**《离散数学》命题逻辑联结词、真值表、主范式项目文档**

1. **题目简介**

**1.1逻辑联接词的运算**

从键盘输入两个命题变元P和Q的真值，利用C++语言实现二元合取、析取、条件和双向条件表达式的计算；充分利用联结词和逻辑运算符之间的相似性来实现程序功能。

**1.2求任意一个命题公式的真值表**

求任意一个命题公式的真值表，并根据真值表求主范式。本实验要求利用C/C＋＋语言，实现任意输入公式的真值表计算。一般将公式中的命题变元放在真值表的左边，将公式的结果放在真值表的右边。命题变元可用数值变量表示，合式公式的表示及求真值表转化为逻辑运算结果；可用一维数表示合式公式中所出现的n个命题变元，同时它也是一个二进制加法器的模拟器，每当在这个模拟器中产生一个二进制数时，就相当于给各个命题变元产生了一组真值指派。算法逻辑如下：

（1）将二进制加法模拟器赋初值0。

（2）计算模拟器中所对应的一组真值指派下合式公式的真值。

（3）输出真值表中对应于模拟器所给出的一组真值指派及这组真值指派所对应的一行真值。

（4）产生下一个二进制数值，若该数值等于2n-1，则结束，否则转（2）。

1. **解题思路**

**2.1实验原理**

**2.1.1 A题**

首先分析联结词和逻辑运算符之间的相似性：

1. 合取：P∧Q 在逻辑运算符当中相当于P&&Q ；
2. 析取：P∨Q 在逻辑运算符当中相当于P||Q；
3. 条件：因为P→Q = ┓P∨Q，所以在逻辑运算符当中相当于！P||Q；
4. 双向条件：因为P←→Q = (┓P∨Q)∧(┓Q∨P),所以在逻辑运算符中相当于（！P||Q）&&（！Q||P）；

**2.1.2 B题**

真值表:表征逻辑事件输入和输出之间全部可能状态的表格。列出命题公式真假值的表。通常以1表示真，0 表示假。命题公式的取值由组成命题公式的命题变元的取值和命题联结词决定，命题联结词的真值表给出了真假值的算法。

**2.1.3 C题**

主析取范式：在含有n个命题变元的简单合取式中,若每个命题变元与其否定不同时存在,而两者之一出现一次且仅出现一次,则称该简单合取式为极小项。由若干个不同的极小项组成的析取式称为主析取范式;

主合取范式：在含有n个命题变元的简单析取式中，若每个命题变元与其否定不同时存在，而两者之一出现一次且仅出现一次，称该简单析取式为极大项。由若干个不同的极大项组成的合取式称为主合取范式；

**2.2设计程序**

**2.1.1 对于A题：**

1. 对各个输入量进行处理，判断输入是否为0或1，进行错误处理；
2. 将正确的输入进行运算并输出结果；
3. 判断是否继续或退出；

**2.1.2 对于B、C题：**

首先判断输入的是一个合理的式子，在公式的最后面加上#表示结束。该字符优先级最低;

然后从式子中查找出变量的个数，开辟一个二进制函数，用来生成真值表，然后用函数运算，输出结果，并根据结果归类给范式，最后输出范式。

1. **核心算法**

将联结词转换为逻辑运算符进行运算：

cout << " 合取:" << endl << " P/" << '\\' << "Q=" << (p && q) << endl;

cout << " 析取:" << endl << " P" << '\\' << "/Q=" << (p || q) << endl;

cout << " 条件:" << endl << " P->Q=" << (!p || q) << endl;

cout << " 双条件:" << endl << " P<->Q=" << ((!p || q) && (!q || p))<< endl;

**3.1循环嵌套算法实现**

for (int i = 0; i < pow2(proposition\_set.size()); i++)

{

Map\_ii bina\_set = toBinary(proposition\_set.size(), i);

for (unsigned int j = 0; j < bina\_set.size(); j++)

{

cout << bina\_set[j] << "\t";

}

int result = calculate(formula, proposition\_set, bina\_set);

\*(m+i) = result;

cout << result << endl;

}

**3.1循环嵌套算法描述**

循环嵌套，外层循环生成2的n次方个二进制数，从0-2^n-1；内层循环通过调用toBinary函数将每个十进制数转换为二进制数；

通过for循环正向打印每个二进制数组；将公式和命题变项以及二进制赋值传入calculate函数当中，得出返回值为将命题变项的值代入求得的公式结果，输出；

**3.2 getProposition函数\_提取公式中命题变项算法实现**

Map\_ic getProposition(string formula){

Map\_ic proposition;

int n\_proposition = 0;

for (unsigned int i = 0; i < formula.length(); i++)

{

char c = formula[i];

if ((c >= 'a' && c <= 'z') || (c >= 'A' && c <= 'Z'))

{

//遍历所有命题变项

int r = findProposition(proposition, c);

if (r == -1)

{

//说明该命题变项尚未被遍历过

proposition[n\_proposition] = c;

n\_proposition++;

}

}

else if (!priority.count(c))

{

cout << c << " is undefined!" << endl;

exit(2);

}

}

return proposition;

}

**3.2 getProposition函数\_提取公式中命题变项算法描述**

Proposition（命题变量），传入参数公式字符串，进行遍历，如果字符在小写字母a~z或大写字母A~Z之间，先通过findProposition函数进行判断该变量是否已经存在，若不存在，将其放入Proposition中，并将n\_proposition++;如果字符不在范围之内，利用count算法到priority中进行搜索，如果返回值==0，则说明也不是联结词，进行错误处理，直接退出；循环遍历结束返回命题变量的map;

**3.3 findProposition函数\_查找命题变项是否存在算法实现**

int findProposition(Map\_ic pSet, char p) {

Map\_ic::iterator it = pSet.begin();

while (it != pSet.end())

{

if (it->second == p)

{

return it->first;

}

it++;

}

return -1;

}

**3.3 findProposition函数\_查找命题变项是否存在算法描述**

传入参数pSet(存放命题变量），p（字符），定义一个迭代器，从map的第一个元素开始遍历，如果变量存在，返回该变量存放的位置，如果不匹配继续遍历，直到迭代器指到end()，返回-1；

**3.4 toBinary函数\_命题变项二进制赋值算法实现**

Map\_ii toBinary(int n\_proposition, int index) {

Map\_ii result;

for (int i = 0; i < n\_proposition; i++)

{

int r = index % 2;

result[n\_proposition - 1 - i] = r;

index = index / 2;

}

return result;

}

**3.4 toBinary函数\_命题变项二进制赋值算法描述**

参数设置为命题变项的个数和十进制数，从右边往左边赋值，返回index十进制数对应的二进制数；

**3.5 pow2函数\_返回指定2的n次方值算法实现**

int pow2(int n) //该函数返回指定数字的二次方的值

{

if (n == 0)

return 1;

else

return 2 \* pow2(n - 1);

}

**3.5 pow2函数\_返回指定2的n次方值算法描述**

采用尾递归的方法,递归计算2的n次方，即n个2相乘。

**3.5 calculate函数\_求解命题变项运算结果过程算法实现**

int calculate(string formula, Map\_ic pSet, Map\_ii value) {

stack<char> opter;

stack<int> pvalue;

opter.push('#');

formula = formula + "#";

for (unsigned int i = 0; i < formula.length(); i++)

{

char c = formula[i];

if ((c >= 'a' && c <= 'z') || (c >= 'A' && c <= 'Z'))

{

pvalue.push(value[findProposition(pSet, c)]);

}

else

{

//此时遍历的是运算符

char tmp = opter.top();

if (priority[tmp] > priority[c])

{

while (priority[tmp] > priority[c] && tmp != '(')

{

check(pvalue, opter);

tmp = opter.top();

if (tmp == '#' && c == '#')

{

return pvalue.top();

}

}

opter.push(c);

}

else

opter.push(c);

}

}

return -1;

}

**3.5 calculate函数\_求解命题变项运算结果过程算法描述**

在公式的后面加上#，以命题公式开始遍历，遍历到命题变项时,参数value为bina\_set二进制赋值，pvalue.push(value[findProposition(pSet, c)]);把命题变项c放入proposition中找到他们的位置，然后以他们的位置去找value对应的值，将该变项的赋值放入pvalue栈中；

遍历到运算符时，拿要进栈的运算符和栈顶的运算符进行比较，如果进栈的运算符优先级低，特例：如果栈顶运算符为（，则继续放，如果不是左括号就就将栈顶运算符运算掉，value栈将变量值pop，push进结果，如果现在栈顶运算符为#，且要进栈的符号也为#，则证明运算完毕，返回结果（pvalue.top());

如果没有#，则将此运算符push进去；

**3.6 check函数\_单步运算求值算法实现**

void check(stack<int>& value, stack<char>& opter) {

int p, q, result;

char opt = opter.top();

switch (opt)

{

case '&':

p = value.top();

value.pop();

q = value.top();

value.pop();

result = p && q;

value.push(result);

opter.pop();

break;

case '|':

p = value.top();

value.pop();

q = value.top();

value.pop();

result = p || q;

value.push(result);

opter.pop();

break;

case '!':

p = value.top();

value.pop();

result = !p;

value.push(result);

opter.pop();

break;

case '^':

q = value.top();

value.pop();

p = value.top();

value.pop();

result = !p || q;

value.push(result);

opter.pop();

break;

case '~':

p = value.top();

value.pop();

q = value.top();

value.pop();

result = (!p || q) && (p || !q);

value.push(result);

opter.pop();

break;

case '#':

break;

case '(':

break;

case ')':

opter.pop();

while (opter.top() != '(')

{

check(value, opter);

}

if (opter.top() == '(')

{

opter.pop();

}

break;

default:

break;

}

}

**3.6 check函数\_单步运算求值算法描述**

该函数传入形式参数为运算符栈opter和命题变项对应运算值栈value，该函数实现运算符栈opter栈顶运算符的运算，主要实现如下：

取运算符栈栈顶运算符opt，根据opt不同的值进行运算：如果opt为二元运算符&，|，^,~,则从运算值栈value中取出栈顶两个元素进行对应运算，如果opt为一元运算符！，则从运算值栈value中取出一个栈顶元素进行取反运算，运算完成后将结果result压入运算值栈顶中，并使栈顶运算符出栈。

1. **所用数据结构**

使用map图存储：

1、优先级**priority**，给涉及到的联结词从大到小赋予优先级，数值越大优先级越高。

**2、proposition ,**存放公式中出现的命题变项

3、result：存放命题变项的二进制赋值结果

使用stack栈存储：

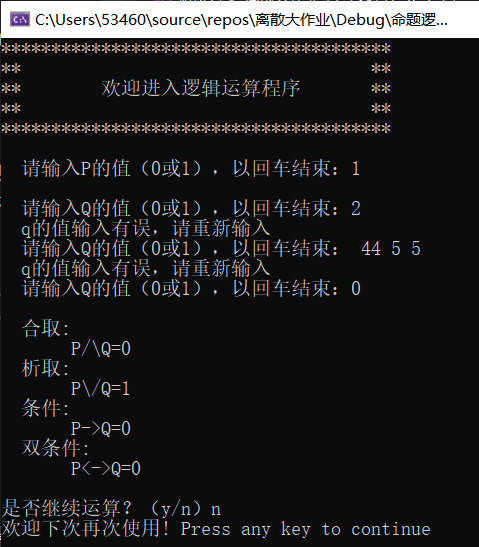
1、opter:用来放联结词

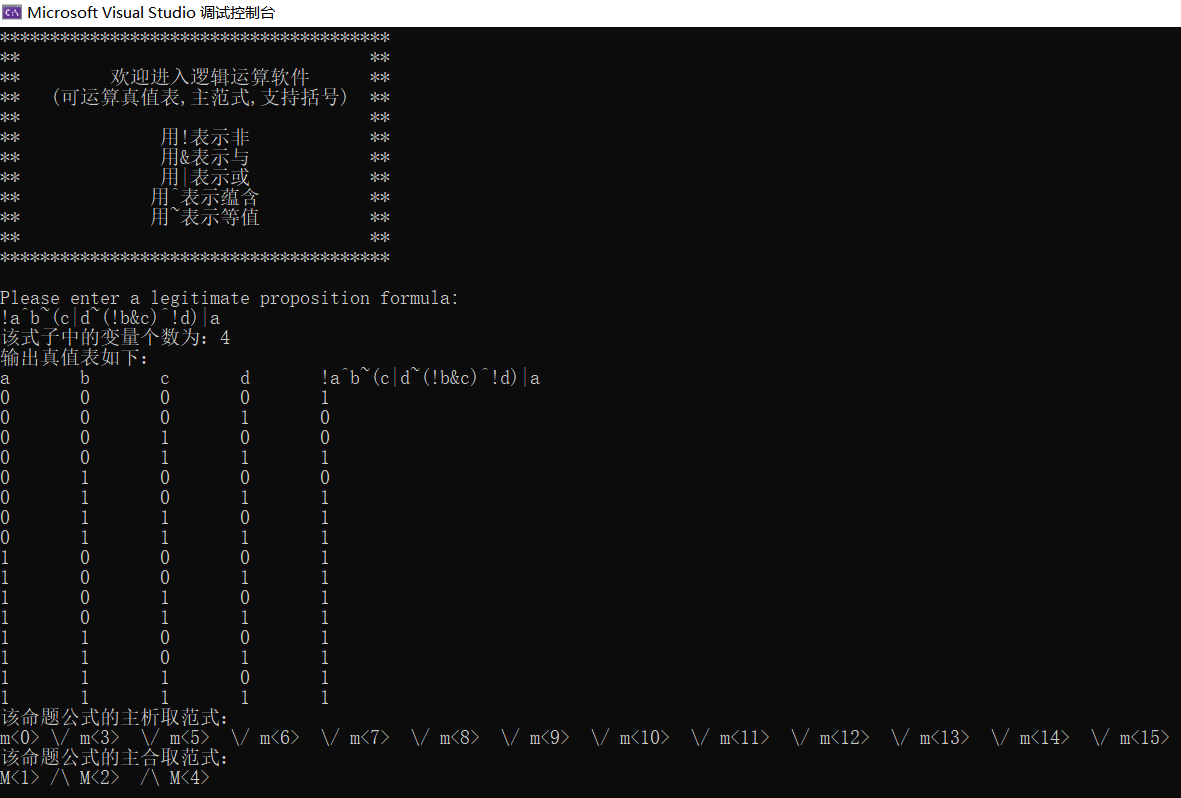
2、pvalue:用来放运算结果

使用数组：

M：malloc一段连续空间存放结果。动态开辟数组

1. **实验结果**





1. **健壮性处理**

//若输入的运算式没有命题变项时，直接退出程序；

if (proposition\_set.size() == 0)

{

cout << "输入的公式不含命题变项！"

return 0;

}

//公式中不存在合法运算符

if (formula.length() == proposition\_set.size())

{

cout << "输入的公式不含运算符！" << endl;

}

当左右括号不匹配的时候，右括号会一直递归循环，程序崩溃，所以在check函数里进行检查，如果没找到左括号但已经找到了栈底元素#，则直接退出程序。

case ')':

opter.pop();

while (opter.top() != '(')

{

check(value, opter);

if (opter.top() == '#')

{

cout << "\n左右括号不匹配\n";

exit(1);

}

}

if (opter.top() == '(')

{

opter.pop();

}

break;

1. **心得体会**

通过修改C语言的源代码发现其中含有一些逻辑错误，查找逻辑运算符的优先级和联结词的优先级进行对比:

联结词优先级：（） > ! > ∧ > ∨ > → > ←→

逻辑运算符的优先级：（） > ！ > && > ||

所以根据优先级有些地方不用加括号，有些地方必须加括号，这是在逻辑运算时要注意的问题。

同时，要进行必要的错误处理，增强程序的健壮性。

而B、C题较难，使用到map等数据结构做起来更简便一些，通过阅读源代码进行学习其逻辑思维，收获了很多，同时也将其不足的地方，比如代码的健壮性不强。还有主析取范式中，极小项之间应该用\/相连，主合取范式中，极大项之间用/\相连。

本次实验让我将联结词和逻辑运算符联系起来，将数学逻辑正确地应用的程序设计当中。