Quine-McCluskey Algorithm Implementation in C++

Ebram Gamal Hanin
900237803
Mohamed Mohamed
900246207
Hazem Nasr
900242124

Contents

Chapter 1

Introduction

This report details the design, implementation, and testing of a C++ program that solves Boolean logic minimization using the Quine-McCluskey (Q-M) algorithm. The Quine-McCluskey algorithm is a tabular method of logic minimization that is more suitable for computer automation than the Karnaugh map. It is a fundamental algorithm in digital logic design, used to find the most-simplified sum-of-products (SOP) expression for a given Boolean function.

The program takes a set of minterms and don't-care terms as input and produces a minimal SOP expression. This minimal expression corresponds to a logic circuit with the minimum number of gates and inputs, which is crucial for optimizing circuit cost and complexity.

This document covers the core program design, including the algorithms and data structures used, the challenges faced during development, the testing strategy employed, and instructions for building and using the program. It also discusses known issues and the contributions of each team member.

Chapter 2

Program Design

The program is a console-based C++ application built as a single-file project (‘main.cpp‘). It implements the full Quine-McCluskey algorithm in two main phases: first, finding all Prime Implicants, and second, solving the Prime Implicant chart to find a minimal cover.

2.1 Algorithms

The core logic is divided into several functions that map directly to the steps of the Q-M algorithm.

2.1.1 Phase 1: Finding Prime Implicants

This phase is implemented in the `getPrimeImplicants` function.

1. **Initial Grouping:** All minterms and don't-care terms are placed into groups based on the number of '1's in their binary representation. This is calculated using `__builtin_popcount(term.num)`. The structure used is a `vector<set<binaryInt>>`.

2. **Iterative Combining:** The program iteratively compares terms from adjacent groups (e.g., group i and group $i+1$). The `are1BitOff` function checks if two terms differ by exactly one bit.

3. If they do, the `combine` function merges them into a new term, marking the differing bit with a dash (stored in the `dashes` member).

4. **Check-off Mechanism:** A master `std::set<binaryInt> implicants` is used. When two terms are combined, they are both erased from this set, and the new, combined term is added.

5. This process repeats (`while(nextColumn(...))`) until no more terms can be combined. The terms remaining in the `implicants` set are, by definition, the Prime Implicants (PIs).

2.1.2 Phase 2: Solving the Prime Implicant Chart

This phase begins in `createPrimeImplicantChart` and is completed in `generateExpression`.

1. **Chart Creation:** A chart is built and stored in a `std::map<binaryInt, std::string>`. The key is the `binaryInt` representing the Prime Implicant, and the value is a "coverage string" of '1's and '0's, indicating which of the original minterms (not don't-cares) it covers.

2. **Find Essential Prime Implicants (EPIs):** The `getEssentialPrimeImplicants` function iterates through each *column* (minterm) of the chart. If a minterm is covered by only one PI, that PI is "essential" and must be part of the final solution.

3. **Chart Reduction:** The `generateExpression` function first adds all EPIs to the `final` solution vector. It then removes these EPIs (rows) from the chart and all minterms (columns) that they cover.

4. **Solving the Remaining Chart (Greedy Heuristic):** If any minterms remain uncovered, the chart is "cyclic" or requires further solving. This program implements a greedy heuristic rather than the more complex Petrick's Method.

    - In a `while` loop, the algorithm finds the PI that covers the *greatest number* of remaining minterms (using `popcount` on the coverage string).
    - This "best" PI is added to the `final` solution.
    - The PI and the minterms it covers are removed from the chart.
    - This repeats until all minterms are covered.

2.2 Data Structures

The choice of data structures was critical for an efficient implementation.

2.2.1 The `binaryInt` Struct

The core data structure for the entire program is a custom struct, `binaryInt`. It is designed to efficiently represent a Boolean term (e.g., A'B-D) using bitwise operations.

```
1  struct binaryInt{
2      unsigned num;        //the ones in the represented number match the ones in ‘
       num‘
3      unsigned dashes;     //the dashes in the represented number match the ones in
       ‘dashes‘
4
5      binaryInt(unsigned n=0, unsigned d=0) : num(n), dashes(d) {}
6
7      //... operator<, operator==
8
9      bool covers(binaryInt b) const{
10         for(int i=0; i < numberOfVariables; ++i){
11             if((num>>i)&1 && !((b.num>>i)&1)) return false;
12             if(!((num>>i)&1) && !((dashes>>i)&1) && ((b.num>>i)&1)) return false
       ;
13         }
```

```
14          return true;
15      }
16  };
```

Listing 2.1: The binaryInt struct definition

A term is represented by two unsigned integers:

- **num:** A bit is '1' in num if the corresponding variable is '1' in the term.
- **dashes:** A bit is '1' in dashes if the corresponding variable is a don't-care ('-') in the term.
- If a bit is '0' in *both* num and dashes, the variable is '0' (complemented) in the term.

For example, for 4 variables (A,B,C,D), the term A'C- (binary 0-1-) would be stored as num = 0010 and dashes = 0101. This struct also overloads operator< so it can be used as a key in std::set and std::map.

2.2.2 Standard Template Library (STL) Containers

- **std::vector<set<binaryInt»:** Used in Phase 1 to hold the groups of terms, indexed by their '1' count.
- **std::set<binaryInt>:** Used in Phase 1 as the master list of implicants. Its auto-sorting and uniqueness properties are ideal for the "check-off" algorithm.
- **std::map<binaryInt, std::string>:** Used in Phase 2 to represent the Prime Implicant chart. It provides an efficient mapping from a PI to its minterm coverage.

Chapter 3

Challenges

Several challenges were encountered and overcome during the development of this program.

- **Representing Boolean Terms:** The most significant initial challenge was designing an efficient way to store and manipulate terms like `10-1`. An early prototype using `std::string` was very slow. The final `binaryInt` struct using bitwise logic is far more efficient but required careful implementation of the `covers`, `are1BitOff`, and `combine` methods to ensure correctness.
- **Maxterm-to-Minterm Conversion:** The program is required to accept Maxterms (e.g., `M(0,2,4)`) as input. Implementing this conversion in the `takeInput` function was non-trivial. The logic must iterate through all 2^N possible terms and add any term that is *not* in the Maxterm set and *not* in the Don't-Care set. An "off-by-one" bug was found in this loop, where it only iterated up to 2^{N-1}, which has since been corrected.
- **Chart Solving Complexity:** After finding Essential Prime Implicants, the remaining chart can be complex. The academically "correct" solution involves complex algorithms like Petrick's Method. We faced a design choice: implement a complex, fully optimal algorithm, or a simpler, faster heuristic. We chose a greedy heuristic for this project, which finds the PI that covers the most remaining minterms. This was a trade-off between development time and guaranteed optimality.
- **Guaranteeing Minimality:** A direct consequence of choosing the greedy heuristic is that no simple procedure guarantees the final solution is the *absolute* minimal one, especially in cases with a cyclic PI chart.
- **Systematic Minimality Testing:** A related challenge is the lack of a systematic way to test the *minimality* of the output. While testing for *validity* (correct coverage) is straightforward, proving minimality would require a separate, fully optimal solver to use as a benchmark.

Chapter 4

Testing

A set of ten tests was designed to exercise all features of the program. Below is a detailed description.

1&2 are the trivial edge cases: the tautology $F(A, B) = 1$ and the contradiction $F(A, B) = 0$. The program produced the correct final expressions (just 1 and 0) and the correct Verilog descriptions (`assign F = 1'b1;` and `assign F = 1'b0;`). No further checks were necessary.

3&4 are random 4- and 8-variable functions provided as minterm and maxterm lists to ensure the program produces output for arbitrary inputs.

5–8 are meaningful circuits of varying sizes (full-adder sum, full-adder carry, two-ones 10-bit detector, and a 12-bit equality comparator). For each, we used a trusted golden model to validate the generated output. Self-checking test benches were run on Vivado and confirmed correctness. A sample test bench (for the 12-bit equality comparator) follows:

```
‘timescale 1ns/1ps

module equalityComp12TB();
  reg [11:0] in_vec;
  wire Final;
  equalityComp12 comparator(
    .A(in_vec[11]),
    .B(in_vec[10]),
    .C(in_vec[9]),
    .D(in_vec[8]),
    .E(in_vec[7]),
    .F(in_vec[6]),
    .G(in_vec[5]),
    .H(in_vec[4]),
    .I(in_vec[3]),
    .J(in_vec[2]),
```

```
    .K(in_vec[1]),
    .L(in_vec[0]),
    .Final(Final)
  );

  integer i;
  initial begin
    for (i = 0; i < 4096; i = i + 1) begin
      in_vec = i;
      #1;
      if (Final != (in_vec[11:6] == in_vec[5:0])) $display("Test %d Failed", i);
    end

    $display("Test complete.");
    #1 $finish;
  end

endmodule
```

9&10 are computationally heavy tests intended to push the algorithm to its limits. The first is a 16-bit odd-parity generator (32,768 minterms), and the second is a 20-bit majority function (about 431,910 minterms). The program took excessively long to solve the first test, so we did not attempt the second. This behavior highlights the primary limitation of the current implementation: exponential time complexity.

Chapter 5

Build and Usage Instructions

5.1 Building the Program

The program is contained in a single file, `main.cpp`, can be ran through any C++ IDE combined with a GNU compiler.

5.2 Using the Program

1. Run the code through the compiler.
2. The program will first prompt for an input file name.

   ```
   Initializing The Program...
   Input File: test.txt
   ```

3. It will then prompt for an output file name. All results will be written to this file.

   ```
   Output File: out.txt
   ```

4. Finally, it will ask if you want to display intermediate steps. Answering 'y' will print the full Prime Implicant chart, which can be very large.

   ```
   ...Do you want to display intermediate steps? (y/n) n
   ```

5. The program will write its results to your output file and generate a `logic_circuit.v` Verilog file.

5.3 Input File Format

The input file (e.g., `test.txt`) must follow a strict three-line format:

- **Line 1:** The number of variables (e.g., `4`).
- **Line 2:** The minterms, comma-separated, prefixed with `m` or `M`.
  - Example (minterms): `m1, m3, m7, m11, m15`
  - Example (Maxterms): `M0, M2, M4, M5, M6`
- **Line 3:** The don't-care terms, comma-separated, prefixed with `d`.
  - Example: `d0, d5)`

Chapter 6

Problems and Remaining Issues

While the program is fully functional, there are known limitations and areas for future improvement.

- **Solver Optimality:** The most significant limitation is the use of a greedy heuristic for solving the PI chart. While much simpler to implement than Petrick's Method or a backtracking algorithm, a greedy choice is not guaranteed to produce the *absolute* minimal SOP expression in all complex cyclic chart cases. It finds a *correct* and *minimal-ish* solution, but perhaps not the *true minimum* number of literals.
- **Perfect Input Validation:** The current input parsing is basic. It checks for the `m`/`M`/`d` prefixes but does not handle malformed input gracefully (e.t., `m1, m3,, m5`. A more correct parser using regular expressions or more careful string splitting would improve usability.
- **Verilog Generation:** The function to generate a Verilog module (`logic_circuit.v`) is a valuable addition. However, it currently uses a fixed set of variable names (`A`, `B`, `C`, etc.). This could be expanded to allow custom variable names specified by the user.

Chapter 7

Member Contributions

This project was a collaborative effort. The responsibilities were divided as follows:

- **Ebram Gamal Hanin:**
  - Wrote the code for the function responsible for the generation of the Verilog module.
  - Helped with code optimization and bug fixes.
  - Wrote the project report
- **Mohamed Mohamed:**
  - Contributed to the design of the implementation used for the algorithm.
  - Wrote the code for generating the PI chart and help.
  - Helped with code optimization and bug fixes.
- **Hazem Nasr:**
  - Contributed to the design of the implementation used for the algorithm.
  - Wrote the code for solving the PI chart.
  - Helped with code optimization and bug fixes.