op2=='|'){

                return true;

            }

            else if(op2=='-'){

                return true;

            }

            else if(op2=='+'){

                return true;

            }

            else if(op2=='('){

                return true;

            }

            else if(op2==')'){

                return true;

            }

        }

        else if(op1=='|'){//栈顶运算符为|

            if(op2=='!'){

                return false;

            }

            else if(op2=='&'){

                return false;

            }

            else if(op2=='|'){

                return true;

            }

            else if(op2=='-'){

                return true;

            }

            else if(op2=='+'){

                return true;

            }

            else if(op2=='('){

                return true;

            }

            else if(op2==')'){

                return true;

            }

        }

        else if(op1=='-'){//栈顶运算符为-

            if(op2=='!'){

                return false;

            }

            else if(op2=='&'){

                return false;

            }

            else if(op2=='|'){

                return false;

            }

            else if(op2=='-'){

                return true;

            }

            else if(op2=='+'){

                return true;

            }

            else if(op2=='('){

                return true;

            }

            else if(op2==')'){

                return true;

            }

        }

        else if(op1=='+'){//栈顶运算符为+

            if(op2=='!'){

                return false;

            }

            else if(op2=='&'){

                return false;

            }

            else if(op2=='|'){

                return false;

            }

            else if(op2=='-'){

                return false;

            }

            else if(op2=='+'){

                return true;

            }

            else if(op2=='('){

                return true;

            }

            else if(op2==')'){

                return true;

            }

        }

    }

    else{//此时在括号外-> (>)>!>&>|>->+

        if(op1=='('){//栈顶运算符为（

            if(op2=='!'){

                return true;

            }

            else if(op2=='&'){

                return true;

            }

            else if(op2=='|'){

                return true;

            }

            else if(op2=='-'){

                return true;

            }

            else if(op2=='+'){

                return true;

            }

            else if(op2=='('){

                return true;

            }

            else if(op2==')'){

                return true;

            }

        }

        else if(op1==')'){//栈顶运算符为）

            if(op2=='!'){

                return true;

            }

            else if(op2=='&'){

                return true;

            }

            else if(op2=='|'){

                return true;

            }

            else if(op2=='-'){

                return true;

            }

            else if(op2=='+'){

                return true;

            }

            else if(op2=='('){

                return false;

            }

            else if(op2==')'){

                return true;

            }

        }

        else if(op1=='!'){//栈顶运算符为！

            if(op2=='!'){

                return true;

            }

            else if(op2=='&'){

                return true;

            }

            else if(op2=='|'){

                return true;

            }

            else if(op2=='-'){

                return true;

            }

            else if(op2=='+'){

                return true;

            }

            else if(op2=='('){

                return false;

            }

            else if(op2==')'){

                return false;

            }

        }

        else if(op1=='&'){//栈顶运算符为&

            if(op2=='!'){

                return false;

            }

            else if(op2=='&'){

                return true;

            }

            else if(op2=='|'){

                return true;

            }

            else if(op2=='-'){

                return true;

            }

            else if(op2=='+'){

                return true;

            }

            else if(op2=='('){

                return false;

            }

            else if(op2==')'){

                return false;

            }

        }

        else if(op1=='|'){//栈顶运算符为|

            if(op2=='!'){

                return false;

            }

            else if(op2=='&'){

                return false;

            }

            else if(op2=='|'){

                return true;

            }

            else if(op2=='-'){

                return true;

            }

            else if(op2=='+'){

                return true;

            }

            else if(op2=='('){

                return false;

            }

            else if(op2==')'){

                return false;

            }

        }

        else if(op1=='-'){//栈顶运算符为-

            if(op2=='!'){

                return false;

            }

            else if(op2=='&'){

                return false;

            }

            else if(op2=='|'){

                return false;

            }

            else if(op2=='-'){

                return true;

            }

            else if(op2=='+'){

                return true;

            }

            else if(op2=='('){

                return false;

            }

            else if(op2==')'){

                return false;

            }

        }

        else if(op1=='+'){//栈顶运算符为+

            if(op2=='!'){

                return false;

            }

            else if(op2=='&'){

                return false;

            }

            else if(op2=='|'){

                return false;

            }

            else if(op2=='-'){

                return false;

            }

            else if(op2=='+'){

                return true;

            }

            else if(op2=='('){

                return false;

            }

            else if(op2==')'){

                return false;

            }

        }

    }

}

int evalRPN(char origin[],int \*varnum){//转换为逆波兰式函数

    int maxlenth=strlen(origin);//记录最大长度

    int lenth=0;//记录逆波兰式的长度

    int status=0;//记录括号内外状态的函数；

    std::stack<char> op;//储存符号的栈

    for(int i=0;i<maxlenth;i++){

        if(origin[i]>='a'&&origin[i]<='z'){//当前为变量,直接记录在新数组内

            RPN[lenth]=origin[i];

            lenth++;

        }

        else{//当前为符号

            if(origin[i]=='('){//遇到左括号，直接放入栈顶

                op.push(origin[i]);

                status++;//更新状态

                continue;

            }

            else if(origin[i]==')'){//遇到右括号，直接将栈内直到它最近的那个左括号之间的进行弹出

                if(op.empty()!=true){//当栈内非空时

                    char front;//记录栈顶元素

                    while(op.empty()!=true){//当栈内非空时

                        front=op.top();

                        if(front=='('){//遇到左括号，直接退出

                            break;

                        }

                        else{//非左括号则记录进RPN

                            RPN[lenth]=front;

                            lenth++;

                            op.pop();

                        }

                    }

                    if(op.empty()!=true){//在非空时退出，栈顶必为左括号,弹出一个左括号

                        op.pop();

                        status--;//更新状态

                        continue;

                    }

                    else{//未找到匹配的左括号，出错了

                        printf("ERROR1\n");

                        continue;

                    }

                }

                else{//当栈内为空时，出错了

                    printf("ERROR2\n");

                    continue;

                }

            }

            else{//对于其他情况

                if(op.empty()==true){//栈内为空，直接放

                    op.push(origin[i]);

                }

                else{//栈内非空

                    char front=op.top();

                    if(PriorityofOperator(front,origin[i],status)==false){//栈顶优先级低，入栈

                        op.push(origin[i]);

                    }

                    else{//栈顶优先级高，先弹出

                        while(op.empty()!=true){//在栈内非空时

                            front=op.top();

                            if(PriorityofOperator(front,origin[i],status)==false){//直到栈顶优先级低为止

                                break;

                            }

                            else{

                                RPN[lenth]=front;

                                lenth++;

                                op.pop();

                                continue;

                            }

                        }

                        op.push(origin[i]);

                    }

                }

            }

        }

    }

    //当所有字符被遍历后，就可以直接将栈内内容一个个弹出来了

    while(op.empty()!=true){

        char front=op.top();

        RPN[lenth]=front;

        lenth++;

        op.pop();

    }

    //加个结束标识符

    RPN[lenth]='#';

    lenth++;

    return lenth;

}

int dual\_cal(int num1,int num2,char op){//双目运算区

    if(op=='&'){

        if(num1==1&&num2==1){

            return 1;

        }

        else{

            return 0;

        }

    }

    else if(op=='|'){

        if(num1==1||num2==1){

            return 1;

        }

        else{

            return 0;

        }

    }

    else if(op=='-'){

        if(num1==1&&num2==0){

            return 0;

        }

        else{

            return 1;

        }

    }

    else if(op=='+'){

        if(num1==num2){

            return 1;

        }

        else{

            return 0;

        }

    }

    else{

        printf("dual ERROR\n");

        return -1;

    }

}

int sing\_cal(int num1,char op){//单目预算区

    if(op=='!'){

        if(num1==0){

            return 1;

        }

        else{

            return 0;

        }

    }

    else{

        printf("sing ERROR\n");

        return -1;

    }

}

void dfs(int cur,int cur\_var){//cur指向当前位置，cur\_var记录已被赋值的变量个数

    if(RPN[cur]=='#'){//到了底了

        int final\_result=num\_stack.top();

        if(final\_result==0){//成假，为一极大项

            int temp=0;

            for(int time=0,point=cur\_var-1;point>=0;point--,time++){//计算对应的项数

                temp+=(var\_value[point]\*(int)pow(2,time));

            }

            bigone.array[bigone.lenth]=temp;

            bigone.lenth++;

            return;

        }

        else if(final\_result==1){//成真，为一极小项

            int temp=0;

            for(int time=0,point=cur\_var-1;point>=0;point--,time++){//计算对应的项数

                temp+=(var\_value[point]\*(int)pow(2,time));

            }

            smallone.array[smallone.lenth]=temp;

            smallone.lenth++;

            return ;

        }

        else{

            printf("no 0 no 1 ERROR\n");

            return ;

        }

    }

    else if(RPN[cur]>='a'&&RPN[cur]<='z'){//当前为一个变量

        if(visit[RPN[cur]-97]!=-1){//已经被赋值了

            num\_stack.push(visit[RPN[cur]-97]);

            dfs(cur+1,cur\_var);//递归到下一层

            num\_stack.pop();//将影响消除

            return;

        }

        else{//未被访问过

            //先设置该变量为0

            var\_value[cur\_var]=0;

            visit[RPN[cur]-97]=0;

            num\_stack.push(visit[RPN[cur]-97]);

            dfs(cur+1,cur\_var+1);

            num\_stack.pop();

            //再设置为1

            var\_value[cur\_var]=1;

            visit[RPN[cur]-97]=1;

            num\_stack.push(visit[RPN[cur]-97]);

            dfs(cur+1,cur\_var+1);

            //清楚影响，返回

            visit[RPN[cur]-97]=-1;

            num\_stack.pop();

            return;

        }

    }

    else{//当前为一个非结束的符号

        if(RPN[cur]=='!'){//为一个单目的

            int num1=num\_stack.top();

            int result=sing\_cal(num1,RPN[cur]);

            num\_stack.pop();

            num\_stack.push(result);

            dfs(cur+1,cur\_var);

            //消除影响返回

            num\_stack.pop();

            num\_stack.push(num1);

            return ;

        }

        else{//为一双目的

            int num2=num\_stack.top();

            num\_stack.pop();

            int num1=num\_stack.top();

            int result=dual\_cal(num1,num2,RPN[cur]);

            num\_stack.pop();

            num\_stack.push(result);

            dfs(cur+1,cur\_var);

            //消除影响返回

            num\_stack.pop();

            num\_stack.push(num1);

            num\_stack.push(num2);

            return ;

        }

    }

}

int main(){

    //part 1:前置条件，读入原始合式

    memset(origin,0,sizeof(origin));

    gets(origin);//读入origin

    memset(RPN,0,sizeof(RPN));//将逆波兰式数组重置

    //part2:转换为逆波兰式,记录可能的变量个数

    int varnum=0;//记录变量个数

    int RPNlenth=0;//记录逆波兰式的长度，最后跟标识符#

    RPNlenth=evalRPN(origin,&varnum);

    //part3:从第一个变量开始，进行dfs

    bigone.lenth=0;

    smallone.lenth=0;

    memset(smallone.array,0,sizeof(smallone.array));//将极小项数组重置

    memset(bigone.array,0,sizeof(bigone.array));//将极大项数组重置

    memset(visit,-1,sizeof(visit));//将全局数组visit重置

    dfs(0,0);

    //part4:输出结果

    //先输出主析取范式

    if(smallone.lenth==0){

        printf("0 ; ");

    }

    else{

        for(int i=0;i<smallone.lenth;i++){

            printf("m%d",smallone.array[i]);

            if(i<smallone.lenth-1){

                printf(" ∨ ");

            }

        }

        printf(" ; ");

    }

    //再输出主合取范式

    if(bigone.lenth==0){

        printf("1\n");

    }

    else{

        for(int i=0;i<bigone.lenth;i++){

            printf("M%d",bigone.array[i]);

            if(i<bigone.lenth-1){

                printf(" ∧ ");

            }

        }

        printf("\n");

    }

}

# 消解算法

## **概述**

输入合式公式的合取范式，判断公式的可满足性，当公式可满足时输出“YES”,否则输出“NO”。

## **步骤流程**

程序的主要流程如图2所示

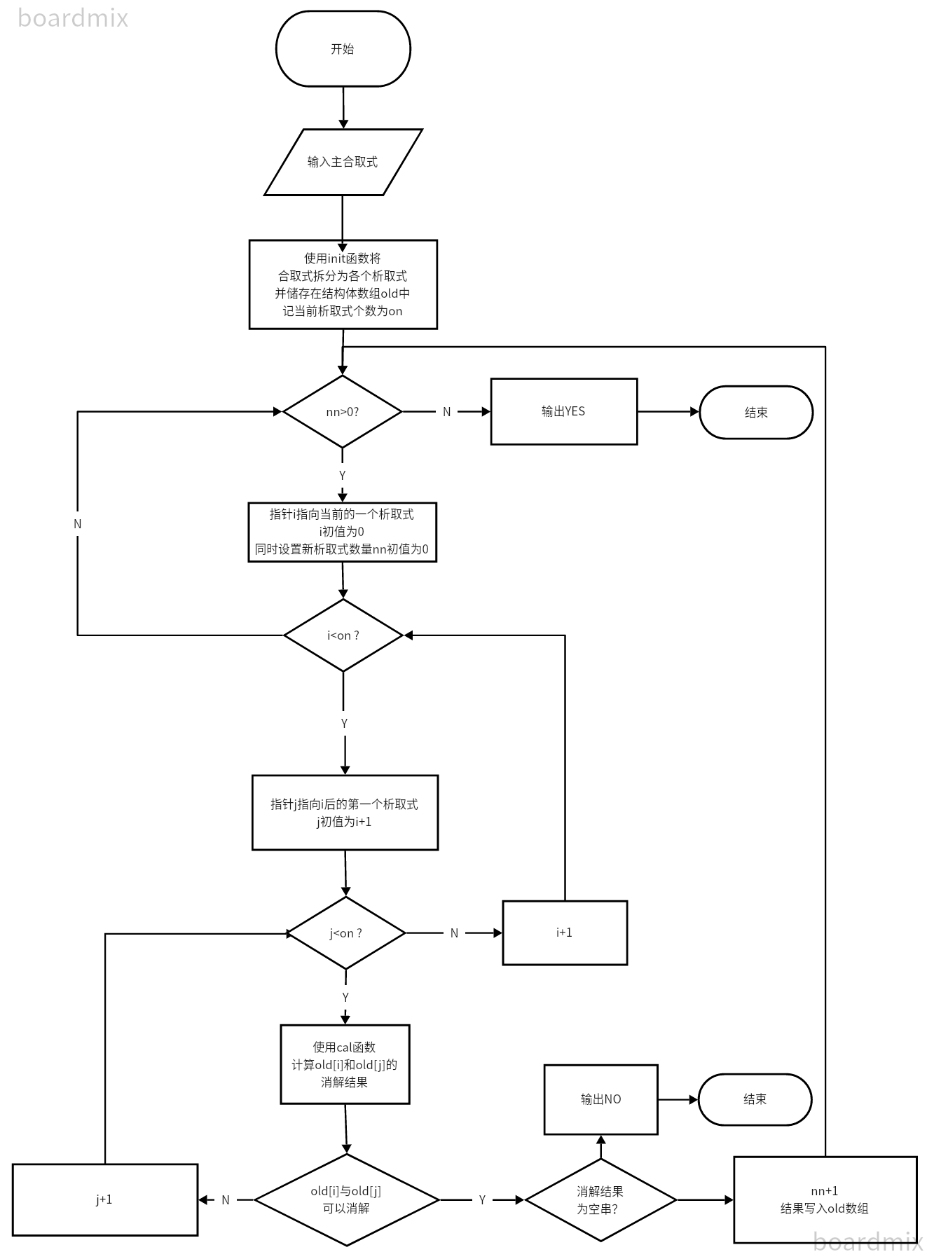


图 3 消解算法主程序流程图

## **程序实现**

表 2 消解算法主程序函数说明表

| 序号 | 名称 | 说明 |
| --- | --- | --- |
| 1. | Init | 用于将主合取式拆分为小的析取子式并储存 |
| 2. | same | 用于判断两个析取子式是否一样 |
| 3. | cal | 用于计算两个析取子式的结果 |

### **Init函数**

Init函数使用strtok函数,将输入的主合取式以‘&’符号为分隔符进行分割，并将获得的各个析取子式存在结构体数组中。

结构体内包含两个数据元：一个是该析取子式中包含的变量个数，一个是该析取子式中的各文字。

### **Same函数**

通过遍历两个析取子式的各个文字判断两者是否一样。

### **Cal函数**

通过遍历两个析取子式的各个变量，查找是否存在变量使得两式可以消解。若不可以,则返回0；若可以，则进行消解，并判断消解结果是否为空串。若为空串则返回-1；否则检查是否与old数组中的已有的式子重复，重复则返回0；否则返回1，并将产生的结果加入到old数组中，nn加1。

## **最终代码**

#include <bits/stdc++.h>

#define max\_size 2048

#define alphabet 30

/\*基本思路（消解思路）：

   1.先拿到原始的主合取式后，将其拆分为各个析取式(用二维表记录）然后进队储存

   2.然后从队中第一个开始向后循环，每次循环中向后取，判断是否能够获得消解式

     不能就继续，能得就把消解式进队并且记录新式子加1

   3.如果遇到了空式子->不可满足，直接退出返回status=false,

     如果新式子=0，意味着所有可能的式子都已得出，则返回true

   4.输出结果

\*/

//全局变量域

char origin[max\_size];//存储原始数据的数组

typedef struct{//存储析取式信息的结构体

    int table[alphabet];//储存变量信息的数组，数组下标对应为（变量名-97），值对应变量的状态（-1没有，0为!p，1为p）

    int num=0;//记录变量个数

}cal\_part;

cal\_part old\_array[max\_size];//程序中进行堆叠的数组

cal\_part new\_array[max\_size];//程序中进行堆叠的数组

int init(char origin[],cal\_part array[]){//构建函数

    int arraynum=0;

    for(char \*p=strtok(origin,"&");p!=NULL;p=strtok(NULL,"&")){

        if(p[0]!='\n'){//截取到了有效内容

            //初始化

            memset(array[arraynum].table,-1,sizeof(array[arraynum].table));

            array[arraynum].num=0;

            //开始进行处理

            int lenth=strlen(p);

            for(int i=0;i<lenth;i++){

                if(p[i]>='a'&&p[i]<='z'){//是个变量

                    array[arraynum].num++;//计数先加一

                    if(i>0&&p[i-1]=='!'){//为！p时

                        array[arraynum].table[p[i]-97]=0;

                    }

                    else{//为p时

                        array[arraynum].table[p[i]-97]=1;

                    }

                }

                else{//是个符号

                    continue;

                }

            }

            //收尾

            arraynum++;

        }

        else{//截到最后的换行了

            continue;

        }

    }

    return arraynum;

}

bool same(cal\_part a,cal\_part b){//判断两个式子是否一样

    if(a.num!=b.num){

        return false;

    }

    else{

        bool dif=false;

        for(int i=0;i<alphabet;i++){

            if(a.table[i]!=b.table[i]){

                dif=true;

                break;

            }

        }

        if(dif==false){

            return true;

        }

        else{

            return false;

        }

    }

}

int cal(cal\_part a,cal\_part b,int old\_number,int new\_number){//消解的函数(消解成功返回1，不成功返回0，出现空式子返回-1）

    bool status=false;//消解状态（true为进入消解态，fasle为非消解态）

    //先判断两个是不是可以消解（出现一个变元反着）

    for(int i=0;i<alphabet;i++){

        if(a.table[i]==0&&b.table[i]==1){

            a.table[i]=-1;

            b.table[i]=-1;

            a.num--;

            b.num--;

            status=true;

            break;

        }

        else if(a.table[i]==1&&b.table[i]==0){

            a.table[i]=-1;

            b.table[i]=-1;

            a.num--;

            b.num--;

            status=true;

            break;

        }

        else{

            continue;

        }

    }

    //如果不可以消解

    if(status==false){

        return 0;

    }

    else{//如果可以消解

        for(int i=0;i<alphabet;i++){

            if(a.table[i]!=-1||b.table[i]!=-1){//有效变元出现

                new\_array[new\_number].num++;

                if(a.table[i]==b.table[i]){//两个变元相同

                    new\_array[new\_number].table[i]=a.table[i];

                }

                else if(a.table[i]==-1){//a没有

                    new\_array[new\_number].table[i]=b.table[i];

                }

                else if(b.table[i]==-1){//b没有

                    new\_array[new\_number].table[i]=a.table[i];

                }

                else{//同一个变元互补,该析取式将化为1，无效消解，将进队重置

                    memset(new\_array[new\_number].table,-1,sizeof(new\_array[new\_number].table));

                    new\_array[new\_number].num=0;

                    return 0;

                }

            }

            else{//变元未出现

                continue;

            }

        }

        //消解完成，判断重复和空式

        if(new\_array[new\_number].num==0){//空式子,重置返回-1

            memset(new\_array[new\_number].table,-1,sizeof(new\_array[new\_number].table));

            new\_array[new\_number].num=0;

            return -1;

        }

        else{

            bool repeat=false;

            for(int i=0;i<old\_number;i++){//验重复

                if(same(new\_array[new\_number],old\_array[i])==true){

                    repeat=true;

                    break;

                }

            }

            if(repeat==false){//没重

                return 1;

            }

            else{//重了

                memset(new\_array[new\_number].table,-1,sizeof(new\_array[new\_number].table));

                new\_array[new\_number].num=0;

                return 0;

            }

        }

    }

}

//主函数区域

int main(){

    //1.初始化,获得原始式子及式子中存在的变量个数

    memset(origin,0,sizeof(origin));

    gets(origin);

    //2.拆分主合取式，获得各部分，并且构建队

    int old\_number=init(origin,old\_array);//记录队伍中有多少析取式

    //3.队伍构建完毕，开始循环

    int new\_number=0;

    do{

        if(new\_number!=0){//new\_array里有析取式，把其中所有的都放入old\_array里

            for(int i=0;i<new\_number;i++){

                old\_array[old\_number]=new\_array[i];

                memset(new\_array[i].table,-1,sizeof(new\_array[i].table));

                new\_array[i].num=0;

                old\_number++;

            }

            new\_number=0;

        }

        for(int i=0;i<old\_number-1;i++){

            for(int j=i+1;j<old\_number;j++){

                int status=cal(old\_array[i],old\_array[j],old\_number,new\_number);

                if(status==0){//未出现新的析取式

                    continue;

                }

                else if(status==-1){//出现空式子

                    printf("NO\n");

                    return 0;

                }

                else{//出现了新的析取式且非空式

                    new\_number++;

                }

            }

        }

    }while(new\_number!=0);//跳出条件，不再产生新的析取式

    printf("YES\n");

    return 0;

}

# 求关系的传递闭包

## **概述**

一次输入一个关系矩阵，使用Washell算法得出出该关系的传递闭包所对应的关系矩阵并输出。

## **步骤流程**

程序的主要步骤流程图如下。

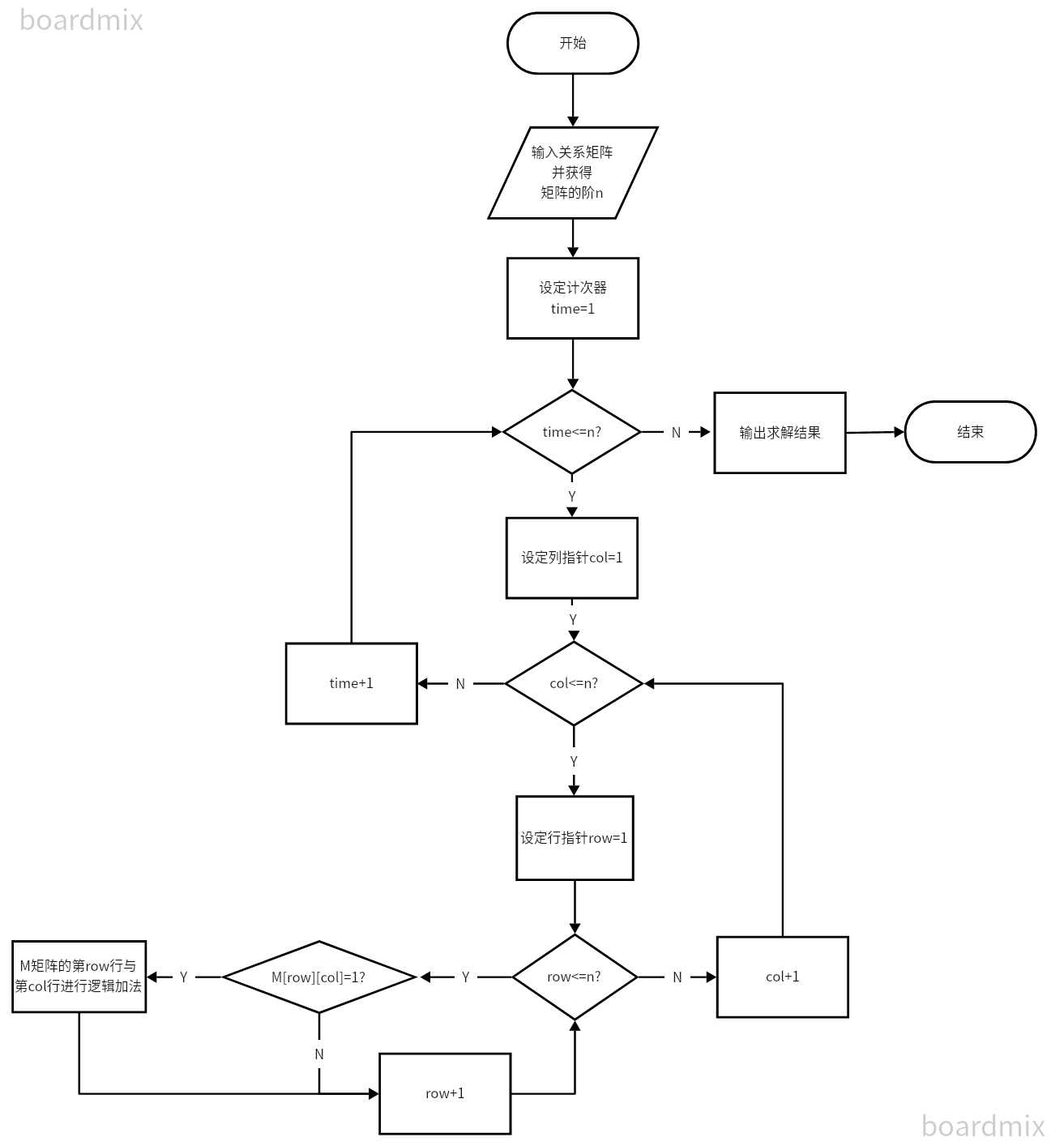


图 4 求关系的传递闭包

## **程序实现**

表 3求关系的传递闭包的函数表

| 序号 | 名称 | 说明 |
| --- | --- | --- |
| 1. | Logic\_add | 用于对矩阵的两行进行逻辑加法 |
| 2. | Get\_point | 用于通过获得矩阵一列代表的点的邻接点 |

### **Logic\_add函数**

该函数用来对矩阵中的两行的元素进行逻辑加法：即对应元素都为0时，结果为0；否则为1。

### **Get\_point函数**

该函数以矩阵的某列为参数，并对该列的每行元素进行遍历，若某行元素为1，则将该行指针存入队列待用。

## **最终代码**

#include <bits/stdc++.h>

/\*思路：

使用矩阵储存关系R的关系矩阵M

然后采用washell算法：

|->观察每一列，第i列中为1的元素<j,i>表示了可以从第j个点去到第i个点；

|  则我们将第j行的元素与第i行的元素进行逻辑加法，更新现在第j行的元素可以去到的点有哪些

|  更新了矩阵

|->重复，直到第n次矩阵更新完成

\*/

//全局变量域

int M[12][12]={0};//基数不超过十二的矩阵

int array[200]={0};//用来前期存数

std::queue<int> ral;//用来记录出发点的队列

//全局函数域

//1.两行的逻辑加法

void logic\_add(int ran,int back,int front){

    //ran为基数，back为结束点，front为出发点

    for(int i=0;i<ran;i++){

        if(M[front][i]==1||M[back][i]==1){//有1

            M[front][i]=1;

        }

        else{//无1

            M[front][i]=0;

        }

    }

    return;

}

//2.获得出发点函数

void get\_point(int ran,int back){

    while(ral.empty()!=true){//重置

        ral.pop();

    }

    for(int i=0;i<ran;i++){

        if(M[i][back]==1){//找到

            ral.push(i);

        }

        else{

            continue;

        }

    }

    return;

}

int main(){

    int num=0,total=0;

    while(scanf("%d",&num)!=EOF){

        array[total]=num;

        total++;

    }

    int ran=(int)sqrt(total);//基数

    num=0;

    for(int i=0;i<ran;i++){//写入矩阵

        for(int j=0;j<ran;j++){

            M[i][j]=array[num];

            num++;

        }

    }

    //开始进行循环更新矩阵了

    for(int time=1;time<=ran;time++){

        for(int back=0;back<ran;back++){

            get\_point(ran,back);

            while(ral.empty()!=true){//有出发点

                int front=ral.front();

                logic\_add(ran,back,front);

                ral.pop();

            }

        }

    }

    //正常输出即可

    for(int row=0;row<ran;row++){

        for(int col=0;col<ran;col++){

            if(col!=ran-1){

                printf("%d ",M[row][col]);

            }

            else{

                printf("%d\n",M[row][col]);

            }

        }

    }

}

# 求偏序关系的极小元和极大元

## **概述**

输入偏序集<A, ≤ >，输入的第一行给出A中的各个元素， 输入的第二行给出偏序关系，用有序对的形式给出。

输出该偏序集的极小元和极大元。

## **步骤梳理**

该程序的主要步骤流程图如下。

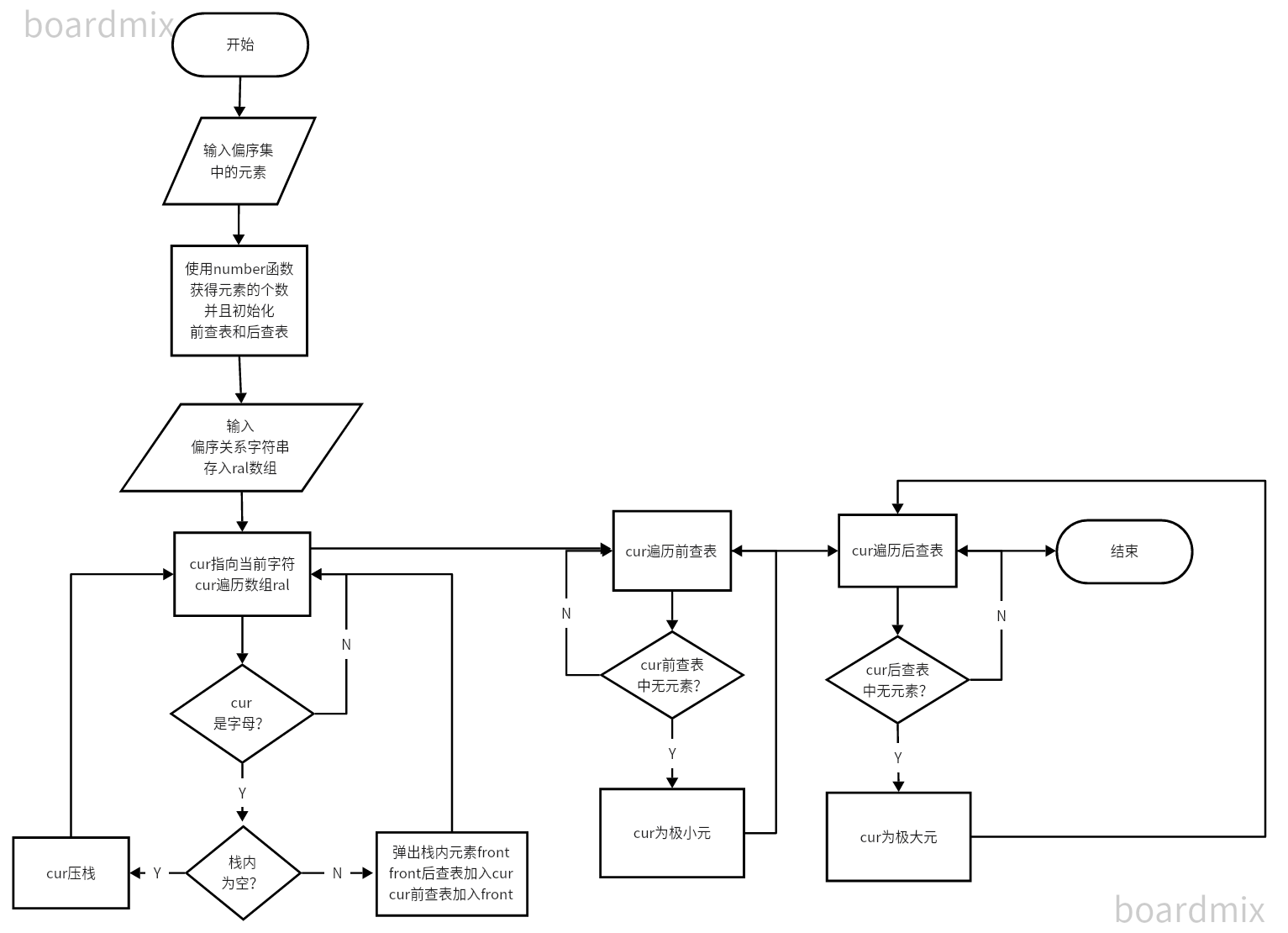


图 5 求极大元极小元的步骤流程图

## **程序实现**

表 4 求极大元极小元函数说明表

| 序号 | 名称 | 说明 |
| --- | --- | --- |
| 1. | Number | 获得偏序集元素个数并初始化 |
| 2. | deal | 处理偏序关系字符串 |

### **Number函数**

该函数对输入的偏序集进行遍历，获得偏序集中的元素个数，并初始化元素的前查表和后查表数组。

### **Deal函数**

该函数对输入的偏序关系进行遍历，当遍历到字母时，会先判断字母栈是否为空。若为空，则将字母压入栈顶。否则则将栈顶元素front弹出，front的后插表加入当前元素，当前元素的前查表加入front.

## **最终代码**

#include <bits/stdc++.h>

/\*思路:

   构建两个搜索表：前查表和后查表；

   |->前查表中记录该结点前面有什么点

   |->后查表记录该节点后面有什么点

   在判断极大元时，搜索后查表，找到后面没有点的点即为极大元

   在判断极小元时，搜索前查表，找到前面没有点的点即为极小元

\*/

//全局变量域

typedef struct node{

    char point;//点的名字

    int num;//点的后继数量

    char array[30];//点的后继表

};

node back[30];//后查表

node front[30];//前查表

//全局函数域

//1.获得点的数量和初始化

int number(char bas[]){

    int lenth=strlen(bas);

    int num=0;

    for(int i=0;i<lenth;i++){

        if(bas[i]>='a'&&bas[i]<='z'){

            num++;

            back[bas[i]-97].num=0;

            back[bas[i]-97].point=bas[i];

            front[bas[i]-97].num=0;

            front[bas[i]-97].point=bas[i];

        }

        else{

            continue;

        }

    }

    return num;

}

//2.处理关系函数

void deal(char ral[]){

    int lenth=strlen(ral);

    std::stack<char> freeze;//存关系的栈

    for(int i=0;i<lenth;i++){

        if(ral[i]>='a'&&ral[i]<='z'){//是字母

            if(freeze.empty()==true){//是前件

                freeze.push(ral[i]);

            }

            else{//是后件

                freeze.push(ral[i]);

                int after=(int)freeze.top()-97;

                freeze.pop();

                int before=(int)freeze.top()-97;

                freeze.pop();

                //开始写表

                front[after].array[front[after].num]=before;//前查表

                front[after].num++;

                back[before].array[back[before].num]=after;//后查表

                back[before].num++;

            }

        }

        else{

            continue;

        }

    }

    return;

}

int main(){

    char bas[50]={0};

    gets(bas);

    int num=number(bas);

    char ral[100]={0};

    gets(ral);

    deal(ral);

    //遍历两表即可

    std::queue<char> mini;//极小元

    std::queue<char> maxi;//极大元

    for(int i=0;i<num;i++){

        if(front[i].num==0){

            mini.push(front[i].point);

        }

        if(back[i].num==0){

            maxi.push(back[i].point);

        }

    }

    while(mini.empty()!=true){

        char cp=mini.front();

        mini.pop();

        printf("%c",cp);

        if(mini.empty()==true){

            printf("\n");

        }

        else{

            printf(",");

        }

    }

    while(maxi.empty()!=true){

        char cp=maxi.front();

        maxi.pop();

        printf("%c",cp);

        if(maxi.empty()==true){

            printf("\n");

        }

        else{

            printf(",");

        }

    }

}

# 代数系统算律的判断

## **概述**

给定一个只含一个二元运算的代数系统，判断该代数系统是否符合交换律、结合律、分配律，是否有幺元、零元？

## 步骤流程

该程序的主要步骤流程图如下。

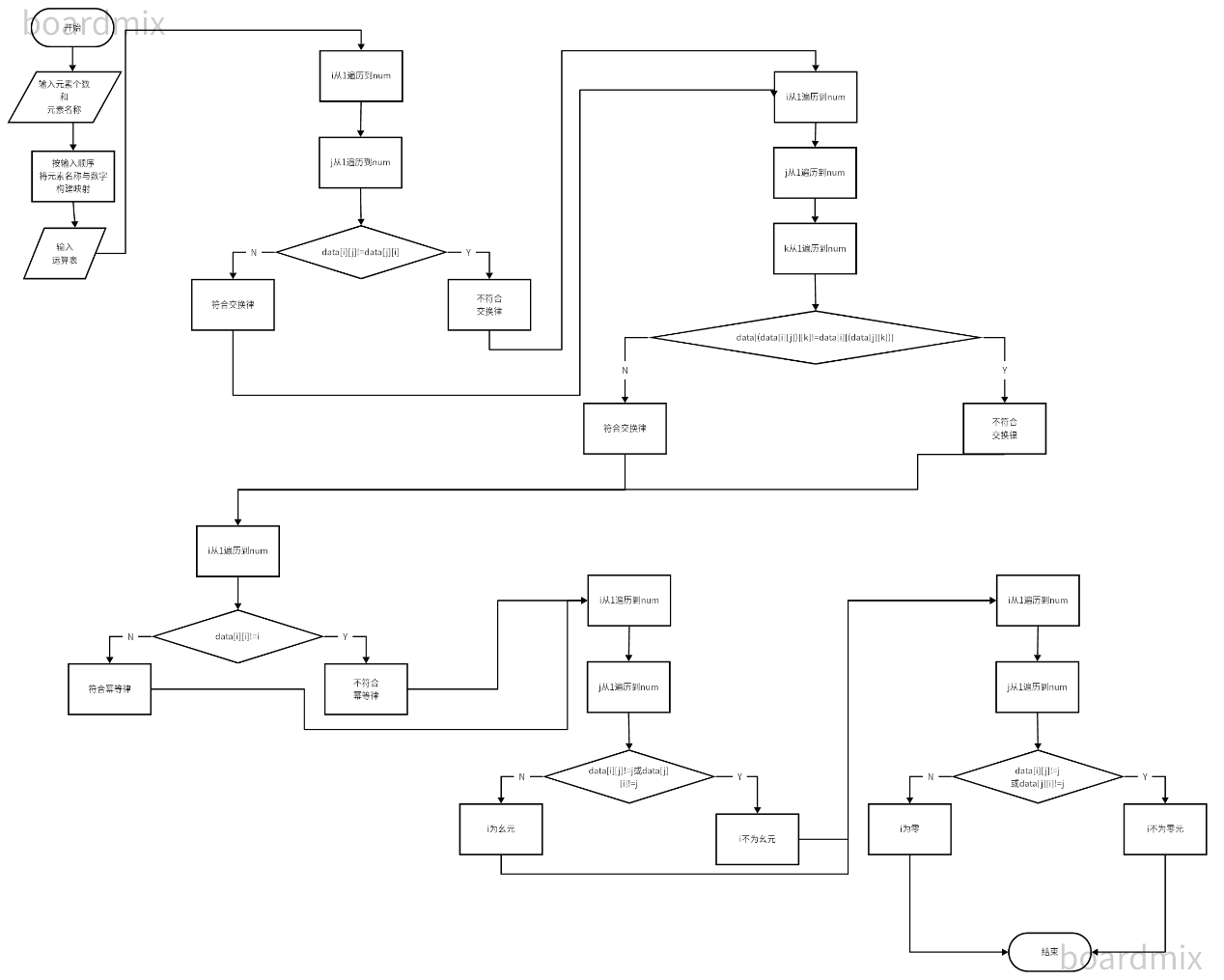


图 6 代数系统算律的判断流程

## **程序实现**

表 5 代数系统的算律判断函数表

| 序号 | 名称 | 说明 |
| --- | --- | --- |
| 1. | isCommutative | 用于判断是否符合交换律 |
| 2. | isAssociative | 用于判断是否符合结合律 |
| 3 | isIdempotent | 用于判断是否符合幂等律 |
| 4 | Identify | 用于检查是否含有幺元 |
| 5 | Zero | 用于检查是否还有零元 |

### **isCommutative函数**

由定义，两层循环，外层指针i遍历行，内层j遍历列，判断是否有data[i][j]!=data[j][i];如果有，则不符合交换律；否则符合。

### **isAssociative函数**

由定义，三层循环，最外层指针i、中层指针j、最内层指针k从1遍历到num,判断是否有

data[data[i][j]][k]!=data[i][data[j][k]];如果有，则不符合结合律；否则符合。

### **isIdempotent函数**

由定义，单层循环，指针i从1遍历到num，判断是否有data[i][i]!=i;如果有，则不符合幂等律，否则符合。

### **identify函数**

由定义，两层循环，外层指针i遍历1到num，内层j遍历1到num，判断是否有data[i][j]!=j||data[j][i]!=j;如果没有有，则i为幺元；否则i不为幺元。

### **zero函数**

由定义，两层循环，外层指针i遍历1到num，内层j遍历1到num，判断是否有data[i][j]!=i||data[j][i]!=i;如果没有，则i为零元；否则i不为零元。

## 最终代码

#include <bits/stdc++.h>

using namespace std;

/\*

基本思路：

检查元素：若为字母的，对应数字；使用二维表储存运算表

1.交换律

按定义，a[i][j]=a[j][i]

2.幂等律

按定义，a[i][i]=i

3.结合律

按定义，a[(a[i][j])][k]=a[i][(a[j][k])]

4.幺元

按定义，存在元素i，使得第i行与第i列都为另一个元素

5.零元

按定义，存在元素i，使得第i行与第i列都为i

\*/

//全局变量域

#define M 128

int data[M][M];//储存运算表

int trans[M];//用来将字符转化为数字的映射表

//全局函数域

void isCommutative(int num){//交换律

    for(int i=0;i<num;i++){

        for(int j=i+1;j<num;j++){

            if(data[i][j]!=data[j][i]){

                printf("commutative law:n\n");

                return;

            }

        }

    }

    printf("commutative law:y\n");

    return;

}

void isAssociative(int num){//结合律

    for(int i=0;i<num;i++){

        for(int j=0;j<num;j++){

            for(int k=0;k<num;k++){

                if(data[data[i][j]][k]!=data[i][data[j][k]]){

                    printf("associative law:n\n");

                    return;

                }

            }

        }

    }

    printf("associative law:y\n");

    return;

}

void isIdempotent(int num){//幂等律

    for(int i=0;i<num;i++){

        if(data[i][i]!=i){

            printf("idempotent law:n\n");

            return;

        }

    }

    printf("idempotent law:y\n");

    return;

}

void identity(int num){//幺元

    for(int i=0;i<num;i++){

        bool status=true;

        for(int j=0;j<num;j++){

            if(data[i][j]!=j||data[j][i]!=j){

                status=false;

                break;

            }

        }

        if(status==true){

            char ch='0';

            for(int t=0;t<M;t++){

                if(trans[t]==i){

                    ch=(char)t;

                    break;

                }

            }

            printf("identity element:%c\n",ch);

            return;

        }

    }

    printf("identity element:n\n");

}

void zero(int num){//零元

    for(int i=0;i<num;i++){

        bool status=true;

        for(int j=0;j<num;j++){

            if(data[i][j]!=i||data[j][i]!=i){

                status=false;

                break;

            }

        }

        if(status==true){

            char ch='0';

            for(int t=0;t<M;t++){

                if(trans[t]==i){

                    ch=(char)t;

                    break;

                }

            }

            printf("zero element:%c\n",ch);

            return;

        }

    }

    printf("zero element:n\n");

}

int main(){

    memset(data,-1,sizeof(data));

    memset(trans,-1,sizeof(trans));

    int num=0;

    scanf("%d",&num);

    getchar();

    char ch='0';

    for(int i=0;i<num;i++){

        scanf("%c",&ch);

        trans[(int)ch]=i;//按录入顺序构建映射

        getchar();

    }

    //录入运算表

    for(int row=0;row<num;row++){

        for(int col=0;col<num;col++){

            scanf("%c",&ch);

            data[row][col]=trans[(int)ch];

            getchar();

        }

    }

    //接下来使用函数；

    isCommutative(num);

    isAssociative(num);

    isIdempotent(num);

    identity(num);

    zero(num);

    return 0;

}

# 模n加群的元素的阶

## **概述**

设Zn为模n整数加群，求Zn中元素的阶。

## **步骤流程**

该程序的主要步骤流程如下。

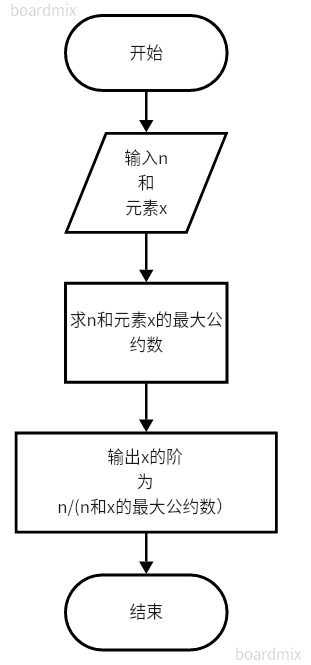


图 7 求Zn中元素的阶的主要流程图

## 程序实现

表 6 求Zn中元素的阶的函数说明表

| 序号 | 名称 | 说明 |
| --- | --- | --- |
| 1. | mc | 获得两个数的最大公约数 |

### mc函数

使用辗转相除法，获得两个数的最大公约数。

## 最终代码

#include <bits/stdc++.h>

using namespace std;

//基本思路：

//在模n加群里，元素的阶=元素/（元素与n的最大公约数）、

int mc(int small,int big){//辗转相处出最大公约数

    while(big%small!=0){

        int temp=big%small;

        big=small;

        small=temp;

    }

    return small;

}

int main(){

    int num;

    int x;

    scanf("%d,%d",&num,&x);

    if(x==0){

        printf("1\n");

    }

    else{

        printf("%d\n",num/mc(x,num));

    }

}

# 二部图的判定

## **概述**

给定无向图的邻接矩阵，判断无向图G是否为二部图。

## **步骤流程**

该程序的主要步骤流程图如下。

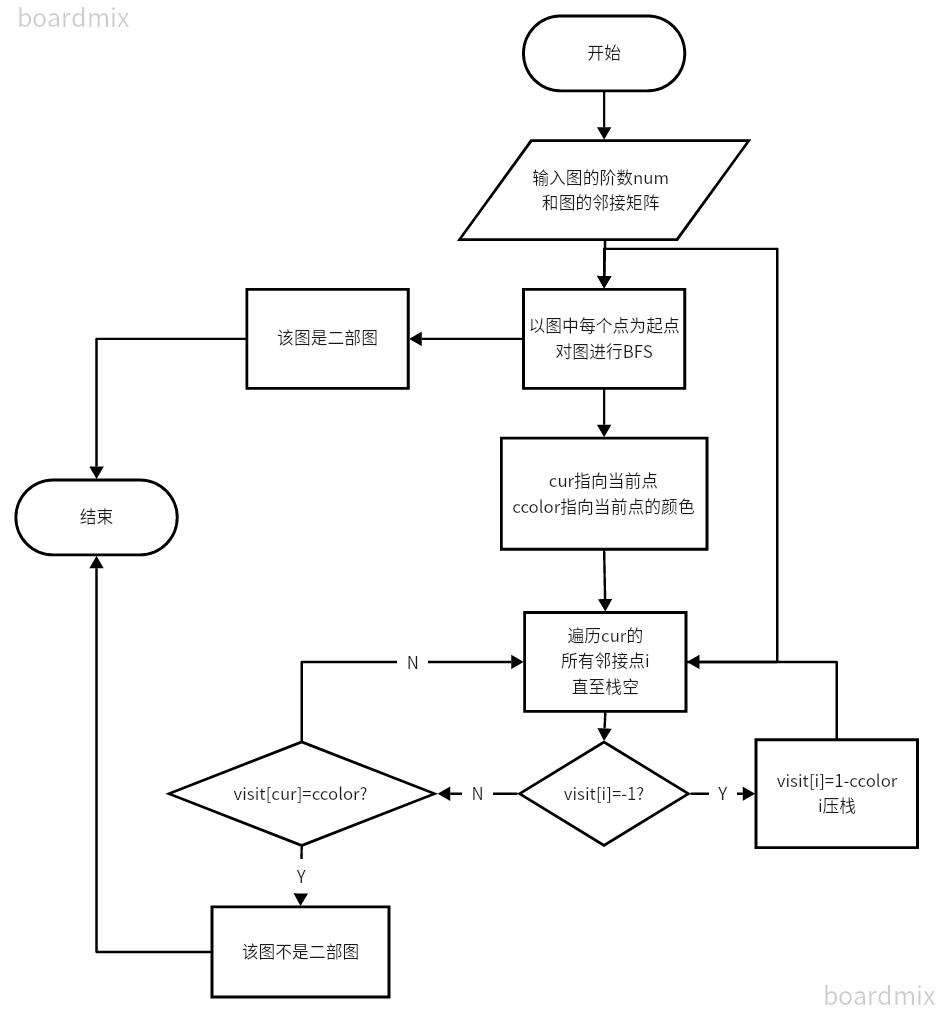


图 8 二部图判定的主要流程图

## **程序实现**

表 7 二部图判定函数说明表

| 序号 | 名称 | 说明 |
| --- | --- | --- |
| 1. | BFS | 以图中每个点为起点进行BFS的函数 |

### BFS函数

该函数是使用着色法，每次以图中的一个点为出发点，对图进行BFS染色的函数。遍历过程中，如果染色遇到冲突，则该图不是二部图，返回false;否则该图即为二部图，返回true；

## **最终代码**

#include <bits/stdc++.h>

using namespace std;

/\*

基本思路：

使用着色法，BFS对整个图进行遍历

着色过程中，将颜色存在visit数组中：0,1为两种颜色，-1表示未访问

\*/

//全局变量域

#define M 100

int data[M][M]={0};

int visit[M];

queue <int> team;//储存邻接点的队列

//全局函数域

bool BFS(int num){//bfs函数

    for(int point=0;point<num;point++){//从每个点开始进行BFS

        memset(visit,-1,sizeof(visit));//访问数组重置

        team.push(point);

        visit[point]=0;

        while(team.empty()!=true){//非空时

            int cur=team.front();

            int ccolor=visit[cur];

            for(int i=0;i<num;i++){

                if(data[cur][i]==0){

                    continue;

                }

                else{//有从cur到i的路径

                    if(visit[i]==-1){

                        visit[i]=1-ccolor;

                        team.push(i);

                    }

                    else{

                        if(visit[i]==ccolor){

                            return false;

                        }

                        else{

                            continue;

                        }

                    }

                }

            }

            team.pop();

        }

    }

    return true;

}

int main(){

    //信息录入

    int num=0;

    scanf("%d",&num);

    for(int i=0;i<num;i++){

        for(int j=0;j<num;j++){

            scanf("%d",&data[i][j]);

        }

    }

    //BFS处理

    bool status=BFS(num);

    if(status==true){

        printf("yes\n");

    }

    else{

        printf("no\n");

    }

}

# 有向图的判定

## **概述**

给定有向连通图的邻接矩阵，判断该连通图是强连通图、单项连通图或弱连通图。

## **步骤流程**

程序的主要步骤流程图如下。

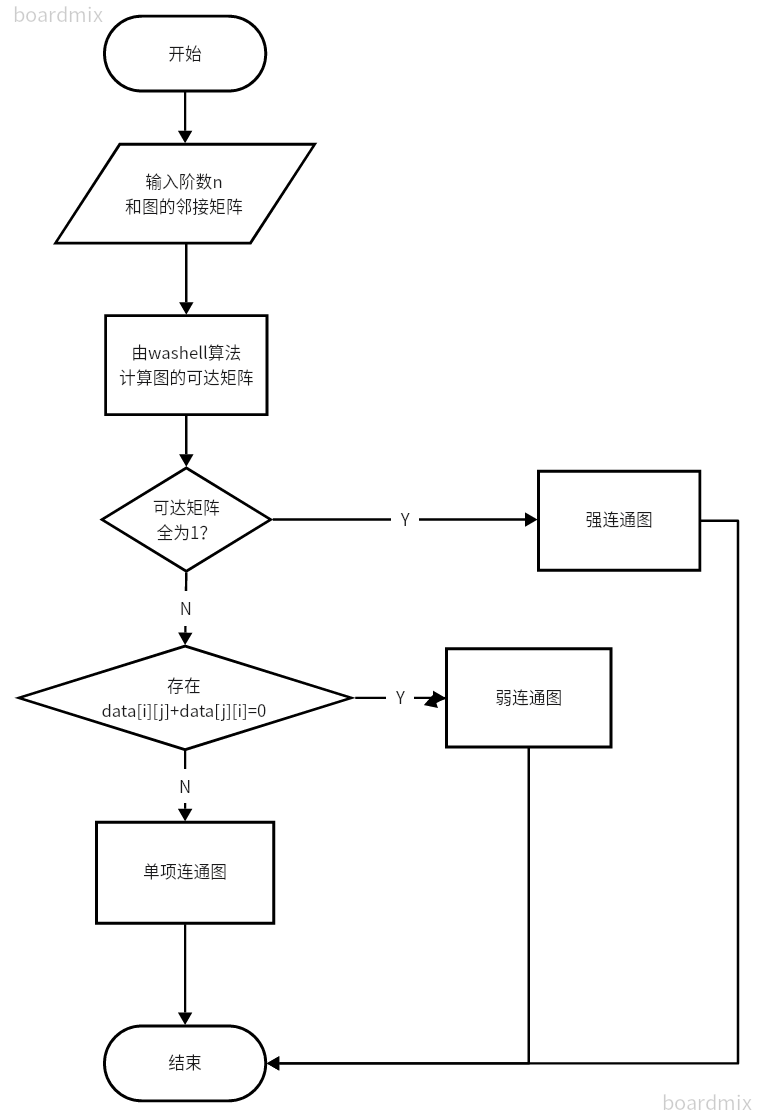


图 9 有向图的判定的主要流程图

## **程序实现**

表 8 有向图的判定函数说明表

| 序号 | 名称 | 说明 |
| --- | --- | --- |
| 1. | Logicadd | 用于对矩阵的两行进行逻辑加法 |
| 2. | Washell | 用于通过washell算法获得可达矩阵 |

### Logicadd函数

该函数用来对矩阵中的两行的元素进行逻辑加法：即对应元素都为0时，结果为0；否则为1。

### Washell函数

该函数通过washell算法，动态更新图的可达矩阵。

## **最终代码**

#include <bits/stdc++.h>

using namespace std;

/\*

基本思路：

1.使用washell算法构建可达矩阵

2.对可达矩阵进行判断

全为1，强连通图；p[i][j]+p[j][i]=1,单项连通图；否则为弱连通图

\*/

//全局变量域

#define M 100

int data[M][M]={0};

//全局函数域

void Logicadd(int num,int add,int added){

    for(int i=0;i<num;i++){

        if(data[add][i]==1||data[added][i]==1){

            data[added][i]=1;

        }

        else{

            data[added][i]=0;

        }

    }

    return;

}

void washell(int num){//washell算法构建可达矩阵

    queue <int> back;

    for(int time=1;time<=num;time++){

        for(int col=0;col<num;col++){

            for(int row=0;row<num;row++){

                if(data[row][col]==1){

                   back.push(row);

                }

                else{

                    continue;

                }

            }

            while(back.empty()!=true){

                int tp=back.front();

                Logicadd(num,col,tp);

                back.pop();

            }

        }

    }

    for(int i=0;i<num;i++){//认为自可达

        data[i][i]=1;

    }

    return;

}

int main(){

    int num;

    scanf("%d",&num);

    for(int i=0;i<num;i++){

        for(int j=0;j<num;j++){

            scanf("%d",&data[i][j]);

        }

    }

    washell(num);

    bool str=true;

    bool bad=false;

    for(int i=0;i<num;i++){

        for(int j=0;j<num;j++){

            if(data[i][j]+data[j][i]==0){

                str=false;

                bad=true;

            }

            else if(data[i][j]+data[j][i]==1){

                str=false;

            }

        }

    }

    if(str==true){

        printf("A\n");

    }

    else{

        if(bad==true){

            printf("C\n");

        }

        else{

            printf("B\n");

        }

    }

}