Special Topics in Computer Aided Design

Lab1 Quine-Mccluskey Algorithm

312510224 林煜睿

1. **Workflow**

   Read input file → Build Implication table → Combine implicants and get *Prime Implicants* (Column Covering) → Use *non-Prime Implicants* to do ***Petrick's Method*** → Print output

```
QuineMcclusky qm;
qm.readfile(argv[1]);
qm.buildImplicationTable();
while(qm.growImplicant()){ };
qm.columnCovering();

// print result
ofstream output(argv[2]);
qm.printImplicants(output);
qm.printMinimumCovering(output);
output.close();
```

2. **Primary Implicant Generation**

   First construct ***vector<list<Implicant>>****implicationTable*, its index corresponding to the number of "1" of implicant, the list stored inside it contains all implicants having same number of ones, where Data structure ***Implcant*** contains two values, ***string*** binary and ***int*** literal, which is its position in binary form and number of ones in its binary, repectively.

   Then traverse *implicationTable* from index 0 to *implicationTable.size()*-2 (since index *implicationTable.size()*-1 has no implicants can combine with them). If current implicant can combine with implicant inside next layer, mark both of them combinable. Continuously executing until all implicants inside *implicationTable* are not combinable.

   Finally construct an ***unordered_map<int***, ***vector<string***>> *mp*, where its key and value are on-set position and prime implicants in binary form. Traverse *mp* to find all essential prime implicants and non-essential prime implicants, and use essential prime implicants to eliminate covered on-set, and we get *remainOnset*.

3. Cover Remaining On-set

By using remaining on-set we get in previous step, I construct a ***implicantCoverage*** table, its represent the minterms that current prime implicant covers.

Ex:

If there exist 4 minterms, a prime implicant covers the first, second and fourth of minterms, I stored it in integer form: 1+2+4 = 7 = 4’b1011.

Also, I store all literal of implicants inside ***vector<int>*** *literalsCount*.

Then use dynamic programming to find the minimum cover. I construct the *dp* vector to record the best solution of corresponding on-set position.

Ex:

dp[3] = *dp*[4’b0011] = the minimum number of implicants that covers the first and second minterms.

dp[12] = *dp*[4’b1100] = the minimum number of implicants that covers the third and fourth minterms.

I also create *parent* and *choice* vector to record the trace-information.

Ex:

If *parent*[5] = 3, since 5 = 4’b0101, 3 = 4’b0011, it means that to cover the first and third minterms(4’b0101), the best solution (using the least number of implicants) is generated from the result that covers the first and second minterms(4’b0011).

The *choice* vector records the implicant we choose to cover the current minterms.

```
vector<int> dp(maxState, INT_MAX); // idx: covered onset, ex: if covered 0,1,3 onset, its idx is 3'b1011 = 11
                                   // value: minimum implicant number that can cover current onsets
vector<int> parent(maxState, -1);
vector<int> choice(maxState, -1);
vector<int> literalsDp(maxState, INT_MAX);
dp[0] = 0;
literalsDp[0] = 0;

for (int i = 0; i < maxState; ++i) {
    if (dp[i] == INT_MAX) continue; // can not cover the i_th onset
    for (size_t j = 0; j < nonEssPrimeImp.size(); ++j) {
        int nextCover = i | implicantCoverage[j];
        if (dp[i] + 1 < dp[nextCover] || (dp[i] + 1 == dp[nextCover] && (literalsDp[i] + literalsCount[j] < literalsDp[nextCover]))) {
            dp[nextCover] = dp[i] + 1;
            literalsDp[nextCover] = literalsDp[i] + literalsCount[j];
            parent[nextCover] = i;
            choice[nextCover] = j;
        }
    }
}
```

Therefore, after executing the function, I can get minimum number of implicants to cover all minterms in $dp[4'b1111] = dp[15]$. Then trace back to find which implicant we choose to fulfill the answer by *parent* and *choice* vector.

```
// top-down to trace the choosed prime implicants
vector<string> implicants;
int curState = maxState - 1;
// cout << dp[curState] << '\n';
while (parent[curState] != -1) {
    implicants.push_back(nonEssPrimeImp[choice[curState]]);
    curState = parent[curState];
}
```

4. Pre-Compiled Table

Since the maximum input variable number is 8, we can generate ***in2Binary*** and ***binary2Int*** table to fasten the execution.

```
//first: binary, second: number of "1" inside binary
const vector<vector<pair<string,int>>> int2Binary = {
    {{"0",0}},
    {{"0",0},{"1",1}},
    {{"00",0},{"01",1},{"10",1},{"11",2}},
    {{"000",0},{"001",1},{"010",1},{"011",2},{"100",1},{"101",2},{"110",2},{"111",3}},
    {{"0000",0},{"0001",1},{"0010",1},{"0011",2},{"0100",1},{"0101",2},{"0110",2},{"0111",3},{"1000",1},{"1001",2},{"1010",2}
    {{"00000",0},{"00001",1},{"00010",1},{"00011",2},{"00100",1},{"00101",2},{"00110",2},{"00111",3},{"01000",1},{"01001",2},
    {{"000000",0},{"000001",1},{"000010",1},{"000011",2},{"000100",1},{"000101",2},{"000110",2},{"000111",3},{"001000",1},{"0
    {{"0000000",0},{"0000001",1},{"0000010",1},{"0000011",2},{"0000100",1},{"0000101",2},{"0000110",2},{"0000111",3},{"000100
    {{"00000000",0},{"00000001",1},{"00000010",1},{"00000011",2},{"00000100",1},{"00000101",2},{"00000110",2},{"00000111",3},
};
```

```
const unordered_map<string,int> binary2Int = {
    {"0",0},
    {"1",1},
    {"00",0},
    {"01",1},
    {"10",2},
    {"11",3},
    {"000",0},
    {"001",1},
    {"010",2},
    {"011",3},
    {"100",4},
    {"101",5},
    {"110",6},
    {"111",7},
    {"0000",0},
    {"0001",1},
    {"0010",2},
    {"0011",3},
    {"0100",4},
    {"0101",5},
    {"0110",6},
    {"0111",7},
    {"1000",8},
    {"1001",9},
    {"1010",10},
    {"1011",11},
    {"1100",12},
    {"1101",13},
    {"1110",14},
    {"1111",15},
    {"00000",0},
```