软件设计和开发

Mini Project
设计与测试报告

2016 Spring

李琛
微电子科学与工程
13307130163@fudan.edu.cn

Table of Content

1 Overview

1.1 问题描述

- 编写一个函数，计算逻辑表达式的真值表
  - 逻辑表达式最多有 8 个输入项，分别为 A, B, C, D, E, F, G, H；
  - 支持的逻辑运算符按优先级从高到低分别为 $\sim$（非）、&（与）、$\wedge$（异或）、|（或）；
- 编写一个函数，根据真值表计算逻辑表达式，以字符串格式输出
  - 使用 Quine-McCluskey 算法对逻辑表达式进行化简。
- 函数接口：
  - std::string expr_to_truthtable(int n, const std::string& expr);

    其中 n 是变量个数，expr 是逻辑表达式，返回真值表。
  - std::string truthtable_to_expr(const std::string& truth_table);

    其中 truth_table 是真值表，返回逻辑表达式。

  若参数无效，函数抛出异常。
- 要求
  - 使用第 5 讲的测试框架对两个函数进行测试；提交源代码与设计文档。
  - 文档内容包括：设计思路、数据结构与算法、重要的类与函数的说明、测试用例的设计等等，也可以写完成 Project 的心得体会、经验与教训等。

1.2 解题概况

最终程序各部分及其作用如下表

Documents	Functions
constants.h	常量的定义
simple_test.h	参与测试
ChartToExpression.h	将真值表转换为逻辑表达式
ExpressionToChart.h	将逻辑表达式转换为真值表
implication	质蕴涵项的实现

下面就各部分的解题思路及实现进行详细讨论

2 Ideas

本次 Project 分为两部分，下面就这两部分分别进行讨论：

2.1 expr_to_truthtable

首先，通过将题目抽象，这一部分是一个经典的表达式求值的问题，其主要问题如下：

利用算符优先关系，实现对给定混合运算表达式的求值。

这是个栈的经典应用问题，常见解法是将中缀表达式（即输入的表达式）转化为后缀表达式后求解。时间复杂度为 $O(n)$，其中 n 为变量的个数

具体的算法会在 main_algorithm 模块详细介绍。

表达式求值是一个非常经典的问题，已经有了许多成熟的解法。甚至在 Python 中，一条 evaluate 语句就可以完成这样的工作。但是对于含字母的表达式，至今没有太好的解决办法。

但是考虑到本次作业的特殊性，一共只有最多 8 个变量，同时是逻辑计算而不是算术计算，这无疑极大的降低了难度。对于普通的算术含参表达式，我们只能多次取不同的大素数代入变量，还不能保证正确性。但是在本题的布尔代数式中，一共只有 $2^8 = 256$ 种取值方式，是完全可以接受的。于是，我们可以先遍历所有变量的可能取值，代入表达式中，从而计算出真值表中对应的值。

因此，总的时间复杂度为 $O(n * 2^n)$, 由于 $n \leq 8$，复杂度不会超过 $8 * 2^8 = 2048$，是完全可以接受的。

2.2 truthtable_to_expr

对于将真值表转化为表达式的问题，我们在数字逻辑电路中曾学习过使用卡诺图的方法，非常直观，同时在对于维数较低时也易于化简。

但是卡诺图的解题过程较为繁琐，同时对于高维的情况扩展性较差。在 $n \geq 6$ 时，就比较难画出合适的卡诺图了。因此，我们选用 Quine-McCluskey 算法。它虽然较为抽象，但是非常模式化，对高维的扩展性很好。

至于算法的时间复杂度方面，由于化简表达式至今还是 NP-hard 问题，因此只存在指数级解法，即问题的时间复杂度为 $O(2^n)$

同时，题目要求将表达式化到最简形式，因此在 Quine-McCluskey 算法后我们必须进行化简。这里采用 petrick 化简算法，时间复杂度同样为 $O(2^n)$

具体的算法会在 main_algorithm 模块详细介绍。

2.3 异常处理

2.3.0.1 expr_to_truthtable

- 对于输入为空字符串，或者都是空格的情况，应该抛出 **EmptyStringError**

- 若输入中出现了非操作符和大写字母以外的字符，应该抛出 **InvalidCharError**

  例如："A + B"　"a & b"

- 若输入的变量没有按照字典顺序命名，则应该抛出 **InvalidVariableError**

  例如：n = 2, expr = "A & C"

- 输入的表达式出现了语法错误，应该抛出 **SyntaxError**

  例如："A &| B"　"A | () B"

- 表达式中括号不匹配，应该抛出 **BracketMismatchingError**

2.3.0.2 truthtable_to_expr

- 对于输入为空字符串，或者都是空格的情况，应该抛出 **EmptyStringError**
- 若输入中出现了 0 和 1 以外的字符，应该抛出 **InvalidCharError**

  例如："123"　"1abc"

- 若输入真值表的长度不是 2 的整数次幂，应该抛出 **InvalidLengthError**

  例如："111"　"00000"

3 Project Structures

3.1 ChartToExpression

对于第一个问题，我们使用 ChartToExpression 类来实现，程序声明如下：

```
const char Operators[] = {'|', '^', '&', '~'};
class ExpressionToChart {
private:
        stack<char> Stack_Operator;
        stack<bool> Stack_Number;
        vector<int> loc[MAX_N];
        string filter(const string &s);
        void PushString(string &s, const char &ch);
        int priority(char Operator);
        bool IsOperator(char ch);
        void GetTwoNumbers(stack<bool> &Stack, bool &first, bool &second);
        bool CalcEq(bool first, bool second, char op);
        string InfixToPostfix(const string &infix);
        int SolvePostfix(const string &postfix);
public:
        ExpressionToChart();
        virtual ~ExpressionToChart();
        string solve(int n, const string &InString);
};
```

其中各变量及函数作用如下：

- 变量

变量	作用
operators[]	储存定义的四种运算符
Stack_Operator	运算符的堆栈
Stack_Number	运算结果的堆栈
loc[MAX_N]	储存字符出现位置，便于代入数值

- 函数

函数	作用
ExpressionToChart()	构造函数
virtual ExpressionToChart()	析构函数
Stack_Number	运算结果的堆栈
string solve(int n, const string &InString)	将输入的表达式转换为真值表输出
string filter(const string &s)	将输入的表达式去除多余空格
string InfixToPostfix(const string &infix)	中缀表达式转换为波兰式
void PushString(string &s, const char &ch)	转化时向波兰式中添加字符
bool IsOperator(char ch)	判断当前字符是否为运算符
int priority(char Operator)	若是运算符，返回其优先级
int SolvePostfix(const string &postfix)	计算后缀表达式的值
void GetTwoNumbers(bool &first, bool &second)	在运算结果栈中取出两个元素用于计算
bool CalcEq(bool first, bool second, char op)	给定运算数和运算符，计算结果

3.2 ExpressionToChart

对于将真值表转换为表达式的问题，我们采用 Quine-McCluskey 算法，主要使用 ExpressionToChart 类实现。同时，由于对质蕴涵项这一概念使用较多，因此将它单独建立 implication 类，以增强内聚度

implication 类声明如下：

```
class implication {
private:
        friend bool operator < (const implication &a, const implication &b) {
                        return a.ones < b.ones;
        }
public:
        int bit;
        int xterm;
        int ones;
        string exp;
        bool used;
        bool selected;
        vector<int> ImpContained;
        int CountOne(int x);
        int TotalVariables;
        implication();
        virtual ~implication();
        implication(int mask, int _xterm, int _TotalVariables);
        string show();
};
```

其中各变量及函数作用如下：

- 变量

变量	作用
int TotalVariables	总变量个数
int bit	当前质蕴涵的二进制表示
int xterm	质蕴涵的任意项，该位为 1 即为任意项
int ones	质蕴涵中 1 的个数
string exp	当前质蕴涵的二进制表示（输出用）
bool used	表示该蕴涵项是否被覆盖 (Q-M 算法中)
bool selected	表示该蕴含项是否是出现在结果中（petrick 化简）
vector<int> ImpContained	该质蕴涵包含的蕴涵项

- 函数

函数	作用
implication()	默认构造函数
virtual implication()	析构函数
implication(int mask, int xterm, int TotalVariables)	构造函数
int CountOne(int x)	统计 x 的二进制表示中 1 的个数
string show();	输出该蕴涵项

ChartToExpression 类声明如下：

```
class ChartToExpression {
private:
        vector<int> MinTerm;
        vector<int> MinTermCovered[1 << MAX_N];
        bool table[1 << MAX_N][1 << MAX_N];
        bool contained[1 << MAX_N];
        int TotalVariables;
        vector<implication> imp, roller;
        vector<implication> primes;
        bool CheckContained(const implication &imp, const int &x);
        int CountOne(int x);
        vector<implication*> UPI;
public:
        ChartToExpression();
        virtual ~ChartToExpression();
        void Simplify();
        void Quine_McCluskey();
        string solve(const string &truth_table);
        void ShowTable();
};
```

其中各变量及函数作用如下：

- 变量

变量	作用
int TotalVariables	总变量个数
vector<int> MinTerm	输入的最小项, 用 int 表示
vector<implication> imp	储存最小项, 参与 Q-M 算法计算
vector<implication> roller	Q-M 算法中与 imp 迭代的数组
vector<int> MinTermCovered[1 « MAX_N]	可以覆盖该最小项的所有蕴涵
bool table[1 « MAX_N][1 « MAX_N]	可视化蕴涵与最小项的关系
bool contained[1 « MAX_N]	表示每个蕴涵项是否需要 petrick 化简
vector<implication> primes	储存质蕴涵项

- 函数

函数	作用
ChartToExpression()	构造函数
virtual ChartToExpression()	析构函数
string solve(const string &truth_table)	输入真值表，返回化简后表达式
void Quine-McCluskey()	实现 Quine-McCluskey 算法
void Simplify()	对结果化简，使用到了 petrick 算法
int CountOne(int x)	统计 x 的二进制表示中 1 的个数
bool CheckContained(const implication &imp, const int &x)	判断蕴涵 Imp 是否包含了最小项 x
void ShowTable()	将中间结果可视化

4 Main Algorithm

4.1 ChartToExpression

4.1.1 表达式转化

首先，我们需要将含字母的表达式转换为只含有数字的表达式，以便后续计算。

核心代码如下：

```
int len = s.length();
for (int i = 0; i < len; i++)
        if (isalpha(s[i])) {
                loc[s[i] - 'A'].push_back(i);
        }
string ans = "";
for (int mask = (1 << n) - 1; mask >= 0; mask--) {
        infix = s;
        for (int i = 0; i < n; i++) {
                int value = (mask >> i) & 1;
                for (int j = 0; j < loc[i].size(); j++) {
                        int pos = loc[i][j];
                        infix[pos] = '0' + value;
                }
        }
        char result = '0' + SolvePostfix(InfixToPostfix(infix));
        ans = ans + result;
}
```

- 首先，遍历输入的字符串，对于每一个字母，用 *loc*[] 数组记下它出现的位置
- 然后，我们使用变量 *mask* 来遍历 2^n 种可能输入。由于输出格式的要求，我们采用倒序遍历，即从 2^n-1 开始，到 0 结束。当 *mask* 值一定时，对于每一位 i，可以用位运算 $value = (mask >> i)\&1$ 来得到第 i 个变量的值 *value*。
- 得到第 i 个变量的值 *value* 之后，通过遍历 *loc*[i] 数组，就可以得到该变量所有出现的位置，从而进行替换。
- 由于变量只有 $A-H$, 取值也只有 $0-1$，都只有一位，所以直接赋值即可。于是完成了从含字母表达式到纯数字表达式的转换。

4.1.2 表达式求值

4.1.2.1 中缀转后缀

4.1.2.1.1 中缀表达式与后缀表达式

- 中缀表达式
  中缀表示法是算术表达式的常规表示法。称它为中缀表示法是因为每个操作符都位于其操作数的中间，这种表示法只适用于操作符恰好对应两个操作数的时候（在操作符是二元操作符如加、减、乘、除以及取模的情况下）。

  Syntax: operand1 operator operand2

  Example: (A+B)*C-D/(E+F)

- 后缀表达式
  在后缀表示法中，操作符严格位于操作数后面。因其使表达式求值变得轻松，所以被普遍使用。

  Syntax : operand1 operand2 operator

  Example : AB+C*DEF+/-

- 由于中缀表示法是书写表达式的常见方式，而后缀表达式更便于计算表达式的值，因此我们常常采用将中缀表达式转化为后缀表达式再求值的方法。

4.1.2.1.2 具体算法

中缀转换为后缀的难点在于操作符的优先级处理。由于后缀的操作符紧跟在操作数后，我们可以用一个栈来存储操作符，具体流程如下：

- 初始化一个空堆栈，将结果字符串变量置空。
- 从左到右读入中缀表达式，每次一个字符。
- 如果字符是操作数，将它添加到结果字符串。
- 如果字符是个操作符，弹出操作符，直至遇见左括号、优先级较低的操作符或者同一优先级的右结合符号。把这个操作符压入堆栈。
- 如果字符是个左括号，把它压入堆栈。
- 如果字符是个右括号，在遇见开括号前，弹出所有操作符，然后把它们添加到结果字符串。
- 如果到达输入字符串的末尾，弹出所有操作符并添加到结果字符串。

Project 中，用 *InfixToPostfix* 函数实现：

```
string InfixToPostfix(const string &infix) {
        string postfix = "";
        for (auto ch : infix) {
                if (IsOperator(ch)) {
                        while (!Stack_Operator.empty() &&
                                IsOperator(Stack_Operator.top()) &&
                                priority(Stack_Operator.top()) >= priority(ch)){
                                PushString(postfix, Stack_Operator.top());
                                Stack_Operator.pop();
                        }
```

```
                        Stack_Operator.push(ch);
                }
                else {
                switch(ch) {
                        case '(' :
                                Stack_Operator.push(ch);
                                break;
                        case ')' :
                                while (!Stack_Operator.empty() &&
                                        Stack_Operator.top() != '(') {
                                        PushString(postfix,Stack_Operator.top());
                                        Stack_Operator.pop();
                                }
                                Stack_Operator.pop();
                                break;
                        default :
                                PushString(postfix, ch);
                }
                }
        }
    while (!Stack_Operator.empty()) {
                PushString(postfix, Stack_Operator.top());
                Stack_Operator.pop();
    }

    postfix.pop_back();
    return postfix;
}
```

4.1.2.2 后缀表达式求值

对后缀表达式求值比直接对中缀表达式求值简单。在后缀表达式中，不需要括号，而且操作符的优先级也不再起作用了。

具体算法如下：

- 初始化一个空堆栈
- 从左到右读入后缀表达式
- 如果字符是一个操作数，把它压入堆栈。
- 如果字符是个操作符，弹出两个操作数，执行对应操作，然后把结果压入堆栈。

  如果不能够弹出两个操作数，后缀表达式的语法就不正确。
- 到后缀表达式末尾，从堆栈中弹出结果。若后缀表达式格式正确，那么堆栈应该为空。

Project 中，用 *SolvePostfix* 函数实现：

```
int SolvePostfix(const string &postfix) {
        while (!Stack_Number.empty()) {
                Stack_Number.pop();
        }
        for (auto ch : postfix) {
                if (IsOperator(ch)) {
                        if (ch == '~') {
                                bool x = Stack_Number.top();
                                Stack_Number.pop();
                                Stack_Number.push(!x);
                        }
                        else {
                                bool x, y;
                                GetTwoNumbers(Stack_Number, x, y);
                                bool result = CalcEq(x, y, ch);
                                Stack_Number.push(result);
                        }
                }
                else
                        if (ch != ' '){
                                Stack_Number.push(ch - '0');
                        }
        }
        int ans = Stack_Number.top();
        Stack_Number.pop();
        return ans;
}
```

4.2 ExpressionToChart

4.2.1 QuineMcCluskey 算法

QuineMcCluskey 算法是最小化布尔函数的一种方法。它在功能上等同于卡诺图，但是它具有文字表格的形式，因此它更适合用于电子设计自动化算法的实现，并且它还给出了检查布尔函数是否达到了最小化形式的确定性方法。

QuineMcCluskey 算法的基本步骤如下：

- STEP 1
  将表达式中的最小项用他们的等价二进制数表示

- STEP 2

  将最小项根据它们所含的 1 的个数进行分组，形成一张最小项表。

  如第一组为不含 1 的，第二组中每个含一个 1，以此类推

- STEP 3

  将每个最小项 i 与它下一组的所有最小项 j 进行比较。显然，j 会比 i 多一个 1。

  如果 i 与 j 只有一位 b 不同（包括任意项），标记这两个最小项配对成功，将 b 位标记为任意项。

  将合并后的新最小项加入新的最小项表中。

- STEP 4

  对于每一次形成的新最小项表，重复 STEP 3，直到不能产生任何的新配对。

- STEP 5

  每一步剩下未配对的即为质蕴涵。（但并不是必要质蕴涵）

Quine-McCluskey Method

List 1						List 2						List 3				
mi	x1	x2	x3	x4		mi	x1	x2	x3	x4		mi	x1	x2	x3	x4
2	0	0	1	0	ok	2,6	0	-	1	0		8,9,12,13	1	-	0	-
4	0	1	0	0	ok	2,10	-	0	1	0		8,12,9,13	1	-	0	-
8	1	0	0	0	ok	4,6	0	1	-	0		Finished				
6	0	1	1	0	ok	4,12	-	1	0	0						
9	1	0	0	1	ok	8,9	1	0	0	-	ok					
10	1	0	1	0	ok	8,10	1	0	-	0						
12	1	1	0	0	ok	8,12	1	-	0	0	ok					
13	1	1	0	1	ok	9,13	1	-	0	1	ok					
15	1	1	1	1	ok	12,13	1	1	0	-	ok					
						13,15	1	1	-	1						

Abbildung 1: 最小项表

由于 QuineMcCluskey 算法的描述比较清晰，按描述模拟即可。

```
1  void Quine_McCluskey() {
2          //keep looping until no more pairs
3          while (imp.size() > 0) {
4                  roller.clear();
5                  //deleting redundant terms
6                  for (int i = 0; i < imp.size(); i++)
7                          for (int j = i + 1; j < imp.size(); j++)
8                                  if (imp[i].exp == imp[j].exp) {
9                                          imp.erase(imp.begin() + j);
```

```
                                }
                //finding new pairs
                for (int i = 0; i < imp.size(); i++)
                        for (int j = i + 1; j < imp.size(); j++)
                                //if i and j can form pairs
                                if (imp[j].ones == imp[i].ones + 1
                                && imp[j].xterm == imp[i].xterm
                                && CountOne(imp[j].bit ^ imp[i].bit) == 1) {
                                imp[i].used = true;
                                imp[j].used = true;
                                implication NewImp = implication(imp[i].bit,
                                        (imp[i].bit ^ imp[j].bit) | imp[i].xterm,
                                        TotalVariables);
                                roller.push_back(NewImp);
                                }
                //terms that cannot be paired are prime
                for (int i = 0; i < imp.size(); i++)
                        if (imp[i].used == false)
                                primes.push_back(imp[i]);
                sort(roller.begin(), roller.end());
                imp = roller;
        }
}
```

4.2.2 化简

化简分为两步：

- 若一个最小项只被一个质蕴涵包含，则该质蕴涵必须选
- 若剩余未被覆盖的最小项均被多个质蕴涵包含，则运用 petrick 算法进行化简

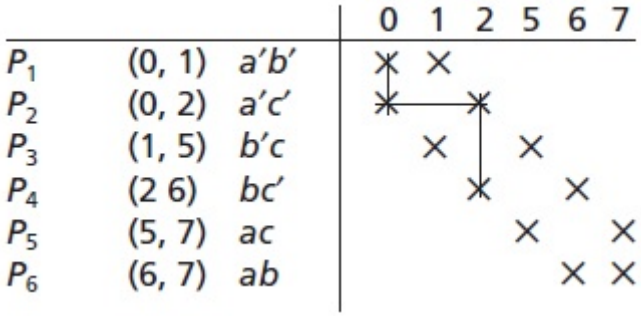
4.2.2.1 基础化简

首先进行筛选，若一个最小项只被一个质蕴涵包含，则直接标记该质蕴涵为已选

```
for (unsigned i = 0; i < MinTerm.size(); i++)
        if (MinTermCovered[i].size() == 1) {
                int index = MinTermCovered[i][0];
                primes[index].selected = true;
                for (unsigned j = 0; j < primes[index].ImpContained.size(); j++)
                        contained[primes[index].ImpContained[j]] = true;
        }
```

4.2.2.2 进阶化简

若出现以下这种情况，基本的化简就无计可施了：



即每个最小项都被至少两个质蕴涵覆盖。

通用的算法是使用 petrick 化简，但是由于本题数据范围较小，可以直接对剩下的质蕴涵，直接穷举它是否被取，然后从所有方案中取选择质蕴涵最少的方案。

这种方法的时间复杂度为 $O(2 \wedge$ 剩余质蕴涵个数$)$，由于剩余质蕴涵个数不会超过 $2^{n/2}$，因此时间复杂度不会超过 $2^{16} = 65536$，是完全可以接受的

```
for (unsigned i = 0; i < primes.size(); i++)
        if (primes[i].selected == false)
                UPI.push_back(&primes[i]);
bool tContained[1 << MAX_N];
int MinOne = 1 << MAX_N;
int result = 0;
for (int mask = 0; mask < (1 << (UPI.size())); mask++) {
        memset(tContained, false, sizeof tContained);
        for (unsigned i = 0; i < UPI.size(); i++) {
                int value = (mask >> i) & 1;
                implication *mImp = UPI[i];
                if (value) {
                        for (unsigned j = 0; j < mImp->ImpContained.size(); j++)
                                tContained[mImp->ImpContained[j]] = true;
                }
        }
        bool valid = true;
        for (unsigned i = 0; i < MinTerm.size(); i++)
                if (contained[i] == false && tContained[i] == false)
                        valid = false;
        if (valid) {
                int ones = CountOne(mask);
                if (MinOne > ones) {
                        MinOne = ones;
```

```
25                          result = mask;
26                  }
27          }
28  }
29  for (int i = 0; i < UPI.size(); i++) {
30          int value = (result >> i) & 1;
31          if (value)
32                  UPI[i]->selected = true;
33  }
```

5 测试

5.1 正确性测试

首先，对于程序的正确性进行测试

5.1.0.3 expr_to_truthtable

```
//testing expression to chart

//check sample
CHECK_EQUAL(ETC.solve(3, "~A | B & C"), "11010101");

//check unit input
CHECK_EQUAL(ETC.solve(1, "A"), "10");
CHECK_EQUAL(ETC.solve(2, "A"), "1010");

//check complex input
CHECK_EQUAL(ETC.solve(4, "A & ~B ^ C & D"), "1101001000100010");
```

5.1.0.4 truthtable_to_expr

```
//testing chart to expression

//check all 0
CHECK_EQUAL(CTE.solve("0000"), "0");

//check all 1
CHECK_EQUAL(CTE.solve("11"), "1");

//check sample
CHECK_EQUAL(CTE.solve("11010101"), "~A | B & C");

//check complex input
CHECK_EQUAL(CTE.solve(
        "1111110111011101111111111111111111111111011101110111111110111011101"),
        "~A | ~E & F | B | C & D");
```

5.1.0.5 cross_validation

```
//cross testing
```

```
CHECK_EQUAL(ETC.solve(4, CTE.solve("1101001000100010")), "1101001000100010");
CHECK_EQUAL(ETC.solve(6, CTE.solve(
        "1111110111011101111111111111111111111101110111011111110111011101"")),
        "1111110111011101111111111111111111111101110111011111110111011101");
```

5.2 健壮性测试

检测程序对于异常输入的处理

5.2.0.6 expr_to_truthtable

划分等价类：

错误类型	抛出异常
空串	EmptyStringError
非法字符	InvalidCharError
参数个数错误	InvalidVariableError
语法错误	SyntaxError
括号不匹配	BracketMismatchingError
正常输入	No Error

```
//testing expression to chart

//check empty input
CHECK_THROW(ETC.solve(1, "    "), EmptyStringError);

//check invalid characters
CHECK_THROW(ETC.solve(2, "A + B"), InvalidCharError);

//check invalid variables
CHECK_THROW(ETC.solve(2, "A | C"), InvalidVariableError);

//check syntax error
CHECK_THROW(ETC.solve(2, "A | ^ B"), SyntaxError);
CHECK_THROW(ETC.solve(1, "AA"), SyntaxError);
CHECK_THROW(ETC.solve(2, "A(A | B)"), SyntaxError);

//check bracket mismatching
CHECK_THROW(ETC.solve(2, "A ^ B)"), BracketMismatchingError);
```

5.2.0.7 truthtable_to_expr

划分等价类：

错误类型	抛出异常
空串	EmptyStringError
非法字符	InvalidCharError
参数个数 $!=2^n$	InvalidLengthError
正常输入	No Error

```
//testing chart to expression

//check empty input
CHECK_THROW(CTE.solve(""), EmptyStringError);

//check invalid input numbers
CHECK_THROW(CTE.solve("1234"), InvalidCharError);

//check invalid truthtable length
CHECK_THROW(CTE.solve("101"), InvalidLengthError);
```

5.3 效率测试

程序运行的速度也是很重要的。

由于 8 个变量的表达式运行时间过长，我们测试的表达式均采用 6 个变量。

测试流程如下：

- 随机产生 100 个长度为 $2^6 = 64$ 的真值表
- 对于每个真值表，运用 cross_validation 的方法，将 expr_to_truthtable 和 truthtable_to_expr 两个函数都运行一遍
- 统计总用时，进而得出长度为 6 时测试数据的平均用时

```
for (int i = 0; i < MAX_CASES; i++) {
        string test_string = "";
        //generate a test string
        int NumVar = 6;
        for (int j = 0; j < (1 << NumVar); j++) {
                int val = rand() % 2;
                test_string += val + '0';
        }
        clock_t start_time = clock();
        string expr = CTE.solve(test_string);
        clock_t cte_time = clock();
        CHECK_EQUAL(ETC.solve(NumVar, expr), test_string);
        clock_t end_time = clock();
```

```
14          // truth_table to expression time
15          double mCTE = (double)(cte_time - start_time) / CLOCKS_PER_SEC;
16          // expression to truth_table time
17          double mETC = (double)(end_time - cte_time) / CLOCKS_PER_SEC;
18          // total time
19          double mTotal = (double)(end_time - start_time) / CLOCKS_PER_SEC;
20          //output the time
21          fout << i + 1 << ',' << mCTE << ',' << mETC << ',' << mTotal << endl;
22  }
```

6 总结与体会

这次 project 给我的帮助主要有两点：学习了新的算法和认识到了测试的重要性。

完成这个 Project 大概花了我两个星期时间，其中确立思路与框架花了一天，写程序花了两天，剩下的时间都用于调试与测试数据。以前我对测试的重要性有所忽视，总认为反正程序很短，运用中可以出了问题再改。但是这次的工程量略微上升了一点，这种方法的弊病就出现了。对于一个表达式而言，需要考虑的 corner case 非常多。如果程序不能有一个很好的架构的话，很难考虑到所有情况，调试将会变得非常麻烦。因此，程序的合理组织比正确的写出程序更加重要。

而就这个 project 而言，它需要的两个函数 expr_to_truthtable 和 truthtable_to_expr，我分别用了两个类 ExpressionToChart 和 ChartToExpression 来完成，而在 ChartToExpression 中，针对质蕴涵项这个概念被多次使用的情况，又另见了 *implication* 类来专门实现质蕴涵的相关操作。这样，这个 project 的架构就比较清晰，也比较利于调试了。

当然，这次 project 让我学习了 Q-M 算法和 petrick 化简，同时也复习了表达式求值这以经典算法。在实现上来说，表达式求值相对易于实现，因此 expr_to_truthtable 这个函数实现的较为轻松。而 Q-M 算法是我较为陌生的一种算法，较为繁琐。我在实现时也遇到了一些困难，但主要是由于我对于算法不太熟悉，思路不够清晰。经过几次的重构，这一部分才算完成。《孙子兵法》中说："谋定而后动，知止而有得"。对于一个工程来说，要时刻保持思路的清晰，这比纯粹的垒代码重要的多。

很惭愧，就做了一点微小的工作。Linus Torvalds 说过："Talk is cheap, show me the code."希望以这次的 Mini_Project 作为起步，通过期末的 Final_Project, 在实践中不断提示自己的工程能力。