离散数学

项目说明文档

命题逻辑真值表、主范式

作者姓名: 高逸轩

学　　号: 2053385

指导教师: 唐剑锋

学院专业: 软件学院 软件工程

同济大学

Tongji University

1 功能简介

1.1 题目要求

根据用户给出的一个逻辑运算表达式，计算得到逻辑表达式的真值表和主析取范式、主合取范式。

1.2 项目需求分析

本项目在实现的过程中，考虑并且满足了以下的需求：

- ✓ 代码可读性高

  本项目在实现过程中，将代码根据功能的不同划分为了不同的代码块，便于阅读。

1.3 项目要求

首先由用户输入一个逻辑运算表达式（需保证输入正确，其中用!表示非 用&表示与 用|表示或 用^表示蕴含 用~表示等值，括号采用英文括号，命题逻辑变量名称大小写敏感），然后得到其真值表和主析取范式、主合取范式。

1.3.1 功能要求

可以根据用户输入的式子首先统计得到变量名称和总数量，然后对逻辑表达式中所有的变量进行不同的赋值，并计算出各种情况下的表达式的值，同时统计得到主析取范式和主合取范式的各种可能，最终输出在屏幕上。

1.3.2 输入格式

一个逻辑运算表达式（需保证输入正确，其中用!表示非 用&表示与 用|表示或 用^表示蕴含 用~表示等值，括号采用英文括号，命题逻辑变量名称大小写敏感）

1.3.3 项目简单示例

```
请输入一个合法的命题公式（命题变元名称大小写敏感，请使用英文括号）：
!a&(!a~b|c^b)

该式子中的变量个数为:3

输出真值表如下:

  a   b   c   !a&(!a~b|c^b)

  0   0   0   1

  0   0   1   0

  0   1   0   1

  0   1   1   1

  1   0   0   0

  1   0   1   0

  1   1   0   0

  1   1   1   0


该命题公式的主合取范式:
        M(1)\/M(4)\/M(5)\/M(6)\/M(7)

该命题公式的主析取范式:
        M(0)/\M(2)/\M(3)
```

2 项目实现

本项目核心部分：

- ✓ 从表达式中统计得到所有变量
- ✓ 对变量进行所有情况的赋值
- ✓ 计算某种赋值情况下的表达式的值
- ✓ 实现主范式的统计

下面将对本项目核心的部分类进行介绍，部分代码如下：

```
// 命题变量单元
struct variable
{
    char name;  // 变量名称
    bool value; // 变量值
};
```

```
class TruthTable
{
public:
    // 构造函数，根据输入的表达式s构建
    TruthTable(const string s = "");

    // 析构函数，释放内存，避免内存丢失
    ~TruthTable() { delete[]variables, delete[]principalConjunctiveNormalForm, delete[]principalDisjunctiveNormalForm; }

    // 输出真值表
    void PrintTruthTable();

    // 输出主范式
    void PrintPrincipalNormForms();

private:

    int* principalDisjunctiveNormalForm;// 主析取范式
    int* principalConjunctiveNormalForm;// 主合取范式
    int variableNumber = 0;             // 变量总数
    variable* variables;                // 存储变量的名称和其对应赋值
    string expression;                  // 表达式

    // 将十进制数字num转换为二进制数字，同时将二进制数字按位赋值给variable变量
    void SetVariablesValue(int num = 0);

    // 计算当前状态下表达式的值
    bool CalculateExpressionValue();

    // 输出全部变量与表达式名称
    void PrintVariablesName()const;

    // 输出全部变量与表达式的值
    void PrintVariablesValue();

    // 将表达式中的变量名转换为数字
    string AlphaToNum();

};
```

在命题变量的记录与赋值中，我开始的思路是采用 C++STL 库的 pair 工具构造<char,bool>的形式来记录名称与其赋值，但由于本次作业不能使用 STL 工具，故采用了自己编写 variable 的方式来实现相通的功能，以方便后续计算。

以下内容按照解题顺序排列，请按照顺序阅读：

2.1 运算原理

在离散数学理论课上，我们学习了以下知识引出了真值表和主范式的概念：

设 p_1，p_2，…，p_n 是出现在公式 A 中全部的命题变项，给 p_1，p_2，…，p_n 指定一组真值，称为对 A 的一个赋值或解释. 使公式为真的赋值称作成真赋值，使公式为假的赋值称作成假赋值。

真值表：命题公式在所有可能的赋值下的取值的列表，其含 n 个变项的公式有 2^n 个赋值。

析取范式:由有限个简单合取式组成的析取式

$A_1 \vee A_2 \vee \ldots \vee A_r$，其中 $A_1, A_2, \ldots, A_r$ 是简单合取式

合取范式:由有限个简单析取式组成的合取式

$A_1 \wedge A_2 \wedge \ldots \wedge A_r$，其中 $A_1, A_2, \ldots, A_r$ 是简单析取式

在含有 n 个命题变项的简单合取式(简单析取式)中,若每个命题变项均以文字的形式出现且仅出现一次，而且第 $i(1 \le i \le n)$ 个文字(按下标或字母顺序排列)出现在左起第 i 位上,称这样的简单合取式(简单析取式)为极小项(极大项)。

主析取范式:由极小项构成的析取范式

主合取范式:由极大项构成的合取范式

2.2 从表达式中统计得到所有变量

我们为 TruthTable 类创立了一个根据表达式字符串建立的构造函数，在其中，我通过遍历表达式中的非运算符的字符，实现了命题逻辑变量的创建和统计，其中，需要实现的原则为“不重复不缺漏”。“不缺漏”可以通过每遇到一个非运算符的字符即把其认为是命题逻辑变量名称来实现。“不重复”则需要将当前的变量名称和已经统计到的命题名称依次对比：若已经出现则为重复的变量名，若未出现则为新的变量名。按照这个原则将整个字符串表达式遍历一遍，即可得到全部的变量名称与数目，代码实现具体如下：

```
// 构造函数，根据输入的表达式s为私有成员赋初值
TruthTable::TruthTable(const string s)
{
    variable temp[33];                                  // 暂时存储变量
    int cnt = 0;                                        // 变量总数
    FOR(i, 0, s.length())
    {
    judge:if (isalpha(s[i]))                            // 若表达式中某位为字母，则将该字母计入变量
    {
        FOR(j, 0, cnt)
            if (s[i] == temp[j].name)                   // 重复出现的变量名不计数
            {
                i++;
                goto judge;
            }

        temp[cnt].name = s[i], temp[cnt].value = false;// 新出现的变量放入数组中记录，同时初始值赋0
        cnt++;
    }
    }
    printf("\n该式子中的变量个数为:%d\n", cnt);          // 输出变量个数

    // 设置TruthTable类的私有成员
    variableNumber = cnt;
    expression = s;
    variables = new variable[cnt];
    principalConjunctiveNormalForm = new int[QuickPow(2, cnt, INT_MAX)];// 主合取范式中最多有2的n次方个极大项
    principalDisjunctiveNormalForm = new int[QuickPow(2, cnt, INT_MAX)];// 主析取范式中最多有2的n次方个极小项
    FOR(i, 0, cnt)
        variables[i] = temp[i];
}
```

红框处为后续求主范式初始化，与统计变量无关。

2.3 对变量进行所有情况的赋值

为得出真值表，根据真值表的定义，我们需要计算2^n种赋值情况下，表达式的值。在此，我们需要按照这个要求，对 n 个变量单元依次赋不同的值(0/1)，这些值的范围是$[0, 2^n-1]$。所以，我们需要将一个十进制数字首先转换位二进制数字，然后再将这个二进制的数字（0/1）按位赋值给 n 个命题单元，在这里我们要注意，二进制数字的低位需要赋值给排列顺序靠后的命题单元，代码实现如下：

```
// 将十进制数字num转换为二进制数字，同时将二进制数字按位赋值给variable变量
void TruthTable::SetVariablesValue(int num)
{
    int cnt = variableNumber;
    // 共为cnt个变量赋值，进行cnt次
    while (cnt)
    {
        variables[cnt - 1].value = num % 2; // 将转化后的二进制数字按位把变量表 从后向前 赋值
        num /= 2;
        cnt--;
    }
}
```

执行结束 SetVariablesValue()函数后，即将十进制数字 num 转换为了二进制数并按位赋值到了 variables 数组中。

2.4 计算某种赋值情况下的表达式的值

在将 variables 数组的值设置结束后，我们可以根据它其中变量的值来计算当前状态下表达式的值。**这一步为本题的关键，我通过网络上的 CSDN、博客园平台都没有找到已经完全实现本功能成功的案例，或多或少都存在部分问题，如单元运算符!计算错误、不能正确处理连续多个!的出现等特殊情况。而本程序保证在已经学习过的知识范围内，对所有表达式的特殊情况均可正确计算。**故本部分功能实现和介绍较为繁琐，还请麻烦老师阅读：

2.4.1 将字母表达式转换为数字

在计算之前，我们需要先遍历字母表达式，将其中的字母变量名称更换为数字字符（0/1），以便于后续的计算。代码实现如下：

```
// 将公式expression中的命题变量转换为其对应赋值（把字母换成数字）的中缀表达式
// 例：!a & b (a=0,b=1) 转换为 !0 & 1
string TruthTable::AlphaToNum()
{
    string ret = expression;
    FOR(i, 0, expression.length())
    {
        // 若当前字符为字母，则其为变量，将其转换为数字字符
        if (isalpha(expression[i]))
        {
            FOR(j, 0, variableNumber)
            {
                // 在cnt个字符中找到对应变量的赋值
                if (variables[j].name == expression[i])
                {
                    ret[i] = variables[j].value + '0';
                    break;
                }
            }
        }
    }
    return ret;
}
```

2.4.2 运算符优先级设置

在正式运算开始前，由于我们选择了利用符号栈和数栈的方式来进行计算，所以我们必须先规定好符号栈中的运算符符号出栈顺序，这通过运算符的运算优先级来决定，我们通过 CompareOperator()函数传参比较，并根据返回值得到两个运算符的优先级比较结果是“高”或者“低”，这个顺序根据课本上的 括号 优于 否定 优于 合取 优于 析取 优于 蕴涵 优于 等值 来规定设置。其中要注意的是，由于！否定运算为所有运算符中唯一的单元运算符，过于特殊，为处理连续多个！否定出现的情况，我们将这种情况的返回值设置为特殊值 0，以方便后续处理。这部分代码具体实现如下：

```
// 比较运算符优先级，顺序为 ( ) ! & | ^ ~ ，括号 否定 合取 析取 蕴涵 等值
// 返回值-1为 a < b
// 返回值 1为 a >= b
// 返回值 0为特殊操作：连续两次否定视为撤销这两次否定操作
int CompareOperator(const char a, const char b)
{
    if (a == '!')
    {
        if (b == '(') return -1;
        else if (b == '!') return 0;
        else return 1;
    }
    else if (a == '&')
    {
        if ((b == '(' || b == '!')) return -1;
        else return 1;
    }
    else if (a == '|')
    {
        if ((b == '&' || b == '(' || b == '!')) return -1;
        else return 1;
    }
    else if (a == '^')
    {
        if ((b == '&' || b == '(' || b == '|' || b == '!')) return -1;
        else return 1;
    }
    else if (a == '~')
    {
        if ((b == '~' || b == ')')) return 1;
        else return -1;
    }
    else if (a == '(')  return -1;
}
```

2.4.3 利用栈来进行计算

将表达式处理结束并设置好运算符的出入栈优先级后，我们开始这次表达式的值的运算，结合已经完成了联结词的作业，其中运算符设置如下：

```
//  &    |    ^    ~
// 合取 析取 蕴涵 等值 四种运算
bool CalcuateValue(const bool a, const char c, const bool  b)
{
    if (c == '&') return a & b;     // 合取
    if (c == '|') return a | b;     // 析取
    if (c == '^') return (!a) | b; // 蕴涵
    if (c == '~') return a == b;    // 等值
}
```

在计算中，我们采用数组来模拟栈进行任务。其中共两个栈，一个放数字，一个放运算符。运算开始前，我们在表达式最后加一个后括号)，在运算符栈中加一个前括号(，以此对括号的匹配成功作为整个表达式的运算结束的标志：

```
bool stack1[33];          // 存放数字的栈
char stack2[33];          // 存放运算符的栈
int top1 = 0, top2 = 0; // 栈顶
string s = AlphaToNum();// 将字母变量转换为数字后的公式

// 模拟将表达式整体加括号，以判断结束标志
s += ')';
stack2[top2++] = '(';
```

然后正式开始计算，遍历整个表达式。如果当前表达式字符为数字，则直接进入数字栈；如果当前表达式字符为运算符，则分为以下多种情况：

（1） 遇到左括号，则直接入栈
（2） 遇到右括号，则弹出栈中运算符进行运算，直到碰到一个左括号出栈为止。
（3） 栈顶元素优先级小于当前元素，当前元素入栈
（4） 栈顶元素优先级大于当前元素，栈顶运算符出栈，进行一次运算
（5） 遇到连续的！否定运算符，则将栈顶的！弹出，同时访问下一个表达式字符即可。

在弹出的运算符需要进行运算时，运算后得到的结果数字需要加入数字栈。由于只有！否定运算符，其他的运算符均为二元运算符，所以需要进行特殊处理。需要注意的是，不能简单的遇到二元运算符即弹出数字

栈顶两个数字运算、碰到一元运算符即弹出数字栈顶一个数字进行运算（这也是网络上大部分的示例的犯错之处，这种写法不能够对左括号后紧跟的!运算符进行正确的运算，违反了德摩根率）。必须在保证前面的元素的左括号处理过后才能这么做。

整体计算函数实现如下（代码过长，采用文本展示，为老师阅读造成的不便请谅解）：

```
1.  // 计算当前各命题变量赋值情况下的表达式的值
2.  bool TruthTable::CalculateExpressionValue()
3.  {
4.      bool stack1[33];        // 存放数字的栈
5.      char stack2[33];        // 存放运算符的栈
6.      int top1 = 0, top2 = 0; // 栈顶
7.      string s = AlphaToNum();// 将字母变量转换为数字后的公式
8.
9.      // 模拟将表达式整体加括号，以判断结束标志
10.     s += ')';
11.     stack2[top2++] = '(';
12.     FOR(i, 0, s.length())
13.     {
14.         // 表达式该位置为0/1，直接入栈
15.         if (isdigit(s[i])) stack1[top1++] = s[i] - '0';
16.
17.         // 表达式该位置为运算符
18.         else
19.         {
20.             // 遇到左括号直接入栈
21.             if (s[i] == '(') stack2[top2++] = s[i];
22.
23.             // 遇到右括号则弹出栈中运算符并进行运算，直到遇到左括号为止
24.             else if (s[i] == ')')
25.             {
26.                 // 栈为空，则直接进入下一次循环，不必进行左括号匹配
27.                 if (!top2) continue;
28.
29.                 // 弹出栈内运算符直到匹配到左括号
30.                 while (stack2[top2 - 1] != '(')
31.                 {
32.                     // !是单元运算符，需要特判
33.                     if (stack2[top2 - 1] == '!')
34.                     {
```

```
35.                         // 取数栈顶元素进行取反，并放回数栈
36.                         bool a = stack1[--top1];
37.                         if (a == 1) stack1[top1++] = 0;
38.                         else stack1[top1++] = 1;
39.
40.                         // 符号栈顶的!弹出
41.                         top2--, i--;
42.                     }
43.
44.                     // 其余二元运算符运算
45.                     else
46.                     {
47.                         // 取 b 为栈顶，a 为栈次顶元素，进行二元运算
48.                         bool b = stack1[--top1];
49.                         bool a = stack1[--top1];
50.                         stack1[top1++] = CalcuateValue(a, stack2[--
    top2], b);
51.                     }
52.                 }
53.                 top2--; // 左括号出栈
54.             }
55.
56.             else if (CompareOperator(stack2[top2 - 1], s[i]) == -
    1) stack2[top2++] = s[i];// 栈顶元素优先级小于当前元素，当前元素入栈
57.             else if (CompareOperator(stack2[top2 - 1], s[i]) == 0) to
    p2--;                 // 特殊情况，连续两个！出现，直接出栈
58.             else if (CompareOperator(stack2[top2 - 1], s[i]) == 1)
                          // 栈顶元素优先级大于当前元素，出栈
59.             {
60.                 // 非运算特殊处理
61.                 if (stack2[top2 - 1] == '!')
62.                 {
63.                     bool a = stack1[--top1];
64.                     if (a == 1) stack1[top1++] = 0;
65.                     else stack1[top1++] = 1;
66.                     top2--, i--;
67.                 }
68.                 // 其他二元运算符运算
69.                 else
70.                 {
71.                     bool b = stack1[--top1];
72.                     bool a = stack1[--top1];
73.                     stack1[top1++] = CalcuateValue(a, stack2[--
    top2], b);
```

```
                    stack2[top2++] = s[i];
                }
            }
        }
    }
    // 最终数栈中仅留下一个数字，即最终运算结果
    return stack1[top1 - 1];
}
```

2.5 主范式的统计

根据定义可知，在各命题变量赋值使得表达式为真时，此时命题变量下标的数字使得所对应为极小项； 在各命题变量赋值使得表达式为假时，此时命题变量下标的数字使得所对应为极大项。根据此定义，记录主范式如下：

```
// 对2的n次方中情况进行遍历
FOR(i, 0, lines)
{
    SetVariablesValue(i);                                                                    // 设置数字i的二进制数码为当前各变量的值
    if (CalculateExpressionValue()) principalDisjunctiveNormalForm[principalDisjunctiveNormalFormNumber++] = i; // 若当前赋值使得公式的真值为1，则为极小项，计入主析取范式
    else principalConjunctiveNormalForm[principalConjunctiveNormalFormNumber++] = i;         // 若当前赋值使得公式的真值为0，则为极大项，计入主合取范式
}
```

2.6 遍历全部情况

以上的真值表计算为计算一个值的步骤，下面展示计算全部2^n种情况的计算：

```
// 输出真值表
void TruthTable::PrintTruthTable()
{
    // 输出各变量名称作为表头
    PrintVariablesName();

    // 真值表共2的n次方行，用lines变量储存
    int n = variableNumber;
    int lines = QuickPow(2, n, INT_MAX);
    FOR(i, 0, lines)
    {
        SetVariablesValue(i);  // 设置数字i的二进制数码为当前各变量的值
        PrintVariablesValue(); // 输出各变量赋值、当前赋值状态下表达式的值
    }
}
```

其中各种控制输出表头、变量名的函数不是重点，仅控制输出格式，故在此不详细展示，若有需要请老师查看代码部分。

3 项目测试

```
**    (可运算真值表,主范式,支持括号)   **
**                                    **
**                  用!表示非         **
**                  用&表示与         **
**                  用|表示或         **
**                 用^表示蕴含        **
**                 用~表示等值        **
**                                    **
****************************************

请输入一个合法的命题公式（命题变元名称大小写敏感，请使用英文括号）:
!a&(!b&!!c^a~b)

该式子中的变量个数为:3

输出真值表如下:

  a   b   c   !a&(!b&!!c^a~b)

  0   0   0   0

  0   0   1   1

  0   1   0   1

  0   1   1   1

  1   0   0   0

  1   0   1   0

  1   1   0   0

  1   1   1   0


该命题公式的主合取范式:
        M(0)\/M(4)\/M(5)\/M(6)\/M(7)

该命题公式的主析取范式:
        M(1)/\M(2)/\M(3)
```

4 心得与总结

在本次求根据表达式求真值表和主范式的作业中，我复习并对真值表的求解、主析取范式、主合取范式的记录有了更深刻的认识。而本题的难点主要在于单元、二元运算符混杂的表达式中，如何进行表达式值的运算。在解决这个问题的过程中，我将运算主体定为波兰式的计算框架，并以此为基础，进行了许多前置要求的设置，如：设计 variable、十进制数字转二进制并对应赋值、优先级比较……

在这个过程中，我体会到了先设定目标，再为了实现项目的这个目标逆向思考的方法：“由结果逆推过程”，同样也是解题的一个重要方式。