# Mini Project doc

**18300750070 黄听**

目录

[Mini Project doc 1](#_Toc69065423)

[1.概述 3](#_Toc69065424)

[2.设计思路与算法简介 4](#_Toc69065425)

[2.1 函数expr\_to\_truthtable的设计思路与算法简介 4](#_Toc69065426)

[2.2函数truthtable\_to\_expr的设计思路与算法简介 4](#_Toc69065427)

[3.类与函数的说明 5](#_Toc69065428)

[3.1与函数expr\_to\_truthtable相关的函数与类 5](#_Toc69065429)

[3.2与函数truthtable\_to\_expr相关的函数与类 9](#_Toc69065430)

[4.测试 16](#_Toc69065431)

[4.1函数expr\_to\_truthtable的测试 16](#_Toc69065432)

[4.2函数truthtable\_to\_expr的测试 17](#_Toc69065433)

[5.心得体会 18](#_Toc69065434)

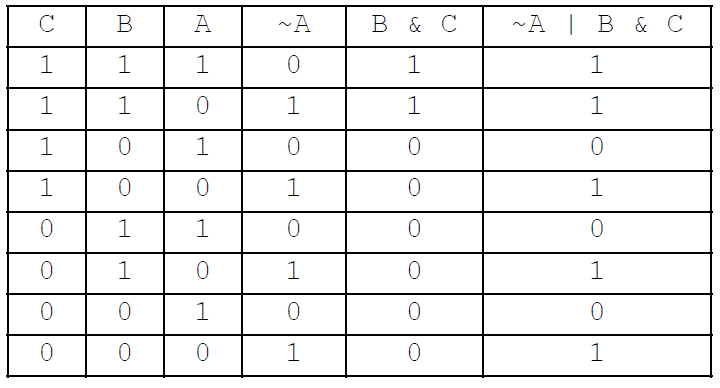
## 1.概述

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"



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。

1. 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\_constants::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"), "1100110011001100100011110000111111001100110011001000000000000000100000000000000010001111000011111000000000000000100000000000000011001100110011001000000000000000110011001100110010000000000000001000000000000000100000000000000010000000000000001000000000000000");

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("1100110011001100100011110000111111001100110011001000000000000000100000000000000010001111000011111000000000000000100000000000000011001100110011001000000000000000110011001100110010000000000000001000000000000000100000000000000010000000000000001000000000000000"), "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("0101111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111"), "~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。