HW1: Quine-McCluskey Method report

311510182 李旻臻

一、程式碼實作

- Quine-McCluskey Method

1. 先將所有的 onset、dcset 轉成二進制，並按照 1 的數量排序

```
for(int i=0;i<on_set.size();i++){
    string temp="";
    int n=on_set[i];
    int cnt_1=0;
    while(n != 0){
        if(n%2 == 1){
            cnt_1++;
        }
        temp += '0'+ n%2;
        n /= 2;
    }
    int x=num-temp.length();
    while(x){
        temp += '0';
        x--;
    }
    reverse( temp.begin() , temp.end() );
    Implicant temp_impl;
    temp_impl.index=temp;
    temp_impl.num_of_one=cnt_1;
    temp_impl.choose_flag=0;
    impli.push_back(temp_impl);
}
```

```
bool compare_one(const Implicant a,const Implicant b){
    return(a.num_of_one < b.num_of_one);
}
```

2. 將相鄰 1 的數量只差 1 的互相比較，若互相只有一個 char 不同，則合併，無法再繼續合併的則為 prime implicant，從 implicant 的 vector 刪除，加進 prime implicant 的 vector 裡，直到 implicant 的 vector 為空。

```
if(diff_num == 1){
    string temp=impli[i].index;
    temp.at(diff_pos) = '-';
    Implicant get;
    get.index = temp;
    if(impli[i].index.at(diff_pos)=='1'){
        get.num_of_one = impli[i].num_of_one-1;
    }
    else{
        get.num_of_one = impli[i].num_of_one;
    }
    get.choose_flag=0;
    vector<Implicant>::iterator it = std::find(temp_impli.begin(), temp_impli.end(), get);
    if(it==temp_impli.end()){
        temp_impli.push_back(get);
    }
    impli[i].choose_flag=1;
    impli[j].choose_flag=1;
```

```
for(int i=0;i<impli.size();i++){
    if(impli[i].choose_flag==0){
        prime_impli.push_back(impli[i]);
    }
}
```

3. 按照 output 方式排序

```
bool sort_prime(const Implicant a,const Implicant b){
    for(int i=0;i<a.index.length();i++){
        if(a.index[i]=='-' && b.index[i]=='0'){
            return 1;
        }
        else if(a.index[i]=='-' && b.index[i]=='1'){
            return 1;
        }
        else if(a.index[i]=='0' && b.index[i]=='1'){
            return 1;
        }
        else if(a.index[i]=='0' && b.index[i]=='-'){
            return 0;
        }
        else if(a.index[i]=='1' && b.index[i]=='-'){
            return 0;
        }
        else if(a.index[i]=='1' && b.index[i]=='0'){
            return 0;
        }
    }
}
```

- Petrick's method

1. 先將 QM 的結果將 onset 需要的 prime implicant 轉成 POS 形式

```
void Petrick_Method::make_POS(vector<Implicant> &prime_impli, vector<Implicant> &impli){
    for(int i=0;i<impli.size();i++){
        vector<Implicant> total;
        for(int j=0;j<prime_impli.size();j++){
            bool flag=0;
            for(int k=0;k<impli[i].index.length();k++){
                if((prime_impli[j].index[k]!='-') && (prime_impli[j].index[k]!=impli[i].index[k])){
                    flag=1;
                    break;
                }
            }
            if(flag == 0){
                total.push_back(prime_impli[j]);
            }
        }
        POS.push_back(total);
    }
}
```

2. 將 POS 轉成 SOP，並按數量排序

```
for(int i=1;i<POS.size();i++){
    vector< set<string> >temp_SOP;
    set<string> fastSOP;
    for(int j=0;j<SOP.size();j++){
        for(int k=0;k<POS[i].size();k++){
            set<string> temp = SOP[j];
            temp.insert(POS[i][k].index);
            temp_SOP.push_back(temp);
        }
    }
    SOP.clear();
    for(int i=0;i<temp_SOP.size();i++){
        string temp = setTOstring(temp_SOP[i]) ;
        if( fastSOP.find(temp)==fastSOP.end() ) {
            fastSOP.insert( temp );
            SOP.push_back( temp_SOP[i] );
        }
    }
}
sort(SOP.begin(),SOP.end(),compare_sop);
```

3. 計算出同數量的 literal，最小 literal 數的即為解

```
int count_pos=0;
int min_it = INT_MAX;
for(int i=0;i<SOP.size();i++){
    int count_it=0;
    if(SOP[i].size()>SOP[0].size()){
        break;
    }
    for (set<string>::iterator iter = SOP[i].begin(); iter != SOP[i].end(); iter++) {
        for(int j=0;j<(*iter).length();j++){
            if((*iter).at(j)!='-'){
                count_it ++;
            }
        }
    }
    if(count_it <min_it ){
        min_it = count_it;
        count_pos = i;
    }
}
```

二、加速方法

- 在一開始十進制轉二進制就已算好 1 的數量，並存進 struct 裡，而不是每次都 run 過一次 string 的每個字元

```
struct Implicant{
    string index;
    int num_of_one;
    bool choose_flag;
    bool operator==(const Implicant& lhs){ return index == lhs.index;}
};
```

```
while(n != 0){
    if(n%2 == 1){
        cnt_1++;
    }
    temp += '0'+ n%2;
    n /= 2;
}
```

- 在 SOP 裡使用 set，能夠不存到相同的 prime implicant，解決 XX=X 的問題

```
vector< set<string> > SOP;
```

- 在每次放進 SOP 的 vector 前，檢查有無相同的 set，解決 X+X=X 的問題，另外在比較的時候不是用 vector 去找，還是轉換成 set，這樣速度會更快(下圖的 fastSOP 部分)

```
for(int i=0;i<temp_SOP.size();i++){
    string temp = setTOstring(temp_SOP[i]) ;
    if( fastSOP.find(temp)==fastSOP.end() ) {
        fastSOP.insert( temp );
        SOP.push_back( temp_SOP[i] );
    }
}
```

- 在將 POS 展開成 SOP 時選用三層 for 迴圈，沒有選用 recursive 的方法，因為在每次 recursive 的過程，須將所有參數位置皆傳遞一遍，相當耗時。

```
for(int i=1;i<POS.size();i++){
    vector< set<string> >temp_SOP;
    set<string> fastSOP;
    for(int j=0;j<SOP.size();j++){
        for(int k=0;k<POS[i].size();k++){
            set<string> temp = SOP[j];
            temp.insert(POS[i][k].index);
            temp_SOP.push_back(temp);
        }
    }
    SOP.clear();
    for(int i=0;i<temp_SOP.size();i++){
        string temp = setTOstring(temp_SOP[i]) ;
        if( fastSOP.find(temp)==fastSOP.end() ) {
            fastSOP.insert( temp );
            SOP.push_back( temp_SOP[i] );
        }
    }
}
sort(SOP.begin(),SOP.end(),compare_sop);
```