Abstract

This project focuses on implementing the Quine-McCluskey (QM) algorithm for the minimization of Boolean functions with five variables. The Python program takes the minterms as input, simplifies the logic expression using the QM technique, and provides the minimized Boolean expression. This method offers a systematic and efficient approach for logic minimization, especially for digital logic design problems where Karnaugh maps become complex.

Objective

The objective of this project is to implement a Python code to simplify Boolean functions using the Quine-McCluskey technique for five variables. The goal is to provide a minimized Boolean expression for any given set of minterms, improving efficiency in logic circuit design.

Problem Statement

Minimizing Boolean functions is critical in digital design to reduce the complexity of circuits. When dealing with a larger number of variables (e.g., 5 or more), traditional methods like Karnaugh maps become impractical. The Quine-McCluskey method provides a tabular approach to minimize Boolean functions systematically. This project aims to create a Python program to simplify such functions using the QM technique.

Introduction and Background

Quine-McCluskey (QM) Technique

The Quine-McCluskey method is a prime implicant method used for the minimization of Boolean functions. It is a tabular method and is capable of handling functions with multiple variables efficiently.

There are 4 main steps in the Quine-McCluskey algorithm:

1. Generate Prime Implicants
2. Construct a Prime Implicant Table
3. Reduce Prime Implicant Table
   (a) Remove Essential Prime Implicants
   (b) Row Dominance
   (c) Column Dominance
4. Solve Prime Implicant Table

Methodology

The implementation involves the following steps:

1. **Input Minterms**: Users are prompted to input the minterms of the Boolean function.
2. **Binary Representation**: Each minterm is converted to its binary equivalent with five variables.

3. **Grouping Minterms**: The minterms are grouped based on the number of 1's in their binary representation.
4. **Prime Implicant Generation**: Pairs of binary strings differing by one bit are merged by replacing the differing bit with a '-' to form prime implicants.
5. **Prime Implicant Chart**: A chart is constructed to map which prime implicants cover which minterms.
6. **Selection of Essential Prime Implicants**: Essential prime implicants are selected to cover all minterms with minimal logic terms.
7. **Minimized Expression**: The minimized Boolean expression is output as the result.

Python Code

The Python code is structured as follows:

Function Definitions:

- **get_minterms()**: This function takes the user input of minterms and processes them into a list of integers.
- **binary_representation(n, variables)**: Converts a number n into its binary form with variables bits (in this case, 5 bits).
- **count_ones(binary)**: Counts the number of 1's in the binary string.
- **find_prime_implicants(minterms, variables)**: Groups the minterms based on their binary representations and finds the prime implicants by combining binary strings that differ by only one bit.
- **get_prime_implicants_chart(prime_implicants, minterms)**: Constructs a prime implicant chart that shows which minterms are covered by which prime implicants.
- **select_essential_prime_implicants(chart)**: Selects essential prime implicants by covering all minterms with the fewest possible prime implicants.
- **implicant_to_expression(implicant)**: Converts a prime implicant into its corresponding Boolean expression using variables A, B, C, D, and E.
- **quine_mccluskey(minterms, variables)**: Integrates all the above steps to apply the QM technique and produce the minimized Boolean expression.

Main Function:

- **main()**: The entry point of the program. It prompts the user for input, runs the QM algorithm, and prints the essential prime implicants and minimized Boolean expression.

Working

1. **User Input**: The user inputs the minterms of the Boolean function.

2. **Binary Conversion**: The minterms are converted to binary.
3. **Grouping and Combining**: Minterms are grouped and combined iteratively to form prime implicants.
4. **Prime Implicant Chart**: A chart is constructed to see which prime implicants cover which minterms.
5. **Minimization**: Essential prime implicants are selected, and the minimized Boolean expression is generated.

CODING

LOGIC

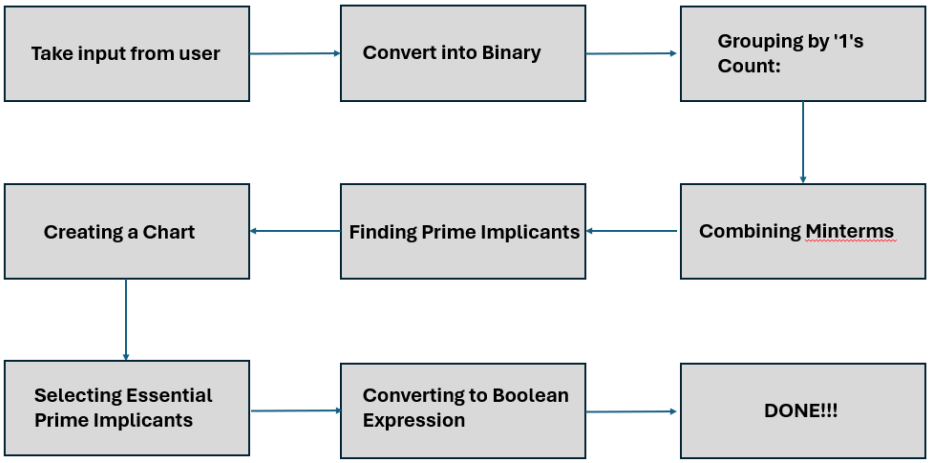

Figure 1. Flowchart.

The code implements the Quine-McCluskey algorithm, which simplifies Boolean expressions by finding the smallest set of terms (called "prime implicants") that cover all the input minterms.

Here's the logic in simpler terms:

1. Getting User Input:

- The code asks the user to enter minterms (the decimal numbers where the Boolean function is true).
- These numbers are then converted into a list of integers.

2. Converting to Binary:

- Each minterm is converted into its binary form to work with it logically.
- The number of variables in the Boolean expression (5 in this case) determines the length of the binary string (e.g., 00011 for 5 variables).

3. Grouping by '1's Count:

- The binary forms are grouped by the number of '1's they contain. For example, all minterms with 2 '1's go into one group, and those with 3 '1's go into another.

4. Combining Minterms:

- Minterms that differ by only one bit are combined by replacing the differing bit with a '-'. This shows that the bit can be either '0' or '1' (called a don't care condition).
- This process is repeated until no further combinations can be made.

5. Finding Prime Implicants:

- The remaining combined terms that cannot be combined any further are the prime implicants.

6. Creating a Chart:

- A chart is created to see which minterms are covered by which prime implicants. This chart shows how each prime implicant relates to the original minterms.

7. Selecting Essential Prime Implicants:

- Some prime implicants are essential, meaning they are the only ones that cover certain minterms. These essential implicants are selected to ensure all minterms are covered.

8. Converting to Boolean Expression:

- The prime implicants are then turned into a Boolean expression by converting their binary form back into variables (like A, B, etc.).
- For example, 101-- becomes A'C.

9. Result:

- Finally, the code prints out the simplified Boolean expression that represents the input minterms in the shortest form possible.

Summary:

- The algorithm simplifies Boolean expressions by grouping minterms, combining them based on similar patterns, and finding the most efficient terms (prime implicants) to express the function. It ensures all input minterms are covered using the fewest terms possible.

CODE

```
import itertools

# Function to get minterms from user input
def get_minterms():
    minterms = input("Enter the minterms separated by commas: ")  # Read minterms from user
    minterms = minterms.replace(" ", "")  # Remove any spaces from the input
    minterms = [int(x) for x in minterms.split(',')]  # Convert input string to a list of integers
    return minterms
# Function to convert a decimal number to a binary string with a fixed length
def binary_representation(n, variables):
    return format(n, '0{}b'.format(variables))  # Format the number to binary with leading zeros

# Function to count the number of '1's in a binary string
def count_ones(binary):
    return binary.count('1')  # Count occurrences of '1' in the binary string

# Function to find prime implicants using the Quine-McCluskey algorithm
def find_prime_implicants(minterms, variables):
    # Group minterms based on the number of '1's in their binary representation
    groups = {}
    for minterm in minterms:
        binary = binary_representation(minterm, variables)
        ones_count = count_ones(binary)
        if ones_count not in groups:
            groups[ones_count] = []
        groups[ones_count].append(binary)

    marked = set()  # Set to keep track of minterms that have been marked as part of an implicant
    prime_implicants = set()  # Set to store prime implicants
    while groups:
        new_groups = {}
        grouped_pairs = set()  # Set to store new implicants formed by combining existing ones
        keys = sorted(groups.keys())  # Sort group keys to ensure proper pairing

    # Combine minterms from adjacent groups to form new implicants
        for i in range(len(keys) - 1):
            group1 = keys[i]
            group2 = keys[i + 1]
            for a in groups[group1]:
                for b in groups[group2]:
                    diff = [idx for idx, (x, y) in enumerate(
                        zip(a, b)) if x != y]
                    if len(diff) == 1:  # Check if the binary strings differ by only one bit
                        new_binary = a[:diff[0]] + '-' + a[diff[0] + 1:]  # Create a new implicant by
replacing differing bit with '-'
                        if new_binary not in new_groups:
                            new_groups.setdefault(
                                group1, []).append(new_binary)
                        marked.add(a)
                        marked.add(b)
                        grouped_pairs.add(new_binary)

     # Update prime implicants with those that were not marked
        prime_implicants.update(
            set(itertools.chain.from_iterable(groups.values())) - marked)
        groups = new_groups  # Move to the next level of groups

    return prime_implicants

# Function to create a chart of prime implicants vs. minterms they cover
def get_prime_implicants_chart(prime_implicants, minterms):
    chart = {}
    for prime in prime_implicants:
        chart[prime] = []
        for minterm in minterms:
            # Check if the minterm is covered by the prime implicant
            if all(prime[i] == '-' or prime[i] == bit for i, bit in
enumerate(binary_representation(minterm, len(prime)))):
                chart[prime].append(minterm)
    return chart
```

```
# Function to select essential prime implicants
def select_essential_prime_implicants(chart):
    essential_prime_implicants = set()
    all_minterms = set(itertools.chain.from_iterable(chart.values()))

    while all_minterms:
        # Find the prime implicant that covers the maximum number of remaining minterms
        max_cover = max(chart, key=lambda k: len(set(chart[k]) & all_minterms))
        essential_prime_implicants.add(max_cover)
        all_minterms -= set(chart[max_cover])  # Remove covered minterms from the set
        chart = {k: v for k, v in chart.items(
        ) if k not in essential_prime_implicants}  # Remove selected implicants from the chart

    return essential_prime_implicants

# Function to convert an implicant to a Boolean expression
def implicant_to_expression(implicant):
    variables = ['A', 'B', 'C', 'D', 'E']  # Define variable names
    term = []
    for i, char in enumerate(implicant):
        if char == '1':
            term.append(variables[i])  # Add variable if the bit is '1'
        elif char == '0':
            term.append(f"{variables[i]}'")  # Add negated variable if the bit is '0'
    return ''.join(term)  # Join all parts to form the expression

# Main function to run the Quine-McCluskey algorithm
def quine_mccluskey(minterms, variables):
    prime_implicants = find_prime_implicants(minterms, variables)
    prime_implicants_chart = get_prime_implicants_chart(
        prime_implicants, minterms)
    essential_prime_implicants = select_essential_prime_implicants(
        prime_implicants_chart)

    # Generate the minimized Boolean expression
    minimized_expression = ' + '.join(implicant_to_expression(implicant)
                                      for implicant in essential_prime_implicants)
    return essential_prime_implicants, minimized_expression

# Main entry point of the script
def main():
    minterms = get_minterms()  # Get minterms from user
    variables = 5  # Number of variables (assuming 5 for this example)
    essential_prime_implicants, minimized_expression = quine_mccluskey(
        minterms, variables)

    print("Essential Prime Implicants:")
    for epi in essential_prime_implicants:
        print(epi)  # Print each essential prime implicant
    print("\nMinimized Boolean Expression:")
    print(minimized_expression)  # Print the minimized Boolean expression

# Run the main function if the script is executed
if __name__ == "__main__":
    main()
```

CODE EXPLANATION

Importing the itertools Module

- The script starts by importing the itertools module, which provides tools for efficient looping and combining data.
- In this case, it's used to flatten nested structures like lists and sets.

Functions

1. **get_minterms()**

- **Purpose**: This function prompts the user to input minterms (numbers corresponding to the rows in a truth table where the function is true).
- **Steps**:
    - The user enters a series of minterms separated by commas.
    - Any spaces are removed, and the input is split into a list.
    - The list of strings is converted into a list of integers, representing the minterms.

---

2. **binary_representation(n, variables)**

- **Purpose**: Converts a decimal number (minterm) into a binary string with a fixed length, depending on the number of variables.
- **Steps**:
    - The number is converted into binary, and if necessary, leading zeros are added to match the number of variables.
    - For example, the decimal 3 would become 00011 for 5 variables.

---

3. **count_ones(binary)**

- **Purpose**: Counts how many 1s are in the binary string.
- **Steps**:
    - This is done by using the count() function on the binary string.
    - It helps later when grouping minterms based on the number of 1s in their binary representation.

---

4. **find_prime_implicants(minterms, variables)**

- **Purpose**: Implements the main logic for finding **prime implicants**.

- **Steps**:
    - **Grouping minterms**:
        - The minterms are first grouped based on how many 1s their binary representation contains. For example, the binary number 00011 (which has two 1s) will go in the group for two 1s.
    - **Combining minterms**:
        - Minterms from adjacent groups (e.g., a group with two 1s and one with three 1s) are checked to see if they differ by only one bit.
        - If they differ by one bit, they are combined into a new binary string with a - symbol, which represents that the bit can be either 0 or 1 (don't care).
    - **Marking and tracking**:
        - Minterms that can be combined are "marked" to prevent duplication, and new implicants formed are stored in the new_groups.
    - **Repeat until done**:
        - This process continues until no further combinations are possible, and the remaining unmarked minterms are the **prime implicants**.

---

5. **get_prime_implicants_chart(prime_implicants, minterms)**

- **Purpose**: Creates a chart that maps prime implicants to the minterms they cover.
- **Steps**:
    - For each prime implicant, it checks which minterms it covers (i.e., if the implicant can produce the minterm based on its binary form).
    - The result is a dictionary where the keys are prime implicants, and the values are the list of minterms they cover.

---

6. **select_essential_prime_implicants(chart)**

- **Purpose**: Selects **essential prime implicants**, which are needed to cover all the minterms.
- **Steps**:
    - It iteratively selects the prime implicant that covers the most uncovered minterms.
    - Once a prime implicant is selected, the minterms it covers are removed from further consideration.
    - The process continues until all minterms are covered.

---

7. **implicant_to_expression(implicant)**

- **Purpose**: Converts a prime implicant into a readable Boolean expression.
- **Steps**:
    - Each 1 in the implicant corresponds to the original variable (e.g., A, B, C, D, E).
    - Each 0 corresponds to the negation of the variable (e.g., A', B', etc.).
    - If there is a -, the variable is ignored because it represents a "don't care" condition.

---

8. **quine_mccluskey(minterms, variables)**

- **Purpose**: The main function that runs the Quine-McCluskey algorithm.
- **Steps**:
    - It first finds the prime implicants.
    - Then, it creates the prime implicant chart and selects the essential prime implicants.
    - Finally, it converts the essential prime implicants into a Boolean expression.

---

9. **main()**

- **Purpose**: The entry point of the script, tying everything together.
- **Steps**:
    - It asks the user for the minterms and runs the Quine-McCluskey algorithm.
    - The essential prime implicants and minimized Boolean expression are printed.

---

10. **if __name__ == "__main__": Block**

- **Purpose**: Ensures that the main() function is only executed when the script is run directly, not when it's imported as a module.

Conclusions

The Python implementation of the Quine-McCluskey method provides an efficient and systematic approach to minimize Boolean functions with five variables. It can handle relatively complex logic minimization problems where traditional methods like Karnaugh maps are cumbersome.

Discussion

The QM technique, while effective for small to medium-sized functions, can become computationally expensive for a large number of variables or minterms. Optimizations can be explored in future versions of this project, such as heuristic-based reductions.

References

- Quine, W. V. "The Problem of Simplifying Truth Functions." American Mathematical Monthly, 1952.
- McCluskey, E. J. "Minimization of Boolean Functions." Bell System Technical Journal, 1956.
- Python Documentation. https://docs.python.org/

THE END