**quineMcCluskey ALgorithm**

By Ahmed Wasseem

and Ahmed ELbarbary

The American University in Cairo

Prof Mohamed Shalan

Digita design Course

Overview:

The quineMcCluskey class is the main class that drives the algorithm. It contains methods for grouping minterms, grouping implicants, combining minterms, and starting the algorithm. It also has methods for printing the truth table, getting the essential primes, and converting binary terms to their minterm equivalent.

The coveredBool class represents a minterm or prime implicant. It contains information about the minterm or prime implicant, including its binary representation and whether it is covered or essential.

The coverChart class is used to group prime implicants and determine the essential primes. It contains methods for building the chart, reducing the chart, and removing chart redundancy. It also has a method for getting the minimized expression.

The utils class contains independent functions that are used in the algorithm. It has checkers for verifying the syntax of a Boolean expression, methods for converting from decimal to binary and from minterm to binary, a method for counting the number of set bits in a given value, a method for extracting unique literals from an expression, and a method for parsing an expression into a normalized string.

Overall, the code is well-organized and easy to understand. It follows good object-oriented programming principles and uses appropriate data structures to represent the different parts of the algorithm. However, there is room for improvement in terms of readability and documentation. Some of the method names and variable names could be more descriptive, and there could be more comments explaining the purpose of each method and class.

1. quineMcCluskey:

The group\_minterms\_by\_bits() method groups the minterms based on their number of bits. The method creates a vector of vectors of coveredBool type, where each vector corresponds to a different number of bits. This method is called only once at the beginning of the algorithm's execution to initiate the algorithm's start.

The group\_implicants() method groups the prime implicants generated during the algorithm's execution. It creates a vector of vectors of coveredBool type, where each vector corresponds to a different number of ones (bits) in the prime implicants' binary representation. The method iterates over the vectors of coveredBool type generated by the previous method and combines them to get the minimal prime implicants. The method is called repeatedly until no more prime implicants can be combined.

The combine\_minterms() method combines two minterms to get a single coveredBool. The method compares the two minterms bit by bit and checks if they differ in only one bit. If they do, the method sets the different bit as a 'don't care' and returns a new coveredBool. If they differ in more than one bit, the method returns an empty coveredBool.

The get\_pos() method returns the sum of products (SoP) expression in a vector of strings. The method iterates over the prime implicants generated by the algorithm and converts them to their minterm equivalent. It then combines the minterms that correspond to the same prime implicant to get the final SoP expression.

The coveredBool\_to\_minterm() method converts a binary term to its minterm equivalent. The method first converts the binary term to its decimal equivalent and then gets the corresponding minterm from the truth table. The method returns a pointer to the resulting string.

The coveredBool\_bit\_difference() method returns the number of bit differences between two coveredBools. The method compares the two coveredBools bit by bit and increments a counter every time they differ in a bit.

The bit\_difference() method returns the number of bit differences between two integers. The method uses the bitwise XOR operator to get the bits that differ between the two integers and counts the number of set bits in the resulting value.

The build\_char\_table() method builds the truth table for the input Boolean expression. The method iterates over all possible combinations of the input variables and evaluates the expression for each combination. It stores the resulting values in a map where the key is the binary representation of the input variables.

The print\_table() method prints the truth table generated by the build\_char\_table() method. The method iterates over the map generated by the build\_char\_table() method and prints the input variables and the corresponding output value.

The set\_function() method sets the input Boolean expression for the algorithm. The method converts the normalized string representation of the expression to its binary representation and stores it in a vector.

Overall, the quineMcCluskey class implements the Quine McCluskey algorithm's essential parts and provides several utility methods to manipulate the input expression and its binary representation.

2. coveredBool:

The coveredBool class represents a minterm in binary form and its associated coverage status. It has two private variables that store the binary representation of the minterm and a boolean flag indicating whether it has been covered by a prime implicant or not.

The class has several methods that allow it to compare, set, and get its values. These methods include getters and setters for the binary representation and coverage status, as well as comparison methods for bitwise equality and coverage status.

Additionally, the class has a method to convert the binary representation to its minterm equivalent for ease of printing. It also has an overloaded << operator to print the coveredBool object directly to the console.

Overall, the coveredBool class is a simple yet necessary component in the Quine-McCluskey algorithm as it provides a convenient way to represent minterms and track their coverage status during the algorithm's execution.

3. coverChart:

The coverChart class is an important component of the Quine-McCluskey algorithm. It is responsible for storing the prime implicants generated during the algorithm's execution and for reducing the chart to its minimal form. The class has two private variables: one that stores the prime implicants and another that stores the minterms' inclusions.

The coverChart class has several methods that allow it to perform its functions. The first method is the build\_chart method, which takes an integer argument representing the number of bits in the minterms. This method iterates over the prime implicants and bitmasks them to create a binary representation of each minterm's inclusions. The resulting binary numbers are then stored in the chart map, where the keys are the binary numbers and the values are the decimal representations of the minterms that correspond to each binary number.

The next method is the reduce\_chart method, which attempts to reduce the chart by removing rows and columns that are redundant. The method first attempts to remove columns that have a single 1 entry. These columns correspond to minterms that are covered by a single prime implicant, which means that they are essential. The method stores these essential prime implicants in a vector called essential\_primes. If the chart has no essential prime implicants, the reduce\_chart method attempts to remove rows that are dominated by other rows. A row is dominated by another row if it has fewer 1 entries and has the same or more 0 entries. If the method is successful in reducing the chart, it returns true. Otherwise, it returns false.

The next method is the remove\_chart\_redundancy method, which attempts to remove rows and columns that are redundant. This method first attempts to remove columns that are dominated by other columns. A column is dominated by another column if it has fewer 1 entries and has the same or more 0 entries. If the chart has no redundant columns, the method attempts to remove rows that are dominated by other rows. A row is dominated by another row if it has fewer 1 entries and has the same or more 0 entries. If the method is successful in removing redundant rows and columns, it returns true. Otherwise, it returns false.

The next method is the get\_minimized\_expression method, which returns the minimal form of the Boolean expression represented by the chart. This method first checks if the chart has any essential prime implicants. If it does, it constructs the minimal form of the expression using only the essential prime implicants. If the chart has no essential prime implicants, the method constructs the minimal form of the expression using a three-step heuristic. The three-step heuristic involves selecting a set of prime implicants that cover all of the minterms, selecting a subset of those prime implicants that cover as many minterms as possible, and then selecting a subset of those prime implicants that have the fewest literals.

The final method is the print\_chart method, which prints the chart to the console. This method is mainly used for debugging purposes and is not essential to the algorithm's operation.

In summary, the coverChart class is an essential component of the Quine-McCluskey algorithm. It stores the prime implicants generated during the algorithm's execution and reduces the chart to its minimal form. The class has several methods that allow it to perform its functions, including the build\_chart method, the reduce\_chart method, the remove\_chart\_redundancy method, the get\_minimized\_expression method, and the print\_chart method.

4. utils:

The Quine McCluskey algorithm is a powerful tool for minimizing Boolean expressions. The implementation of the algorithm in the given code is efficient and robust. However, there are some areas where the code can be improved.

One improvement that can be made is to optimize the code for speed. The algorithm's time complexity is O(2^n), where n is the number of unique literals in the Boolean expression. While this is an inherent limitation of the algorithm, there are some optimizations that can be made to reduce the running time. For example, the algorithm can be parallelized to take advantage of multi-core processors. Additionally, the algorithm can be optimized to reduce the number of unnecessary comparisons and operations.

Another improvement that can be made is to handle edge cases more gracefully. For example, the code does not handle expressions with empty parentheses or invalid characters. These cases should be detected and handled properly to prevent the algorithm from crashing or producing incorrect results.

The code could also be made more modular and easier to understand. The code currently contains several classes and functions that are tightly coupled, making it difficult to modify or extend the code. By breaking the code into smaller, more modular components, it would be easier to maintain and extend in the future.

Finally, the code can be improved by adding more error checking and input validation. For example, the code could check if the input expression is a valid canonical sum of products expression before attempting to minimize it. This would prevent the algorithm from running on invalid input and producing incorrect results.

Overall, the Quine McCluskey algorithm implemented in the given code is a solid implementation that produces correct results for most inputs. However, there is room for improvement in terms of speed, modularity, error checking, and input validation.

5. Driving code quineMcCluskey(start)

The start function is the main driving code for the Quine McCluskey algorithm. It begins by prompting the user to choose between using a test case or inputting their own Boolean function. If the user selects a test case, the function retrieves a pre-defined Boolean expression and the number of unique literals used in the expression. The test cases are pre-defined Boolean expressions that are used to test the algorithm's functionality. These test cases are designed to evaluate the correctness of the algorithm's output when applied to various Boolean expressions.

If the user chooses to input their own function, the function prompts the user to enter the Boolean function in sum of products form and the number of unique literals used in the expression. The user is required to enter the Boolean function in the standard sum of products form, where each product term is separated by a plus sign (+), and each literal is a variable or its negation.

After retrieving the Boolean expression and the number of unique literals, the function creates a vector to store the function and a set to store the unique literals used in the expression. The vector is used to store the Boolean expression, while the set is used to store the unique literals used in the expression. The unique literals are identified by their alphabetic representation, starting from 'A'.

The function then generates a column array by grouping the minterms by their number of bits and then grouping the implicants by their number of ones. The column array is used to identify the prime implicants in the Boolean expression. The grouping of minterms and implicants is the core of the Quine McCluskey algorithm. The algorithm groups the minterms and implicants by their number of bits and ones, respectively, to identify the common terms that can be combined to minimize the Boolean expression.

The function then prints the original Boolean expression, the minimized Boolean expression in product of sums form, and the minimized Boolean expression in sum of products form. It also prints a truth table for the original Boolean expression and identifies the prime implicants and the essential prime implicants. The truth table is used to evaluate the Boolean expression for all possible input values of the variables. The prime implicants are the minimal terms that cover all minterms of the Boolean expression. The essential prime implicants are the prime implicants that cover at least one minterm that is not covered by any other prime implicant.

Finally, the function prompts the user to choose whether they want to try another function or exit the program. If the user chooses to try another function, the function starts again from the beginning. Otherwise, the program exits. The start function is the main driving code for the Quine McCluskey algorithm and provides a user-friendly interface for minimizing Boolean expressions.

Conclusion:

The Quine McCluskey algorithm is a powerful tool for minimizing Boolean expressions. It can handle expressions of up to 26 unique literals and can produce correct results for various expressions. In this article, we will delve deeper into the algorithm's implementation, its time and space complexity, and how it handles different expressions.

The algorithm's implementation starts with grouping the given minterms according to the number of bits they have. The algorithm then iterates over each group of minterms, comparing each minterm with the others in the group to find the prime implicants. The algorithm then groups the prime implicants according to the number of bits they have and proceeds to simplify the expression by removing redundant implicants.

The algorithm's time and space complexity are O(2^n), where n is the number of unique literals in the expression. The actual running time and space depend on the number of minterms and prime implicants generated during the algorithm's execution. The algorithm's time complexity is high, but it produces the most simplified expression possible.

The Quine McCluskey algorithm can handle various expressions, but it requires a bit of preprocessing to ensure that the expression is in its canonical form. The canonical form has the literals sorted alphabetically, and the minterms are in increasing order. The algorithm works on the assumption that the given expression is in its canonical form, and any deviation from this can result in incorrect results.

In conclusion, the Quine McCluskey algorithm is a powerful tool for minimizing Boolean expressions. Its implementation is efficient and robust, and it can handle various expressions. However, it requires the expression to be in its canonical form, and its time and space complexity are high. Nonetheless, the algorithm produces the most simplified expression possible, making it an invaluable tool for digital design engineers.