

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 *Implicant* contains two values, *string* binary and *int* literal, which is its position in binary form and number of ones in its binary, respectively.

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.

```
8 //first: binary, second: number of "1" inside binary
9 const vector<vector<pair<string,int>>> int2Binary = {
10     {"0",0},
11     {"0",0}, {"1",1},
12     {"00",0}, {"01",1}, {"10",1}, {"11",2},
13     {"000",0}, {"001",1}, {"010",1}, {"011",2}, {"100",1}, {"101",2}, {"110",2}, {"111",3},
14     {"0000",0}, {"0001",1}, {"0010",1}, {"0011",2}, {"0100",1}, {"0101",2}, {"0110",2}, {"0111",3}, {"1000",1}, {"1001",2}, {"1010",2}, {"1011",3}, {"1100",2}, {"1101",3}, {"1110",3}, {"1111",4},
15     {"00000",0}, {"00001",1}, {"00010",1}, {"00011",2}, {"00100",1}, {"00101",2}, {"00110",2}, {"00111",3}, {"01000",1}, {"01001",2}, {"01010",2}, {"01011",3}, {"01100",2}, {"01101",3}, {"01110",3}, {"01111",4}, {"10000",1}, {"10001",2}, {"10010",2}, {"10011",3}, {"10100",2}, {"10101",3}, {"10110",3}, {"10111",4}, {"11000",2}, {"11001",3}, {"11010",3}, {"11011",4}, {"11100",3}, {"11101",4}, {"11110",4}, {"11111",5},
16     {"000000",0}, {"000001",1}, {"000010",1}, {"000011",2}, {"000100",1}, {"000101",2}, {"000110",2}, {"000111",3}, {"001000",1}, {"001001",2}, {"001010",2}, {"001011",3}, {"001100",2}, {"001101",3}, {"001110",3}, {"001111",4}, {"010000",1}, {"010001",2}, {"010010",2}, {"010011",3}, {"010100",2}, {"010101",3}, {"010110",3}, {"010111",4}, {"011000",2}, {"011001",3}, {"011010",3}, {"011011",4}, {"011100",3}, {"011101",4}, {"011110",4}, {"011111",5}, {"100000",1}, {"100001",2}, {"100010",2}, {"100011",3}, {"100100",2}, {"100101",3}, {"100110",3}, {"100111",4}, {"101000",2}, {"101001",3}, {"101010",3}, {"101011",4}, {"101100",3}, {"101101",4}, {"101110",4}, {"101111",5}, {"110000",2}, {"110001",3}, {"110010",3}, {"110011",4}, {"110100",3}, {"110101",4}, {"110110",4}, {"110111",5}, {"111000",3}, {"111001",4}, {"111010",4}, {"111011",5}, {"111100",4}, {"111101",5}, {"111110",5}, {"111111",6},
17     {"0000000",0}, {"0000001",1}, {"0000010",1}, {"0000011",2}, {"0000100",1}, {"0000101",2}, {"0000110",2}, {"0000111",3}, {"0001000",1}, {"0001001",2}, {"0001010",2}, {"0001011",3}, {"0001100",2}, {"0001101",3}, {"0001110",3}, {"0001111",4}, {"0010000",1}, {"0010001",2}, {"0010010",2}, {"0010011",3}, {"0010100",2}, {"0010101",3}, {"0010110",3}, {"0010111",4}, {"0011000",2}, {"0011001",3}, {"0011010",3}, {"0011011",4}, {"0011100",3}, {"0011101",4}, {"0011110",4}, {"0011111",5}, {"0100000",1}, {"0100001",2}, {"0100010",2}, {"0100011",3}, {"0100100",2}, {"0100101",3}, {"0100110",3}, {"0100111",4}, {"0101000",2}, {"0101001",3}, {"0101010",3}, {"0101011",4}, {"0101100",3}, {"0101101",4}, {"0101110",4}, {"0101111",5}, {"0110000",2}, {"0110001",3}, {"0110010",3}, {"0110011",4}, {"0110100",3}, {"0110101",4}, {"0110110",4}, {"0110111",5}, {"0111000",3}, {"0111001",4}, {"0111010",4}, {"0111011",5}, {"0111100",4}, {"0111101",5}, {"0111110",5}, {"0111111",6}, {"1000000",1}, {"1000001",2}, {"1000010",2}, {"1000011",3}, {"1000100",2}, {"1000101",3}, {"1000110",3}, {"1000111",4}, {"1001000",2}, {"1001001",3}, {"1001010",3}, {"1001011",4}, {"1001100",3}, {"1001101",4}, {"1001110",4}, {"1001111",5}, {"1010000",2}, {"1010001",3}, {"1010010",3}, {"1010011",4}, {"1010100",3}, {"1010101",4}, {"1010110",4}, {"1010111",5}, {"1011000",3}, {"1011001",4}, {"1011010",4}, {"1011011",5}, {"1011100",4}, {"1011101",5}, {"1011110",5}, {"1011111",6}, {"1100000",2}, {"1100001",3}, {"1100010",3}, {"1100011",4}, {"1100100",3}, {"1100101",4}, {"1100110",4}, {"1100111",5}, {"1101000",3}, {"1101001",4}, {"1101010",4}, {"1101011",5}, {"1101100",4}, {"1101101",5}, {"1101110",5}, {"1101111",6}, {"1110000",3}, {"1110001",4}, {"1110010",4}, {"1110011",5}, {"1110100",4}, {"1110101",5}, {"1110110",5}, {"1110111",6}, {"1111000",4}, {"1111001",5}, {"1111010",5}, {"1111011",6}, {"1111100",5}, {"1111101",6}, {"1111110",6}, {"1111111",7},
18 };
19
```

```
21 const unordered_map<string,int> binary2Int = {
22     {"0",0},
23     {"1",1},
24     {"00",0},
25     {"01",1},
26     {"10",2},
27     {"11",3},
28     {"000",0},
29     {"001",1},
30     {"010",2},
31     {"011",3},
32     {"100",4},
33     {"101",5},
34     {"110",6},
35     {"111",7},
36     {"0000",0},
37     {"0001",1},
38     {"0010",2},
39     {"0011",3},
40     {"0100",4},
41     {"0101",5},
42     {"0110",6},
43     {"0111",7},
44     {"1000",8},
45     {"1001",9},
46     {"1010",10},
47     {"1011",11},
48     {"1100",12},
49     {"1101",13},
50     {"1110",14},
51     {"1111",15},
52     {"00000",0},
```