2022 NYCU Computer Aided Design - Homework 1 Report

IEE 陳冠瑋 310510221 / Oct. 25, 2022

1 Read input file

將 input variable, on set 與 don't care set 讀入到相應的 global parameter 中。

```
// Read Input
while(getline(fin, line)){
    istringstream sin(line);
    sin >> str;
    // if(debug) cout << str << endl;
    if(str == ".i"){
        getline(fin, line);
        stringstream var(line);
        var >> inVar;
    }
    if(str == ".m"){
        getline(fin, line);
        stringstream onset_in(line);
        while(onset_in >> onset){
            onSet.push_back(stoi(onset)); // string to int
            // if(debug) if(debug) cout << stoi(onset) << endl;
        }
    }
    if(str == ".d"){
        getline(fin, line);
        stringstream dcset_in(line);
        while(dcset_in >> dcset){
            dcSet.push_back(stoi(dcset)); // string to int
            // if(debug) cout << stoi(dcset) << endl;
        }
    }
}
```

2 Algorithm Flow

我主要將 Quine-McCluskey 演算法拆解為 6 大步驟，其中前 3 步是演算法的核心，分別是：

- **Generate Table**
  - 生成 Implicant Table
- **Prime Implicant Generation (PI Generation)**
  - 從 Implicant Table 找出所有的 Prime Implicants
- **Column Covering**
  - 將 Prime Implicants 與 On set 建表，把所有 Prime Implicat 用打叉表示，並找出 Minimum Number of Prime Implicant 和 Minimum Cover

後 3 步主要處理輸出的排序與計算 literal 的數量。

- **Sort Prime Implicants (Sort PI)**
  - 將找到的所有 Prime Implicants 做排序
- **Sort Essential Prime Implicants (Sort Essential PI)**
  - 將找到的 Minimum Number of Prime Implicant 的所有 Prime Implicant 做排序
- **Count Literal**
  - 計算所有的 Minimum Number of Prime Implicant 有多少 0 與 1

```
if(debug) cout << "===== Quine McCluskey Algorithm ====" << endl << endl;
if(debug) cout << "========== Generate Table ==========" << endl;
imp = QM.gen_implication_table();

/////////////////////////////////////////////////////////////////////////
if(debug) cout << "=========== PI Generation ==========" << endl;
QM.PI_generation();

/////////////////////////////////////////////////////////////////////////
if(debug) cout << "========== Column Covering =========" << endl;
QM.column_covering();

/////////////////////////////////////////////////////////////////////////
if(debug) cout << "============== Sort PI =============" << endl;
if(debug) cout << "p = " << pi.size() << endl;
sorted_pi = sort_result(pi);

/////////////////////////////////////////////////////////////////////////
if(debug) cout << "========= Sort Essential PI ========" << endl;
if(debug) cout << "mc = " << essential_pi.size() << endl;
sorted_essential_pi = sort_result(essential_pi);

/////////////////////////////////////////////////////////////////////////
if(debug) cout << "=========== Count Literal ==========" << endl;
literal = count_literal(sorted_essential_pi);
```

3 Generate Table

先定義一個 dec2bin()函式，將十進位輸入的 on set 與 don't care set 轉換成二進位表示。

```
// Covert dec(int) to binary(string) //
string dec2bin(int dec){
    // if(debug) cout << "dec: " << dec << endl;
    string str_bin;

    for(int i = 0; i < inVar; i++){
        if(dec == 0){
            str_bin += '0';
        }
        else{
            str_bin += '0' + dec % 2;
            dec /= 2;
        }
    }
    reverse(str_bin.begin(), str_bin.end());
    // printf("str_bin: %s\n", str_bin.c_str());
    return str_bin;
}
```

將 on set 與 don't care set 送進 imp(1-D vector)裡。

```
// Get binary implicant from onSet and dcSet, and generate implication table //
vector<string> gen_implication_table(){
    vector<string> imp; // implicant
    for(int i = 0; i < onSet.size(); i++){
        imp.push_back(dec2bin(onSet[i])); // push onSet into implicant table
    }
    for(int i = 0; i < dcSet.size(); i++){
        imp.push_back(dec2bin(dcSet[i]));
    }

    // Print implicant
    if(debug){
        for(int i = 0; i < imp.size(); i++){
            cout << imp[i] << endl;
        }
    }

    return imp;
}
```

顯示 implicant (imp)裡面有的元素。

```
========== Generate Table ==========
0100
0101
0110
1000
1001
1010
1101
0000
0111
1111
```

4 PI Generation

PI Generation 會遞迴地做到 imp 都被標註(拿完)為止。主要有兩步驟：Grouping 與 Merging。

```
// Prime Implicant Generation //
void PI_generation(){
    if(debug) cout << "####### PI Generation Start #######" << endl;
    if(debug) cout << "number of input imp: " << imp.size() << endl << endl;
    vector< vector<string> > group;
    int round = 1;

    while(!imp.empty()){
        if(debug) cout << "////////////// Round " << round << " /////////////" << endl << endl;

        ////////////////////////////////////////////////////////////////////////////////
        // Grouping
        if(debug) cout << "============ Grouping =============" << endl;
        group = group_imp();
        ////////////////////////////////////////////////////////////////////////////////
        // Merge and Generate PI
        if(debug) cout << "============= Merging =============" << endl;
        find_PI(group);
        ////////////////////////////////////////////////////////////////////////////////
        round++;

        if(debug){
            cout << "Prime Implicants (pi): " << endl;
            for(int i = 0; i < pi.size(); i++){
                cout << pi[i] << endl;
            }
            cout << endl;
            cout << "Implicants (imp): " << endl;
            for(int i = 0; i < imp.size(); i++){
                cout << imp[i] << endl;
            }
            cout << endl;
        }
    }
    if(debug) cout << "####### PI Generation Finish ######" << endl << endl;
}
```

Grouping 是 Quine-McCluskey 的關鍵步驟，若有事先做好分群，可以大幅降低後面做 Merging 的次數。Grouping 會先利用每個 implicants 1 的個數來進行分群與排序，並將分群後的結果放到一個 group (2-D vecotr)中，並且將不同群裡面的 implicant 做排序(非必要，只是為了讓 output 的順序與答案相近)。

```
// Grouping //
vector< vector<string> > group_imp(){
    vector<vector <string> > group;
    group.resize(inVar + 1); // 0, 1, 2, ..., inVar
    // Example: group[groupidx]
    // group[0] = {"0000"}
    // group[1] = {"0001", "0010", "0100", "1000"}
    // group[2] = {"0011", "0101", "0110", "1001", "1010", "1100"}
    // ...
    // group[4] = {"1111"}

    vector<string> tmp_imp;
    int group_idx;
    int curr_imp;

    // Grouping
    for(int i = 0; i < imp.size(); i++){
        group_idx = 0;
        for(int j = 0; j < inVar; j++){
            if(imp[i][j] == '1'){
                group_idx++;
            }
        }
        group[group_idx].push_back(imp[i]);
    }

    // Sorting
    for(int i = 0; i < group.size(); i++){
        for(int j = 0; j < group[i].size(); j++){
            sort(group[i].begin(), group[i].end());
        }
    }
```

顯示 Grouping 之後的結果：

```
============= Grouping ==============
[Group 0]
0000
---------
[Group 1]
0100
1000
---------
[Group 2]
0101
0110
1001
1010
---------
[Group 3]
0111
1101
---------
[Group 4]
1111
---------
```

接者就是做 Merging，Merging 的用意是將相鄰的 Group 的 Implicants 一一做比對，看使否只相差一個 bit，如果相差 1 個 bit，它們彼此就可以 merge，並且標註 ”x” 當作標記，表示不是 Prime Implicant。因此就先宣告一個 prime_flag vector (1-D vector)來記錄比對過後的 Implicants 是否為 Prime Implicant，一開始先將 prime_flag 初始為 true，如果遇到可以 merge 的情況，就將 prime_flag 設為 true 代表這兩個比對的 implicant 可以 merge，並且不是 Prime Implicants，並且將可以 merge 的 Implicant 先暫時放到一個 tmp_imp (1-D vector)中，因為等等要清空原本的 imp vector，並更新新的 imp vector。

```
// Generate Prime Implicant //
void find_PI(vector< vector<string>> group){
    string merge_imp;
    vector<bool> prime_flag(imp.size(), true); // 1: can merge(IMP), 0: cannot merge(PI)
    vector<string> tmp_imp;
    tmp_imp.clear();

    int cur_group_pos = 0;
    int nxt_group_pos = 0;

    if(debug){
        cout << "Initialize PI flag" << endl;
        for(int i = 0; i < imp.size(); i++){
            cout << prime_flag[i] << " ";
        }
        cout << endl << endl;
    }

    if(debug) cout << "after merge" << endl;
    for(int i = 0; i < inVar; i++){ // group index
        if(debug) cout << "[Group " << i << "]" << endl;
        nxt_group_pos += group[i].size();
        // if(debug) cout << "current pos: " << cur_group_pos << endl;
        for(int j = 0; j < group[i].size(); j++){
            for(int k = 0; k < group[i+1].size(); k++){
                merge_imp = merge(group[i][j], group[i+1][k]);
                if(merge_imp != "x"){ // is not a prime implicant
                    // if merg_imp is not "x", it means that it can merge, so it is not a prime implicant

                    if(debug) cout << merge_imp << endl;
                    prime_flag[cur_group_pos+j] = false;
                    prime_flag[nxt_group_pos+k] = false;

                    tmp_imp.push_back(merge_imp);
                }
            }
        }
        cur_group_pos += group[i].size();
        if(debug) cout << "---------" << endl;
    }

    if(debug) cout << endl;

    if(debug){
        cout << "PI flag" << endl;
        for(int i = 0; i < imp.size(); i++){
            cout << prime_flag[i] << " ";
        }
        cout << endl;
        cout << "-------------------------------" << endl << endl;
    }
```

接著將被標註的 Implicant (prime_flag[i] == true)，放入 pi (1-D vector)中，其中 pi 代表 Prime Implicant。然後，清空原本的 imp vector，並將 tmp_imp 刪除重複的 imp，最後把 tmp_imp 複製到 imp 中。

```
// find prime implicnat
for(int i = 0; i < imp.size(); i++){
    if(prime_flag[i] == true){

        pi.push_back(imp[i]);
    }
}

// clear imp vector
imp.clear();

// remove duplicates elements (tmp_imp)
sort(tmp_imp.begin(), tmp_imp.end());
tmp_imp.erase(unique(tmp_imp.begin(), tmp_imp.end()), tmp_imp.end());

// push new imp
for(int i = 0; i < tmp_imp.size(); i++){
    imp.push_back(tmp_imp[i]);
}
```

第一輪 merging 後的結果：

```
============= Merging =============
Initialize PI flag
1 1 1 1 1 1 1 1 1

after merge
[Group 0]
0-00
-000
---------
[Group 1]
010-
01-0
100-
10-0
---------
[Group 2]
01-1
-101
011-
1-01
---------
[Group 3]
-111
11-1
---------

PI flag
0 0 0 0 0 0 0 0 0
------------------------------

Prime Implicants (pi):

Implicants (imp):
-000
-101
-111
0-00
01-0
01-1
010-
011-
1-01
10-0
```

顯示整個 PI Generation 的過程：

```
=========== PI Generation ==========
####### PI Generation Start #######
number of input imp: 10

////////////// Round 1 /////////////

============ Grouping ============
[Group 0]
0000
---------
[Group 1]
0100
1000
---------
[Group 2]
0101
0110
1001
1010
---------
[Group 3]
0111
1101
---------
[Group 4]
1111
---------
============= Merging ============
Initialize PI flag
1 1 1 1 1 1 1 1 1

after merge
[Group 0]
0-00
-000
---------
[Group 1]
010-
01-0
100-
10-0
---------
[Group 2]
01-1
-101
011-
1-01
---------
[Group 3]
-111
11-1
---------

PI flag
0 0 0 0 0 0 0 0 0
------------------------------

Prime Implicants (pi):

Implicants (imp):
-000
-101
-111
0-00
01-0
01-1
010-
011-
1-01
10-0
100-
11-1
```

```
////////////// Round 2 /////////////

============ Grouping ============
[Group 0]
-000
0-00
---------
[Group 1]
01-0
010-
10-0
100-
---------
[Group 2]
-101
01-1
011-
1-01
---------
[Group 3]
-111
11-1
---------
[Group 4]
---------
============= Merging ============
Initialize PI flag
1 1 1 1 1 1 1 1 1 1 1 1

after merge
[Group 0]
---------
[Group 1]
01--
01--
---------
[Group 2]
-1-1
-1-1
---------
[Group 3]
---------

PI flag
1 1 0 0 1 1 0 0 0 1 0 0
------------------------------

Prime Implicants (pi):
-000
0-00
10-0
100-
1-01

Implicants (imp):
-1-1
01--
```

```
////////////// Round 3 /////////////

============ Grouping ============
[Group 0]
---------
[Group 1]
01--
---------
[Group 2]
-1-1
---------
[Group 3]
---------
[Group 4]
---------
============= Merging ============
Initialize PI flag
1 1

after merge
[Group 0]
---------
[Group 1]
---------
[Group 2]
---------
[Group 3]
---------

PI flag
1 1
------------------------------

Prime Implicants (pi):
-000
0-00
10-0
100-
1-01
01--
-1-1

Implicants (imp):

####### PI Generation Finish ######
```

5 Column Coverage

先建立一個 column coverage table (2-D vector)，橫軸是 on set，縱軸是 Prime Implicants，並將整個 table 初始化為 false。

```
// Column Covering //
void column_covering(){
    vector< vector<bool> > column_coverage_table(onSet.size()); // 2-D table
    vector< vector <int> > pi_dec(pi.size());
    // ex:
    // pi[0] = 0-00   / pi_dec[0] = {0,4}
    // pi[1] = -000   / pi_dec[0] = {0,8}
    // ...
    // pi[6] = -1-1  / pi_dec[0] = {5,7,13,15}
    vector<int> dc_value_set;

    int on_dec;
    int dc_dec;
    int exp;


    // Initialize column coverage table
    for(int i = 0; i < onSet.size(); i++){
        for(int j = 0; j < pi.size(); j++){
            column_coverage_table[i].push_back(false);
        }
    }

    if(debug){
        cout << "-Initialize Column Coveraging Table-" << endl;
        cout << "       ";
        for(int i = 0; i < onSet.size(); i++){
            if(onSet[i] > 9){
                cout << onSet[i] << " ";
            }
            else{
                cout << " " << onSet[i] << " ";
            }
        }
        cout << endl << "-----------------------------------" << endl;
        for(int j = 0; j < pi.size(); j++){
            cout << pi[j] << " | ";
            for(int i = 0; i < onSet.size(); i++){
                cout << column_coverage_table[i][j] << "  ";
            }
            cout << endl;
        }
        cout << endl;
    }
```

接者把 Prime Implicants 的字串讀進 column coverage table 中，其中必須將 Prime Implicant 轉成十進制，因為有考慮 don't care，所以也必須將 don't care 有可能的值都一併考慮進去。因此，我將所有 don't care 有可能的 2 進制結果做排列(permutation)並加上原本的 on set 值，並移除掉重複的結果。並將計算出來的十進位結果，放入 pi_dec (2-D vector)中

```
// Read PI string (reverse)
for(int i = 0; i < pi.size(); i++){
    on_dec = 0;
    exp = 0;

    dc_value_set.clear();
    dc_value_set.push_back(0);
    for(int j = inVar - 1; j >= 0; j--){
        dc_dec = 0;
        if(pi[i][j] == '1'){
            on_dec += pow(2, exp);
        }
        if(pi[i][j] == '-'){
            dc_dec = pow(2, exp);

            // Permuation all possible don't care value
            int dc_value_set_size = dc_value_set.size();
            for(int k = 0; k < dc_value_set_size; k++){
                dc_value_set.push_back(dc_value_set[k] + dc_dec);
            }
        }
        exp++;
    }

    // Show DC possible value
    // if(debug){
    //   cout << "DC possible value: " << endl;
    //   for(int i = 0; i < dc_value_set.size(); i++){
    //       cout << dc_value_set[i] << " ";
    //   }
    //   cout << endl;
    // }

    // remove same element
    sort(dc_value_set.begin(), dc_value_set.end());
    dc_value_set.erase(unique(dc_value_set.begin(), dc_value_set.end()), dc_value_set.end());
// Push PI possible dec value
for(int j = 0; j < dc_value_set.size(); j++){
    pi_dec[i].push_back(on_dec + dc_value_set[j]);
}

// Print possible PI
if(debug){
    cout << "PI possible value" << endl;
    cout << pi[i] << endl;
    for(int j = 0; j < pi_dec[i].size(); j++){
        cout << pi_dec[i][j] << " ";
    }
    cout << endl << "-----------------------------------" << endl;
}
```

顯示 Prime Implicant 所有可能的值：

```
PI possible value
-000
0 8
-----------------------------------
PI possible value
0-00
0 4
-----------------------------------
PI possible value
10-0
8 10
-----------------------------------
PI possible value
100-
8 9
-----------------------------------
PI possible value
1-01
9 13
-----------------------------------
PI possible value
01--
4 5 6 7
-----------------------------------
PI possible value
-1-1
5 7 13 15
-----------------------------------
```

將 column coverage table 中，pi_dec 與 on set 相等的地方打叉，並將 column coverage table 顯示出來。

```cpp
// modify table
for(int k = 0; k < onSet.size(); k++){
    for(int i = 0; i < pi.size(); i++){
        for(int j = 0; j < pi_dec[i].size(); j++){
            if(pi_dec[i][j] == onSet[k]){
                column_coverage_table[k][i] = 'x';
            }
        }
    }
}

if(debug){
    cout << "     - Column Coveraging Table -     " << endl;
    cout << "       ";
    for(int i = 0; i < onSet.size(); i++){
        if(onSet[i] > 9){
            cout << onSet[i] << " ";
        }
        else{
            cout << " " << onSet[i] << " ";
        }
    }
    cout << endl << "-----------------------------------" << endl;
    for(int j = 0; j < pi.size(); j++){
        cout << pi[j] << " | ";
        for(int i = 0; i < onSet.size(); i++){
            cout << column_coverage_table[i][j] << "  ";
        }
        cout << endl;
    }
}

find_essential_PI(column_coverage_table, pi_dec);
```

顯示 Column Coverage Table：

```
-----------------------------------
   - Column Coveraging Table -
     4  5  6  8  9 10 13
-----------------------------------
-000 | 0  0  0  1  0  0  0
0-00 | 1  0  0  0  0  0  0
10-0 | 0  0  0  1  0  1  0
100- | 0  0  0  1  1  0  0
1-01 | 0  0  0  0  1  0  1
01-- | 1  1  1  0  0  0  0
-1-1 | 0  1  0  0  0  0  1
```

接著，就是尋找 Essential Prime Implicants。

```
find_essential_PI(column_coverage_table, pi_dec);
```

尋找 Essential Prime Implicants 是演算法中比較複雜的一部份。

首先，我對 Table 的橫軸旁邊先宣告了兩個 vector，col_sel_flag 與 col_pi_cnt，分別用來記錄這個 column 是否已經選擇過，還有不同的 Prime Implicant 對應剩餘的 column 的數量。然後，對縱軸宣告 essential_pi_flag，來記錄這個 Prime Implicants 是否為 Essential Prime Implicant。並將這些 flag 與 counter 初始化為 0。

```
void find_essential_PI(vector< vector<bool> > column_coverage_table, vector< vector <int> > pi_dec){
    vector<bool> essential_pi_flag;
    vector<bool> col_sel_flag;
    vector<int> col_pi_cnt;

    ///////////////////////////////////////////////////////////////////////////
    // Step 0: Initialize

    // Initialize essential_pi_flag
    for(int i = 0; i < pi.size(); i++){
        essential_pi_flag.push_back(false);
    }

    // Initialize col_sel_flag
    for(int i = 0; i < onSet.size(); i++){
        col_sel_flag.push_back(false);
    }

    // Initialize col_pi_cnt
    for(int i = 0; i < onSet.size(); i++){
        col_pi_cnt.push_back(0);
    }
```

第二步就正式進入到尋找 Essential Prime Implicant 的步驟。我先找整個 Column Coverage Table，每個個別的 column 上面如果只有一個'x'，代表這個 row 對應的 Prime Implicant 是 Essential Prime Implicant，因此就先將對應的 essential_pi_flag 標記成 ture。然後標記剛剛找到的所有 Essential Prime Implicants 對應的所有 column，將這些 column 標記成已經 cover 到，把對應的 col_sel_flag 設為 true。

```
////////////////////////////////////////////////////////////////////////////////
// Step 2: Find essential prime implicant.

// If column has a single 'x', then the implicant associated with the row is essential.
for(int i = 0; i < onSet.size(); i++){ // Column index
    for(int j = 0; j < pi.size(); j++){ // Row index
        if(column_coverage_table[i][j] == true){
            col_pi_cnt[i] += 1;
        }
    }
    if(col_pi_cnt[i] == 1){
        for(int j = 0; j < pi.size(); j++){
            if(column_coverage_table[i][j] == true){
                essential_pi_flag[j] = true;
            }
        }
    }
}

// Push Essential PI
for(int i = 0; i < essential_pi_flag.size(); i++){
    if(essential_pi_flag[i] == true){
        essential_pi.push_back(pi[i]);
    }
}

// Show col_pi_cnt
if(debug){
    cout << "-----------------------------------" << endl << "Total: ";
    for(int i = 0; i < onSet.size(); i++){
        cout << col_pi_cnt[i] << "  ";
    }
    cout << endl;
}

// Show Essential PI
if(debug){
    cout << "-----------------------------------" << endl;
    cout << "Essential PI: " << endl;
    for(int i = 0; i < essential_pi.size(); i++){
        cout << essential_pi[i] << endl;
    }
    cout << endl;
}
```

```
// Eliminate all columns coveraged by essential primes
for(int i = 0; i < onSet.size(); i++){ // Column index
    for(int j = 0; j < pi.size(); j++){ // Row index
        if(essential_pi_flag[j] == true){
            if(column_coverage_table[i][j] == true){
                col_sel_flag[i] = true;
            }
        }
    }
}

// Show col_sel_flag
if(debug){
    cout << "-----------------------------------" << endl;
    cout << "Column selected: " << endl;
    for(int i = 0; i < col_sel_flag.size(); i++){
        cout << col_sel_flag[i] << " ";
    }
    cout << endl << endl;
}
```

顯示 Column Coverage Table, col_pi_cnt, essential_PI 與 col_sel_flag：

```
--------------------------------
   - Column Coveraging Table -
      4  5  6  8  9 10 13
--------------------------------
-000 | 0  0  0  1  0  0  0
0-00 | 1  0  0  0  0  0  0
10-0 | 0  0  0  1  0  1  0
100- | 0  0  0  1  1  0  0
1-01 | 0  0  0  0  1  0  1
01-- | 1  1  1  0  0  0  0
-1-1 | 0  1  0  0  0  0  1
--------------------------------
Total: 2  2  1  3  2  1  2
--------------------------------
Essential PI:
10-0
01--

--------------------------------
Column selected:
1 1 1 1 0 1 0
```

再來，就是要尋找 minimum set of row that cover the remaining columns。

先計算剩下沒被 cover 到的 column 的數量，並記錄在 remain_column 這個變數中

```
//////////////////////////////////////////////////////////////////////////////
// Step 3: Find minimum set of row that cover the remaining columns
int remain_column_cnt;
int max_cover;
int round_min_coverage;
bool selected;
vector<int> column_coverage_cnt;

round_min_coverage = 1;


remain_column_cnt = 0;
for(int i = 0; i < col_sel_flag.size(); i++){
    if(col_sel_flag[i] == false){
        remain_column_cnt++;
    }
}
```

如果剩餘的 column 數大於 0 的話，就要持續做這個步驟。

首先，因為作業題目希望 literal 最小，所以先從左到右，然後「從下到上」的方式掃描整個 column coverage table。如果 col_sel_flag 是 false (代表這個 column 沒被選到)，那就開始從下到上計算它每個非 Essential Prime Implicants 與 column coverage table 打”x”的地方的 column coverage，並將最多的 column cover 數量記錄下來，放到 max_cover 變數中。

然後先從左到右，從下到上去看哪個 Prime Implicant 可以有最多的 column coverage，並將這個 Prime Implicant 標記成 Essential Prime Implicant (雖然它不是，但方便實作 XD)，如果有找到，就把這個 Prime Implicant 加到 essential_pi 這個 vector 中，並將這個 Essential Prime Implicant 對應有的 column 標記起來，代表已經 cover 到，並且 break 出這個迴圈並重新迭代。

```
while(remain_column_cnt > 0){

    if(debug) cout << "---- Round (min coverage): " << round_min_coverage << " ----" << endl << endl;

    max_cover = 0;
    for(int i = 0; i < pi.size(); i++){
        column_coverage_cnt.push_back(0);
    }

    // Botton-Up (count which pi cover the most column)
    for(int i = 0; i < col_sel_flag.size(); i++){
        if(col_sel_flag[i] == false){
            for(int j = pi.size()-1; j >= 0; j--){
                if(essential_pi_flag[j] == false && column_coverage_table[i][j] == true){
                    column_coverage_cnt[j]++;
                }
                if(column_coverage_cnt[j] > max_cover){
                    max_cover = column_coverage_cnt[j];
                }
            }
        }
    }

    // Show column coverage cnt
    if(debug){
        cout << "Column coverage cnt: " << endl;
        for(int i = 0; i < pi.size(); i++){
            cout << pi[i] << ": " << column_coverage_cnt[i] << endl;
        }
        cout << endl;
    }


    selected = false;
    // Set essential pi
    for(int i = 0; i < col_sel_flag.size(); i++){
        if(selected == true){
            break;
        }
        if(col_sel_flag[i] == false){
            for(int j = pi.size()-1; j >= 0; j--){
                // Set essential_pi flag & push essential_pi
                if(essential_pi_flag[j] == false && column_coverage_cnt[j] == max_cover){
                    selected = true;
                    essential_pi_flag[j] = true;
                    essential_pi.push_back(pi[j]);
                    for(int k = 0; k < col_sel_flag.size(); k++){
                        if(column_coverage_table[k][j] == true){
                            col_sel_flag[k] = true;
                        }
                    }
                    break;
                }

            }
        }
    }
```

```
// Show Essential PI
if(debug){
    cout << "-----------------------------------" << endl;
    cout << "Essential PI: " << endl;
    for(int i = 0; i < essential_pi.size(); i++){
        cout << essential_pi[i] << endl;
    }
    cout << endl;
}


// Show col_sel_flag
if(debug){
    cout << "-----------------------------------" << endl;
    cout << "Column selected: " << endl;
    for(int i = 0; i < col_sel_flag.size(); i++){
        cout << col_sel_flag[i] << " ";
    }
    cout << endl;
}

remain_column_cnt = 0;
for(int i = 0; i < col_sel_flag.size(); i++){
    if(col_sel_flag[i] == false){
        remain_column_cnt++;
    }
}

round_min_coverage++;
```

顯示 Column Coverage 的過程：

```
---- Round (min coverage): 1 ----

Column coverage cnt:
-000: 0
0-00: 0
10-0: 0
100-: 1
1-01: 2
01--: 0
-1-1: 1

-----------------------------------
Essential PI:
10-0
01--
1-01

-----------------------------------
Column selected:
1 1 1 1 1 1 1
```

6 Sort PI & Sort Essential PI

上面前三個步驟就已經完成 Quine-McCluskey 演算法最主要的步驟。接下來就是要對 Prime Implicant 與 Minimum Covering 的結果作排序。

```
// Sort Result //
vector<string> sort_result(vector<string> imp){
    vector<string> sorted_imp;
    sort(imp.begin(), imp.end());

    for(int i = 0; i < imp.size(); i++){
        sorted_imp.push_back(imp[i]);
    }
    // Show sorted result
    if(debug){
        for(int i = 0; i < sorted_imp.size(); i++){
            cout << sorted_imp[i] << endl;
        }
        cout << endl;
    }
    return sorted_imp;
}
```

再來就是計算 Minimum number or prime implicant 的 literal (‘0’與’1’的個數)。

```
// Count Literal //
int count_literal(vector<string> str){
    int literal = 0;
    for(int i = 0; i < str.size(); i++){
        for(int j = 0; j < str[i].size(); j++){
            if(str[i][j] == '0' || str[i][j] == '1'){
                literal++;
            }
        }
    }

    if(debug) cout << "literal: " << literal << endl;

    return literal;
}
```

顯示 Sort PI, Sort Essential PI 與 Literal 的結果：

```
============== Sort PI =============
p = 7
-000
-1-1
0-00
01--
1-01
10-0
100-

========= Sort Essential PI ========
mc = 3
01--
1-01
10-0

=========== Count Literal ==========
literal: 8
```

7 Write Output file

最後就是把題目要求的結果寫入到 output file。

```cpp
// Write Output File //
void write_ouput(ofstream &fout, vector<string> pi, vector<string> essential_pi, int literal){
    int pi_num = 0;

    fout << ".p " << pi.size() << endl;
    for(int i = 0; i < pi.size(); i++){
        pi_num ++;
        fout << pi[i] << endl;
        if(pi_num >= 15){
            break;
        }
    }

    fout << endl;

    fout << ".mc " << essential_pi.size() << endl;
    for(int i = 0; i < essential_pi.size(); i++){
        fout << essential_pi[i] << endl;
    }
    fout << "literal=" << literal;
}
```