Quine-McCluskey Method Report

312510239 王則惟

首先第一部分會先透過 inputfile 的 function 將 input 檔案吃進來，並透過 Decimal_To_Binary 的 function，將 input 檔案裡的十進位數字改成二進位，並分別存到 on_set 和 don't_care 裡，並且將兩者也都存入 all_set 中，以便後續查找 prime implicant。

第二部分就是透過 find_prime_implicants 的 function 來查找 prime_implicants，我是使用兩個 for 迴圈來查找所有 all_set 裡的值，並且使用 while 迴圈包起來直到 all_set 裡的值都被 cover 到為止。

```
void inputfile(string filename){
    ifstream input(filename);
    string tmp;
    int decimal;
    string binary;
    input>>tmp>>var_num;
    input>>tmp;
    while (!input.eof())
    {
        input>>tmp;
        if (tmp==".d") break;
        decimal=stoi(tmp);
        binary=Decimal_To_Binary(decimal);
        on_set.push_back(binary);
        all_set.insert(binary);
    }
    while (!input.eof())
    {
        input>>decimal;
        binary=Decimal_To_Binary(decimal);
        dont_care.insert(binary);
        all_set.insert(binary);
    }
    input.close();
}
//Change decimal to binary
string Decimal_To_Binary(int decimal) {
    if (decimal == 0) return string(var_num, '0');
    string binary = "";
    while (decimal > 0) {
        binary = (decimal % 2 == 0 ? "0" : "1") + binary;
        decimal /= 2;
    }
    while (binary.size() < var_num) {
        binary = "0" + binary;
    }
    return binary;
}
```

```
void find_prime_implicants() {
    while (!all_set.empty()) {
        vector<bool> check_list(all_set.size(), false);
        set<string> n_prime;
        string tmp;
        int counta = 0;

        for (auto it1 = all_set.begin(); it1 != all_set.end(); it1++, counta++) {
            int countb = counta + 1;
            for (auto it2 = next(it1); it2 != all_set.end(); it2++, countb++) {
                int diff=0;
                tmp=*it1;
                for( int i=0; i<(*it1).length(); i++ ) {
                    if( (*it1)[i]!=(*it2)[i]){
                        diff++;
                        tmp[i]='-';
                    }
                }
                if (diff==1) {
                    check_list[counta] = true;
                    check_list[countb] = true;
                    n_prime.insert(tmp);
                }
            }
        }
        for (int i = 0; i < check_list.size(); i++) {
            if (!check_list[i]) {
                prime_implicants.push_back(*next(all_set.begin(), i));
            }
        }
        all_set.clear();
        all_set.insert(n_prime.begin(), n_prime.end());
    }
    return;
}
```

第三部分就是透過 minimum_cover 的 function 來找到最小的 cover，透過講義的方式，我們使用 patrick 演算法來求最佳解，一開始先找出必要的 prime implicant，再將剩下的 prime_implicant 丟進 patrick_algorithm 中來找出所有可能的解，並且同時計算目前最小的解的 prime implicant 個數，最後在我們找到的解空間裡找最佳解時，只需考慮同樣 prime implicant 最小的即可，並透過比較 literal 來找出題目要求的最佳解。

最後再透過 outputfile 的 function 來輸出即可。

```
void patrick_algorithm(set<string>& patrick_set, int idx, int iter) {
    if (iter > min_imp) return;
    if (idx == notess.size()) {
        if (iter <= min_imp) {
            min_imp = iter;
            patrick_imp.push_back(patrick_set);
        }
        return;
    }
    const auto& implicants = notess[idx];
    for (const auto& implicant : implicants) {
        if (patrick_set.insert(implicant).second) {
            patrick_algorithm(patrick_set, idx + 1, iter + 1);
            patrick_set.erase(implicant);
        } else {
            patrick_algorithm(patrick_set, idx + 1, iter);
        }
    }
    return;
}
void minimum_cover() {
    vector<vector<int>> prime_Imp_chart;
    generate_prime_implicant_chart(prime_Imp_chart);
    select_essential_solutions(prime_Imp_chart);
    find_uncovered_implicants(prime_Imp_chart);
    set<string> min_cov;
    patrick_algorithm(min_cov,0,0);
    set<string> best_solutions;
    int min_literal = INT_MAX;
    for (const auto& solution : patrick_imp) {
        if (solution.size() == min_imp) {
            int count = 0;
            for (const auto& iter : solution) {
                count += literalcount(iter);
            }
            if (count <= min_literal) {
                min_literal = count;
                best_solutions.clear();
                best_solutions=solution;
            }
        }
    }
    for (const auto& best_solution : best_solutions) {
        best_sol.push_back(best_solution);
    }
    return;
}
```

```
void generate_prime_implicant_chart(vector<vector<int>>& prime_Imp_chart) {
    prime_Imp_chart.assign(prime_implicants.size() + 1, vector<int>(on_set.size() + 1, 0));
    for (int i = 0; i < prime_implicants.size(); i++) {
        for (int j = 0; j < on_set.size(); j++) {
            if (checkPrime(prime_implicants[i], on_set[j])) {
                prime_Imp_chart[i][j] = 1;
                prime_Imp_chart[prime_implicants.size()][j]++;
                prime_Imp_chart[i][on_set.size()]++;
            }
        }
    }
}
void select_essential_solutions(vector<vector<int>>& prime_Imp_chart) {
    for (int i = 0; i < on_set.size(); i++) {
        if (prime_Imp_chart[prime_implicants.size()][i] == 1) {
            prime_Imp_chart[prime_implicants.size()][i] = 0;
            for (int j = 0; j < prime_implicants.size(); j++) {
                if (prime_Imp_chart[j][i] == 1) {
                    best_sol.push_back(prime_implicants[j]);
                    prime_Imp_chart[j][on_set.size()] = 0;
                    for (int k = 0; k < on_set.size(); k++) {
                        if (prime_Imp_chart[j][k] == 1) {
                            prime_Imp_chart[prime_implicants.size()][k] = 0;
                        }
                    }
                    break;
                }
            }
        }
    }
}
void find_uncovered_implicants(vector<vector<int>>& prime_Imp_chart) {
    for (int i = 0; i < on_set.size(); i++) {
        if (prime_Imp_chart[prime_implicants.size()][i] != 0) {
            vector<string> primes;
            for (int j = 0; j < prime_implicants.size(); j++) {
                if (prime_Imp_chart[j][i] != 0) {
                    primes.push_back(prime_implicants[j]);
                }
            }
            notess.push_back(primes);
        }
    }
}
```

```
void outputfile(string filename) {
    ofstream output(filename);
    stringstream streambuf;
    string s;
    streambuf << prime_implicants.size();
    output << ".p " << streambuf.str() << endl;
    sort(prime_implicants.begin(), prime_implicants.end(), [](const string &a, const string &b) {
        return literal_compare(a,b);
    });

    int size = prime_implicants.size();
    for (int i = 0; i < min(size, 15); i++) {
        s = binary_to_literal(prime_implicants[i]);
        output << s << endl;
    }
    output << endl;
    streambuf.str("");
    streambuf.clear();
    streambuf << best_sol.size();
    output << ".mc " << streambuf.str() << endl;
    sort(best_sol.begin(), best_sol.end(), [](const string &a, const string &b) {
        return literal_compare(a,b);
    });
    size = best_sol.size();
    int literals = 0;
    for (int i = 0; i < size; i++) {
        s = binary_to_literal(best_sol[i]);
        literals += literalcount(best_sol[i]);
        output << s << endl;
    }
    output << "literal=" << literals << endl;
    output.close();
}
```