Mini Project doc

18300750070 黄听

目录

1.概述

该程序用于逻辑表达式和真值表之间的相互转化，其达到的要求如下（引用自PPT）：

1. 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~（非）、&（与）、^（异或）、|（或）；可使用括号改变计算的顺序。输出结果是一个由'0'或'1'组成的字符串。

例如，逻辑表达式"~A | B & C"的真值表字符串为"11010101"

C	B	A	~A	B & C	~A | B & C
1	1	1	0	1	1
1	1	0	1	1	1
1	0	1	0	0	0
1	0	0	1	0	1
0	1	1	0	0	0
0	1	0	1	0	1
0	0	1	0	0	0
0	0	0	1	0	1

2. 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。

两个函数接口分别为：

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。

2.设计思路与算法简介

2.1 函数 expr_to_truthtable 的设计思路与算法简介

该函数的主要功能为将逻辑表达式转化为由字符'0'和'1'所组成的字符串形式真值表。为实现该功能，本函数的主要设计思路是利用 C++的 regex 库对输入逻辑表达式字符串进行正则表达式匹配与替换，以完成字符串的转换。该函数首先通过替换空格、制表符与换行符为空以达成规范输入字符串的效果。然后，本程序通过不断转换一个由 0 和 1 组成的长度为 8 的 vector 中各项的值（每一次将一位取反，不重复）以遍历所有输入，并通过正则表达式替换，将输入的逻辑表达式中的 A-H 这 8 个字母替换为 0 或 1，产生遍历所有输入的由 0，1 与四种逻辑运算符构成的逻辑表达式。接着，根据非，与，异或，或的优先级对该逻辑表达式进行正则表达式替换，以计算得到最终的“0”或“1”结果。在这一步中，对于有括号的情况，函数利用了递归的方法，将括号内的表达式先进行运算，以达到去除括号的作用。最后，程序将这些“0”或“1”拼接为符合输出要求的真值表字符串。

2.2 函数 truthtable_to_expr 的设计思路与算法简介

根据题目要求，本函数利用 Quine-McCluskey 算法对输入的真值表对应的逻辑表达式进行化简成为最简逻辑表达式。该算法的主要步骤如下：

① 将输入的由'0'和'1'组成的字符串中'1'的数量和位置进行统计，并由此产生相应的最小项的数目和编码。

② 对各最小项的编码转化为二进制，对其中 1 的数量进行计数，并按此分为若干组，生成列表。

③ 将相邻组中的各项编码逐一进行比较，若只有一位不相同的，将原来两项做好标记，将这两项合成的新项（不同位记为'-'，其余位不变）列入新表。

④ 对于新表重复上面的操作，直至无法产生新表。

⑤ 取出各个列表中未被标记的项（这些项称为质蕴涵项），并通过其二进制编码确定每个质蕴涵项所包含的最小项编码，列入表中。

⑥ 找出所有能包含所有最小项的质蕴涵项组合。

⑦ 找到这些组合中项数最少且变量数最少的组合，其中所有质蕴涵项相或即为所求的输出表达式。

3.类与函数的说明

3.1 与函数 expr_to_truthtable 相关的函数与类

3.1.1 类 E2T

这是包含由逻辑表达式转为真值表所用到的变量与函数的类。其名称是 expression to truthtable 的缩写。其成员 string express 是输入的逻辑表达式，int nvar 是变量的个数，vector<int> bits 是包含用于替换 A-H 字母的 0 和 1 组成的数组。下面介绍该类的成员函数。

3.1.2 构造函数 E2T(int n, const string& expr)

该函数用于检查输入是否合法并构造类。步骤如下：

① 利用 regex pat1("[\t\n]");删除输入字符串中的空格、换行符与制表符。

② 检测输入中 n 的值是否大于 8 或小于根据变量而得到的最小值，若是，抛出 num_err 异常。

③ 通过两个 errpat

```
regex errpat1("[^A-H()|^&~]");
regex
errpat2("\\w{2}|[|^&]{2}|~[|^&]|^[|^&]|[|^&]$|[|^&~]\\)|\\([|^&]|\\(\\)|\\)\\(|\\)~|\\w\\(|\\)\\w");
```

分别检测是否为空串，有除了 A-H 字母、括号与逻辑运算符之外的非法输入字符以及是否有两个或以上字母、括号或逻辑运算符相邻等的非法输入，并通过 is_bracket_valid 函数判断是否有括号不匹配的非法输入，若有，抛出 expr_err 异常。

④ 构造类。

3.1.3 函数 bool is_bracket_valid(const string& str)

先通过正则表达式替换清除字符串中所有不为小括号的字符，并利用<stack>，通过堆栈的方法判断括号是否匹配。

3.1.4 函数 string expr_to_binary()

利用 8 个 regex pattern("A")-("H")将输入的逻辑表达式字符串中的'A'-'H'字母替换为 vector bits 中的 0 或 1。这里利用了 to_string(int)函数将整数形式的 0 或 1 转化为字符串形式。

3.1.5 函数 string solve_binary(const string& bistr)

该函数用于求解由 0,1 以及逻辑运算符和括号组成的逻辑表达式。首先，利用 regex pan("\\(([^\\(\\)]+)\\)");匹配内部没有括号的括号，对其先进行求解。

代码

```
while (regex_search(restr, m, pan))
    restr=regex_replace(restr,pan,solve_binary(m[1]),std::regex_const
ants::format_first_only);
```

通过递归的方式对括号内的不含括号的逻辑表达式进行求解以达到去括号的效果。这里用了参数 std::regex_constants::format_first_only，一次只替换一个，以防止替换过程紊乱，结果出错。

在逻辑表达式中没有括号的情况下，通过若干个 regex pattern：

```
regex not0("~0");
regex not1("~1");
regex and00("0&0");
regex and01("0&1");
regex and10("1&0");
regex and11("1&1");
regex xor00("0\\^0");
regex xor01("0\\^1");
regex xor10("1\\^0");
regex xor11("1\\^1");
regex or00("0\\|0");
regex or01("0\\|1");
regex or10("1\\|0");
regex or11("1\\|1");
```

对表达式内可能会出现的非、与、异或和或运算进行正则表达式匹配，并按照优先级顺序进行替换，例如，对与的替换代码如下：

```
while (regex_search(restr, and00) || regex_search(restr, and01) ||
regex_search(restr, and10) || regex_search(restr, and11))
    {
        restr = regex_replace(restr, and00, "0");
        restr = regex_replace(restr, and01, "0");
        restr = regex_replace(restr, and10, "0");
        restr = regex_replace(restr, and11, "1");
    }
```

这里将与的四种可能形式 (0&0,0&1,1&0,1&1) 替换为相应的运算结果 (0,0,0,1)，并利用 while 循环确保逻辑表达式中所有的&都被替换完成。

3.1.6 函数 string gen_truthtable()

该函数用于依次遍历 A-H 各变量所能取到的值，放入函数 string expr_to_binary() 与函数 string solve_binary(const string& bistr)中进行替换和计算，并将运算结果拼接为输出要求的真值表字符串。

vector bits 中，bits[0]对应的是变量 A，bits[1]对应的是变量 B，依此类推。按照题目要求，bits 的初始赋值为 8 个 1。

通过 for 循环进行对 bits 的赋值与运算、拼接字符串等操作:

```
for (int i = 0; i < alltimes; i++)
{
    string bistr = expr_to_binary();
    string bit = solve_binary(bistr);
    truth_table += bit;
    bits[0] ^= 1;
    if (nvar >= 2 && i % 2 == 1) bits[1] ^= 1;
    if (nvar >= 3 && i % 4 == 3) bits[2] ^= 1;
    if (nvar >= 4 && i % 8 == 7) bits[3] ^= 1;
    if (nvar >= 5 && i % 16 == 15) bits[4] ^= 1;
    if (nvar >= 6 && i % 32 == 31) bits[5] ^= 1;
    if (nvar >= 7 && i % 64 == 63) bits[6] ^= 1;
    if (nvar >= 8 && i % 128 == 127) bits[7] ^= 1;
}
```

其中bits的有效位数是前nvar（变量个数）位，alltimes是2的nvar次方，为bits可能取到的值的个数。

```
string bistr = expr_to_binary();
string bit = solve_binary(bistr);
```

用于转换输入逻辑表达式与计算结果。

```
truth_table += bit;
```

用于拼接结果为真值表字符串。

```
bits[0] ^= 1;
if (nvar >= 2 && i % 2 == 1) bits[1] ^= 1;
if (nvar >= 3 && i % 4 == 3) bits[2] ^= 1;
if (nvar >= 4 && i % 8 == 7) bits[3] ^= 1;
if (nvar >= 5 && i % 16 == 15) bits[4] ^= 1;
```

```
if (nvar >= 6 && i % 32 == 31) bits[5] ^= 1;
if (nvar >= 7 && i % 64 == 63) bits[6] ^= 1;
if (nvar >= 8 && i % 128 == 127) bits[7] ^= 1;
```

用于按照 i 的值依次对 bits 中有效位上的 0 或 1 进行取反，以达到遍历所有可能值的效果。

3.2 与函数 truthtable_to_expr 相关的函数与类

3.2.1 类 T2E

这是包含由真值表转为字符串所用到的变量与函数的类。其名称是 truthtable to expression 的缩写。其成员 string truthtable 是输入的真值表字符串，vector<int> intminterm 是最小项编号组成的 vector（十进制形式），vector<string> bitminterm 是最小项编号组成的 vector（二进制形式），int bitcount 为输入真值表的位数，int mincount 为最小项的个数，vector<set<string>> table 为最小项二进制形式按其中 1 的个数排列形成的列表，其第一维为二进制形式中 1 的数量，vector<string> prime 为质蕴涵项组成的 vector，set<int> rescomb 为最终得到的结果中所需的质蕴涵项组合中质蕴涵项的编号组成的 set，int allflag 用于处理真值表全为 0 或全为 1 的特殊情况，初始值为-1。下面介绍该类的成员函数。

3.2.2 构造函数 T2E(const string& ttable)

该函数首先判断是否有以下几种情况的非法输入：

① 输入字符串中有 0 或 1 之外的字符。

② 输入字符串为空串。

③ 输入字符串的长度不为 2 的 1-8 次方。

若有，抛出 truth_table_err 异常。

再判断真值表是否全为 0 或全为 1，若是，分别将 allflag 赋值为 0 与 1。这将在函数 truthtable_to_expr 中得以处理，将分别直接返回“0”与“1”字符串，不进行接下来的运算。

最后，构造类。先赋值 truthtable 并将其前后颠倒（考虑到题目要求的顺序，A 为最低位所对应的项）。再通过代码

```
for (i = 1; i <= 8; i++) if (pow(2, i) == truthtable.length()) break;
bitcount = i;
```

对 bitcount 进行计数。

通过代码

```
for (i = 0; i < truthtable.length(); i++)
{
    if (truthtable[i] == '1')
    {
        countone++;
        intminterm.push_back(i);
        binminterm.push_back(int_to_binstr(bitcount, i));
    }
}
mincount = countone;
```

构造出 intminterm 和 binminterm 两个 vector 并对 mincount 进行计数。

通过代码

```
table = vector<set<string>>(bitcount + 1);
for (int i = 0; i < mincount; i++)
{
    int onecount = count_str_one(binminterm[i]);
    table[onecount].insert(binminterm[i]);
}
```

对每个最小项中 1 的个数进行计数，并按照计数结果列入 table 内。

这个函数完成了 2.2 中的步骤①与②。

3.2.3 函数 int count_str_one(const string& a)

该函数通过遍历字符串，对字符串中字符'1'的数目进行计数。

3.2.4 函数 string bin_to_expr(const string& binstr)

该函数将由二进制数组成的字符串转换为最小项逻辑表达式字符串。通过 for 循环

```
for (int i = binstr.length() - 1; i >= 0; i--)
{
    int j = binstr.length() - 1 - i;
    if (binstr[i] == '1')
    {
        char ch = 65 + j;
        restr += ch;
        restr += "&";
    }
    else if (binstr[i] == '0')
    {
        char ch = 65 + j;
        restr += "~";
        restr += ch;
        restr += "&";
    }
}
```

对二进制字符串进行遍历，根据当前下标在 restr 中加入相应的 A-H 字母以及与运算符'&'，若当前位为'0'，在字母前加入取反符号'~'。

最后利用代码

```
if (restr[restr.length() - 1] == '&') restr.pop_back();
```

将可能产生的多余的'&'字符去除。

3.2.4 函数 string int_to_binstr(const int& size, int a)

该函数将十进制 int 整数转化为用字符串形式表示的二进制数。函数通过将传入的 int a 反复除 2 取余数的方式转化为二进制字符串。

3.2.5 函数 bool is_empty(const vector<set<string>>& table)

该函数通过遍历 table 的每一项是否为空以判断整张 table 是否为空。

3.2.6 函数 bool is_one_diff(const string& a, const string& b)

该函数通过遍历两个二进制数字符串，计数两个字符串的不同位的数目以判断两个字符串是否只有一位不同。

3.2.7 函数 string get_diff(string a, const string& b)

该函数通过遍历两个只有一位不同的字符串，找到不同位，并将其替换为'-'。

3.2.8 函数 bool is_prime_icl_min(const string &p, const string& minterm)

该函数输入一个质蕴涵项字符串，一个最小项字符串，通过遍历两个字符串，判断有质蕴涵项中非'-'位是否全与最小项字符串中相应位字符相同，以判断该质蕴涵项是否包含该最小项。

3.2.9 函数 vector<set<string>> comb_step_1(const vector<set<string>>& table, set<string>& primeset)

该函数用以完成 2.2 中的步骤③。代码

```
for (unsigned int i = 0; i < table.size() - 1; i++)
{
    set<string> ::iterator itj;
    int j = 0;
    for (itj = table[i].begin(); itj != table[i].end(); itj++)
    {
        string xj = *itj;
        set<string> ::iterator itk;
        int k = 0;
        for (itk = table[i+1].begin(); itk != table[i+1].end(); itk++)
        {
            string xk = *itk;
            if (is_one_diff(xj, xk))
            {
                newtable[i].insert(get_diff(xj, xk));
                checked[i][j] = 1;
                checked[i + 1][k] = 1;
```

```
            }
            k++;
        }
        j++;
    }
}
```

用以遍历 table 中相邻表项（'1'的个数相差为 1）中各二进制字符串并进行成对比较，利用函数 is_one_diff 确定是否一对字符串仅在一位上不同，若是，通过函数 get_diff 将不同位替换为'-'，储存在新表 newtable 中，并将这两项记为 checked（vector<vector<int>> checked 用以完成这项操作，它的各项初始值均为 0，若某项记为 checked，其对应项的值改为 1）。

代码

```
for (unsigned int i = 0; i < table.size(); i++)
    for (unsigned int j = 0; j < table[i].size(); j++)
        if (!checked[i][j])
        {
            set<string> ::iterator itk;
            int k = 0;
            for (itk = table[i].begin(); itk != table[i].end(); itk++)
            {
                string xk = *itk;
                if(k == j) primeset.insert(xk);
                k++;
            }
        }
```

将未被记为 checked 的项（即已确认为质蕴涵项的项）插入 set primet 中。

最后，该函数返回新表 newtable。

3.2.10 函数 void gen_prime()

该函数利用函数 comb_step_1 构造包含所有质蕴涵项的 vector prime。这里用了一个 set 容器 set<string> primet 用以临时储存质蕴涵项，以保证最终存入 prime 的质蕴涵项没有重复项。在 while 循环

```
while (!is_empty(table))
    table = comb_step_1(table, primet);
```

中，函数 comb_step_1 产生新表再次被放入函数 comb_step_1 中处理，直至新表为空，即所有质蕴涵项都已产生的情况为止。

3.2.11 函数 void comb_step_2(const vector<set<set<int>>>& petrick, const int& n, set<int> tree, set<set<int>>& rescombt)

该函数利用 Petrick's method 方法完成 2.2 中的步骤⑥。该函数基本上按照参考资料中的 Petrick 算法进行。该函数传入的 vector<set<set<int>>> petrick 是各项下标的集合的集合的组合，其第一维 set 存放的是用"或"连接的各项，第二位 set 中存放的是这些项中用"与"连接的各项。本函数对其中各项进行与运算。本函数还考虑了 petrick 的大小为 1 的情况，此时直接返回 petrick[0]。其中调用的函数见 3.2.14-3.2.16。

3.2.12 函数 gen_prime_comb()

该函数用于产生最终输出要求的质蕴涵项组合。

代码

```
int impsize = prime.size();
vector<vector<int>> primetable(impsize, vector<int>(mincount, 0));
for (int i = 0; i < impsize; i++)
    for (int j = 0; j < mincount; j++)
        primetable[i][j] = is_prime_icl_min(prime[i], binminterm[j]);
```

通过遍历函数 comb_step_1 产生的质蕴涵项，生成每个质蕴涵项是否包含各个最小项的表格，完成了 2.2 中的步骤⑤。

代码

```
for (int j = 0; j < mincount; ++j)
{
    set<int> x;
```

```
    for (unsigned int i = 0; i < prime.size(); ++i) if
(primetable[i][j] == true) x.insert(i);
    petrick.push_back(x);
}
```

通过遍历上述产生的表格产生包含各个最小项的质蕴涵编号的组合，即 Petrick 算法中各乘积项中的下标。

利用函数 comb_step_2 得到了所有能包含所有最小项的质蕴涵项组合的集合。

代码

```
unsigned int min = 50000;
set<set<int>> ::iterator it;
for (it = rescombt.begin(); it != rescombt.end(); it++)
{
    set<int> comb = *it;
    if (comb.size() < min)
        min = comb.size();
}
vector<set<int>> rescombs;
set<set<int>> ::iterator it1;
for (it1 = rescombt.begin(); it1 != rescombt.end(); it1++)
{
    set<int> comb = *it1;
    if (comb.size() == min)
        rescomb = comb;
}
```

用以寻找上述集合中包含最少项数质蕴涵项的组合，作为最优解。

3.2.13 函数 string gen_expr()

该函数利用函数 bin_to_expr，把最优解质蕴涵项组合中各质蕴涵项转化为逻辑表达式字符串的形式，并用或符号'|'进行拼接。最后利用代码

```
resexpr.pop_back();
```

将最后一个多余的'|'去除。

3.2.14 函数 bool is_pet_icl(const set<int>& s1, const set<int>& s2)

该函数用来判断 s1 是否能包含 s2（s1 与 s2 是"或"连接的项，包含是指类似于 X

可以包含 XY（X + XY = X））。

3.2.15 函数 set<set<int>> arr_pet(const set<set<int>>& p)

该函数用两层遍历由“或”连接的各项，调用函数 is_pet_icl 判断项之间的包含关系，并将被包含的项去除。

3.2.16 set<set<int>> mul_pet(const set<set<int>>& p1, const set<set<int>>& p2)

该函数对两个逻辑多项式作与。由于与有 XY · X = XY 的性质，与 set 容器插入相同项不变的性质相似，这里利用 set 容器的 insert 操作完成与运算。

4.测试

本程序利用课程提供的 simple_test.h 进行测试，测试函数为 int test_main()。

4.1 函数 expr_to_truthtable 的测试

```
CHECK_THROW(expr_to_truthtable(2, "A&B|C"), num_err);
```

用于测试输入变量个数小于表达式中变量个数的情况，应抛出异常。

```
CHECK_THROW(expr_to_truthtable(9, "A&B|C"), num_err);
```

用于测试输入变量个数大于 8 的情况，应抛出异常。

```
CHECK_THROW(expr_to_truthtable(1, ""), expr_err);
```

用于测试输入表达式为空串的情况，应抛出异常。

```
CHECK_THROW(expr_to_truthtable(1, "      "), expr_err);
```

用于测试输入表达式只有空格的情况，应抛出异常。

```
CHECK_THROW(expr_to_truthtable(4, "A&b|C"), expr_err);
```

用于测试输入表达式含有非法字符的情况，应抛出异常。

```
CHECK_THROW(expr_to_truthtable(4, "A&B||C"), expr_err);
CHECK_THROW(expr_to_truthtable(2, "&A|B"), expr_err);
```

用于测试输入表达式语法错误的情况，应抛出异常。

```
CHECK_THROW(expr_to_truthtable(3, "A|((B&(~C)))"), expr_err);
```

用于测试输入表达式括号不匹配的情况，应抛出异常。

```
CHECK_EQUAL(expr_to_truthtable(1, "A"), "10");
```

用于测试输入变量最小的正常情况。

```
CHECK_EQUAL(expr_to_truthtable(3, "~A|B&C"), "11010101");
```

用于测试正常情况。

```
CHECK_EQUAL(expr_to_truthtable(3, "~A  |  B& C"), "11010101");
```

用于测试输入表达式中含有空格和制表符的正常情况。

```
CHECK_EQUAL(expr_to_truthtable(3, "~(A|B)&C"), "00010000");
```

用于测试输入表达式中含有括号，改变优先级的正常情况。

```
CHECK_EQUAL(expr_to_truthtable(4, "B|C"),"1111110011111100");
```

用于测试输入表达式中变量个数小于输入变量个数的正常情况。

```
CHECK_EQUAL(expr_to_truthtable(8, "A&B&C&D|B&E&G|~C&~E&F&H"),
"1100110011001100100011110000111111001100110011001000000000000000100000000000000010001111000011111000000000000000010000000000000001100110011001100100000000000000011001100110011001000000000000000100000000000000001000000000000000010000000000000001000000000000000010000000000000001000000000000000");
```

用于测试输入变量最多（输入变量为8个，输出256位二进制字符）的正常情况。

4.2 函数 truthtable_to_expr 的测试

```
CHECK_THROW(truthtable_to_expr(""), truth_table_err);
```

用于测试输入字符串为空串的情况，应抛出异常。

```
CHECK_THROW(truthtable_to_expr("    "), truth_table_err);
```

用于测试输入字符串只有空格的情况，应抛出异常。

```
CHECK_THROW(truthtable_to_expr("12100000"), truth_table_err);
```

用于测试输入字符串含有非法字符的情况，应抛出异常。

```
CHECK_THROW(truthtable_to_expr("111100000"), truth_table_err);
```

用于测试输入字符串长度不为 2 的 1 至 8 整数次方的情况，应抛出异常。

```
CHECK_EQUAL(truthtable_to_expr("1111"), "1");
```

用于测试输入字符串字符全为 1 的正常情况。

```
CHECK_EQUAL(truthtable_to_expr("0000"), "0");
```

用于测试输入字符串字符全为 0 的正常情况。

```
CHECK_EQUAL(truthtable_to_expr("01"), "~A");
```

用于测试输入字符串长度最小的正常情况。

```
CHECK_EQUAL(truthtable_to_expr("0011"), "~B");
CHECK_EQUAL(truthtable_to_expr("11010101"), "~A|B&C");
CHECK_EQUAL(truthtable_to_expr("0100000001000001"),
"~A&B&C|~A&~B&~C&~D");
```

用于测试正常情况。

```
CHECK_EQUAL(truthtable_to_expr("1100110011001100100011110000111111001
1001100110010000000000000001000000000000000100011110000111110000000000
0000001000000000000000011001100110011001000000000000000011001100110011
0100000000000000010000000000000001000000000000000100000000000000010000
000000000000"), "A&B&C&D|B&E&G|~C&~E&F&H");
```

用于测试输入字符串长度最大（输入 256 位真值表字符串，输出含有 8 个变量的逻辑表达式字符串）的正常情况。

```
CHECK_EQUAL(truthtable_to_expr("0101111111111111111111111111111111111
1111111111111111111111111111111111111111111111111111111111111111111111
1111111111111111111111111111111111111111111111111111111111111111111111
1111111111111111111111111111111111111111111111111111111111111111111111
1111111111111111111111111111111111111111111111111111111111111111111111
111111111111"), "~A|~C|~D|~E|~F|~G|~H");
```

用于测试用于测试输入字符串长度最大且含有 254 个 1 的正常情况。

5.心得体会

这次的期中 Mini Project 是我初学 C++之后做的第一个规模比较大的 project，让我对于 C++语言以及软件工程都有了新的认识。我现在简单谈谈我这次写 project 的心得体会。

首先是学会了 C++里面的 regex 和 set 两个库。以前在学习 Python 的时候，也涉及到类似于函数 expr_to_truthtable 的表达式处理的情况，当时利用 Python 语

言灵活的字符串处理与正则表达式功能解决了问题。这次看到题目，自然而然地就想到了也通过正则表达式来做，在网上搜索得知 C++也有正则表达式的库 regex，我学习了库里相关的函数，掌握了在 C++内使用正则表达式的技巧。在编写函数 truthtable_to_expr 的时候，我发现 C++除了 vector 容器，还有 set 这种不包含重复元素的容器，能够很好地运用在这个函数之中。因此，我也自学了这个 set 相关的知识。

然后是学会了一些优化算法的技巧。在写函数 truthtable_to_expr 的过程中，我主要采取了两项措施来优化运算时间。第一个是 comb_step_2 这个函数，我本来是用暴力求解的方法，用二叉树把每个质蕴涵项包括/不包括的组合遍历，选出能够包含所有最小项的组合，代码如下：

```
void comb_step_2_1(vector<vector<int>>& chart, int n, set<int> tree,
set<set<int>>& rescombt)
{
    int size = chart[0].size();
    vector<int> checked(size, 0);
    if (n == chart.size())
    {
        set<int> ::iterator it;
        for (it = tree.begin(); it != tree.end(); it++)
        {
            int x = *it;
            for (int i = 0; i < size; i++)
                if (chart[x][i] == 1)
                    checked[i] = 1;
        }
        int flag = 0;
        for (int i = 0; i < size; i++)
            if (checked[i] == 0)
                flag = 1;
        if (flag == 0) rescombt.insert(tree);
        return;
    }
    else
    {
```

```
            tree.insert(n);
            comb_step_2_1(chart, n + 1, tree, rescombt);
            tree.erase(n);
            comb_step_2_1(chart, n + 1, tree, rescombt);
        }
    }
```

后来看到参考资料上面的 Petrick 算法，重新编写了这个函数。第二个优化的地方是 table 成员，我首先使用的是 vector<<vector>int>，后面发现变量数目增加时，vector<vector>中有大量重复项，当改成 vector<<set>int>后解决了这个问题。这两个优化的地方显著地减少了运算时间。

经过半个学期的学期和实际编程，我发现了很多 C++语言的闪光点，它含有诸多实用的功能，如 string，vector 和 set 等。这次的 project 也让我认识到 C++和软件工程和我们微电子专业的关系。手工难以完成的逻辑表达式和真值表的互相转化可以利用代码来在短时间内实现。这让我进一步坚定了要学好这门语言和软件工程的决心。希望我能继续学到更多有用的 C++语言与软件工程知识，并积极实践应用。