CAD HW1

(1) Name and student ID

Name: 許智凱
Student ID: 108061239

(2) Execution of the program

Please just follow the commands given by TA.

```
$ cd CS3130_PA1
$ make
$ ./pa1 <.in file> <.out file>
(e.g., $ ./pa1 case00.in case00.out)
```

The output file is in .../verifier.

(3) Data structure and the algorithm

We basically follow the concept of Quine-McCluskey algorithm, without other heuristic algorithms. Here is the class we design.

```
class Quine_McCluskey
{
public:
    Quine_McCluskey() {};
    ~Quine_McCluskey() {};
    void read_input_data(string file_name);
    void solve();
    void print_implicant() const;
    void print_prime_implicant() const;
    void print_result() const;
    void gen_output_file(string file_name) const;
private:
    int nVar, nProdTerm;
    bool sort_by_num_of_dc(const string&, const string&);   // sort from a large one to
    bool sort_by_num_of_1(const string&, const string&);
    string merge(const string&, const string&);
    void merge_implicnats_in_one_set(vector<set<string>>&, int);
    void merge_implicants_in_two_sets(vector<set<string>>&, vector<bool>&, int, int);
    bool can_cover(const string&, const string&) const;
    void gen_prime_implicant();         // still not prime ???
    void column_covering();
    vector<string> impli;           // implicants, input data
    vector<string> prime_impli;     // prime implicants
    vector<string> result;          // optimization result
};
```

Generate prime implicants

We handle implicants directly in the type of **string**. We use **vector** to store implicants, prime implicants.

Our program can be divided into two parts:
gen_prime_implicant() and *column_covering()*.
In the *gen_prime_implicant()*, there are 3 steps:

1. Sort implicants by size (the number of dc terms).

   We use **vector<set<string>>(nSize)** to store. The vector length is (the number of variables + 1). Each element in the vector is a **set**. Since we don't want to repeatly check whether an implicant exists or not, so we use set to store. The time of its insert O(logn), but insert in a vector is O(n) if we need to do that check.

2. Merge implicants with the same size.

   This part is apparently easy. Since we have sort implicants by their size (the number of dc terms), there can be only one difference if we want to merge them. we just need to care how many places they are different at, and the index of that positon.

```
int cnt = 0, ind = -1;
string newstr;
for (int i = 0; i < nVar && cnt <= 1; i++) {
    if (s1[i] != s2[i]) {
        cnt++;
        ind = i;
    }
}
if (cnt == 1) {
    newstr = s1;
    newstr[ind] = '-';
}
```

3. Merge implicants across different sizes, and repeat 2. in each iteration.

   The code below is pseudo code. For the reason that there are two choices in *merge_implicnats_in_one_set()* is that we're not sure how can we get the best result. In the theory, 2. can ensure that we have more prime implicants, but in the real experients it didn't bring us

better literal numbers. Maybe it has something to do with our alorithm in *column_covering()*. It's not good enough, so cannot pick out the best prime implicants.

```
for (int i = 0; i < nSize - 1; i++) {
    for (int j = i + 1; j < nSize - 1; j++) {
        merge_implicants_in_two_sets(i, j);

        1. merge_implicnats_in_one_set(i + 1);
        2. for (int k = j; k < nSize - 1; k++)
              merge_implicnats_in_one_set();
    }
}
```

The merging across different sets is kind of complicated. We explains it in the next section.

Column covering

In the *column_covering()*, we need to choose prime implicants with minimum total number of literals that are able to cover all the implicants. Each implicant represents a column, and each prime implicants represent a row. (We imagine the graph in ppt of the course.) There are 3 steps:

1. Preprocess. Get the information we need.

   We need to know:
   - col_cov_by[i]: which prime implicants can cover the implicant i
   - row_cov[i]: which implicants the prime implicant i can cover
   - n_coverby[i]: how many prime implicants cover the implicant i
   - n_uncover[i]: how many **uncovered** implicants the prime implicant can cover.
   - is_covered[i]: has the prime implicant i been used?
   - is_used[i]: has the implicant i been covered?

   ```
   vector<vector<int>> col_cov_by(nProdTerm);
   vector<vector<int>> row_cov(n_prime_impli);
   vector<int> n_coverby(nProdTerm);
   vector<int> n_uncover(n_prime_impli);
   vector<bool> is_covered(nProdTerm);
   vector<bool> is_used(n_prime_impli);
   ```

2. Begin with the essential prime implicants.

   This part we just need to find out those implicants that are covered by only one prime implicant, i.e., the one with n_coverby[i] == 1. Also, we need to maintain the variables *n_coverby*, *n_uncover*, *is_covered*, and *is_used*.

3. Choose the prime implicants that covers the most uncovered columns.

   We search n_uncover to find the largest one that the prime implicant it corresponds to has not been used. Also, we need to maintain the variables carefully. Mark it used. Push it into result. Mark all the columns it covers covered. Maintain the variable *n_uncover*.

(4) other details of the implementation

A. The merging across different sets

The merging is very tricky. For the case below,

```
10-01-
100-1-
10111-
```

it looks like they cannot be merged, but in fact, we can do this.

```
merge 10-01- and 10111- -> get 10-01- and 101-1-
merge 100-1- and 10111- -> get 100-1- and 10-11-
```

Then

```
merge 101-1- and 100-1-
or merge 10-11- and 10-01- -> get 10--1-
```

In the function *merge_implicants_in_two_sets()*, we implement such function.

B. In the double for loop of gen_prime_impli()

As discussed above, we're confused to this part. Here we choose i + 1 as the real implementaion.

```
merge_implicnats_in_one_set(impli_by_size, i + 1);
```

C. Sorting

After generate the prime implicants, we do a sorting. This can enhance the efficiency in the column covering.

```
sort(prime_impli.begin(), prime_impli.end(), [this](const string& s1, const string&
    return this->sort_by_num_of_dc(s1, s2);
});
```

D. Variables maintain in column_covering()

It’s really a tricky part, too. It’s very easy to have bug here.

```
bool flag = true;
while (flag) {
    flag = 0;
    int max = 0, ind = -1;
    for (int i = 0; i < n_prime_impli; i++) {
        if (!is_used[i] && n_uncover[i] > max) {
            max = n_uncover[i];
            ind = i;
        }
    }
    if (ind != -1) {
        result.push_back(prime_impli[ind]);
        is_used[ind] = true;
        for (int k : row_cov[ind]) {
            if (!is_covered[k]) {
                is_covered[k] = true;
                for (int l : col_cov_by[k]) {
                    n_uncover[l]--;
                }
            }
        }
    }
    for (int i = 0; i < nProdTerm; i++) {
        if (!is_covered[i]) {
            flag = 1;
            break;
        }
    }
}
```