ESE 556 - VLSI Physical and Logic Design Automation

Two Level Logic Synthesis

Submitted by

Thomas Kral (111558962)

Felix Chimbo Cungachi (113470260)

Karan Rajendra (115375145)

1. Abstract

This project addresses the optimization of Boolean functions using Two-Level Logic Synthesis, aimed at reducing the complexity of digital circuits. The primary method of optimization utilized is the branch-and-bound algorithm, supplemented by Quine's method for computing prime implicants. This approach allows for the systematic reduction of logic expressions to their minimal forms, ensuring efficient circuit design with potentially lower costs and power consumption.

The implementation handles up to 10 different variables across a set of 10 logic equations, making it applicable to a range of digital systems from small-scale integrated circuits to more complex combinational logic designs. The program begins by converting the provided minterms and optional don't-care conditions into their binary representations. These are then processed to group and combine minterms based on the number of '1's they contain, iterating to derive prime implicants.

A prime implicant chart is constructed to map each minterm to its respective implicants, facilitating the identification of essential prime implicants necessary for the minimal Boolean expression. The branch-and-bound technique is meticulously applied to further minimize the number of implicants, aiming for the most cost-effective solution.

This approach not only offers a practical application in digital electronics design but also serves as an educational tool for understanding the underlying processes of logic minimization. The outcomes of this project indicate that using a combination of the Quine method for initial implicant generation and branch-and-bound for optimization effectively minimizes Boolean expressions, leading to simpler and more manageable digital circuits.

2. Introduction

In the realm of digital electronics, the design and optimization of logic circuits are crucial for developing efficient, high-performance systems. Digital circuits are foundational in various applications, ranging from simple logic gates in consumer electronics to complex combinational circuits in advanced computing systems. An essential aspect of designing these circuits involves minimizing the Boolean expressions that describe their functionality. Minimization not only simplifies the circuit design but also reduces the cost, size, and power consumption of the resulting hardware.

Boolean Function Minimization Concepts

Minimization of Boolean functions involves reducing the complexity of Boolean expressions while ensuring they remain logically equivalent to the original expressions. This process is crucial in the design of digital logic circuits and can significantly impact their efficiency and performance. Key concepts in this process include implicants, prime implicants, and essential prime implicants:

- **Implicants:** An implicant of a Boolean function is a product term (a conjunction of literals) that implies the function. This means that whenever the implicant is true, the function is also true. Implicants are considered building blocks from which the function can be constructed.
- **Prime Implicants:** A prime implicant is an implicant that cannot be covered by a more general (less specific) implicant. In other words, it is a minimal implicant that still ensures the function's truth. Prime implicants are critical because they represent the simplest individual terms that cover all necessary conditions of the function without redundancy.
- **Essential Prime Implicants:** An essential prime implicant is a prime implicant that covers an output of the function that no other prime implicant covers. These are particularly important as they must be part of the final expression to ensure all outputs of the function are correctly realized. Identifying these implicants helps in significantly reducing the complexity of the Boolean expression by ensuring that only necessary and sufficient terms are included.

Approach and Methodology

This project introduces a sophisticated approach by incorporating the branch-and-bound algorithm into the logic minimization process. This algorithm is widely recognized in operations research for solving various optimization problems and is particularly adept at reducing the solution space effectively, thereby enhancing the efficiency of the minimization process.

The branch-and-bound method, coupled with Quine's technique for deriving prime implicants, provides a robust framework for deriving minimal expressions for Boolean functions. This project specifically targets the minimization of Boolean expressions containing up to 10 variables, making it highly relevant for both educational purposes and practical applications in circuit design. Through the application of this combined approach, the project aims to demonstrate a significant reduction in the complexity of logic circuits, underlining the practical benefits of these computational techniques in digital circuit design.

This report details the methodology implemented, explores the theoretical underpinnings of the algorithms used, and evaluates the effectiveness of the branch-and-bound algorithm in the context of two-level logic synthesis. By the conclusion, the reader will gain insights into how advanced computational techniques can be leveraged to optimize digital circuits, reflecting on the broader implications for electronics design and optimization.

3. Problem Statement

The task at hand is digital logic minimization. The approach to this problem can be broken up into two different ‘steps,’ or algorithms. The first step is to apply the Quine-McClusky algorithm to find prime implicants and essential prime implicants. The final step is to take this generated solution space of prime implicants and find the optimal selection that provides a minimal covering. The rest of this section will go into further detail of what these steps mean, and how they should operate.

The Quince-McClusky program will find prime implicants and essential prime implicants from the set of minterms found, and this process will be shown further in section 5. Once we have our set of prime implicants, including our essential prime implicants, we can feed this through to our Branch and Bound algorithm. The goal of branch and bound is to exhaustively try every combination of prime implicants so that all minterms are covered in our final equation, but the amount of variables used are minimized. This is desired since more variables will mean a repeated use of gates, resulting in a larger fan in/out, meaning larger parasitics and longer propagation delay.

4. Related Work

As previously discussed there will be two algorithms used to minimize the amount of variables used and those are the Quince-McClusky algorithm and the Branch and Bound algorithm. Another method that comes similar to this is the Karnaugh Map method where it essentially breaks down the boolean expressions in the form of a table where it gets paired up and can be reduced to get a simpler expression. This can retrieve both minterms and maxterms where it is also a visual representation of all the possible easy expressions that can be simplified[2]. Now to go into more detail of the Quince-McClusky algorithm which is shown to be a better version of the karnaugh map method where it has been used when it comes to designing digital circuits. Digital gates are basic electronic components and should be simplified to be able to reduce the number of gates a circuit will have for more efficiency. Vittha Jadhav and Amar Buchade[3] elaborated the idea of the Quince-McClusky algorithm by implementing their method that eliminates a variable's positional weight in mintermlist which is also known as E-sum. With this their goal is to increase the performance by reducing the number of comparisons which is done by their E-sum method. As a result they have managed to introduce a new method called Modified Quince-McCluskey (MQM) that in fact reduced the number of boolean functions and increased its performance.

Now they weren't the only ones trying to improve the algorithm as Adrain Duşa[4] discussed how the algorithm has many problems such as it being memory hungry and slow for a large number of conditions. Adrian's approach enhances the algorithm by following an exhaustive calculation that will get the same as the original algorithm where it will work vectors instead of complex matrices. As a result Adrian enhances and extends the Quince-McClusky by having it exceed the previous runtime that was the issue to begin with. Javad Safaei and Hamid Beigy[5] proposed a new algorithm that they called GQMC where 'G' stands for greedy. Their approach involves checking if there is an unchecked or checked essential prime implicant where depending if it is checked then all of the samples that have been covered will also be checked and when it is not then all it will enter another method where it will do the checking for it. As a result their method can be described as a classification method but due to their findings it was very slow. This shows how the Quince-McClusky algorithm can be involved in other methods that do not involve in it only benign for design.

Now let's discuss the branch and bound algorithm. The Brand and Bound Algorithm is a method used in optimization problems to systematically explore potential solutions by iteratively bounding the search space to efficiently converge towards the best solution available[7]. He He, Hal Daume III and Jason Eisner [8] both want to improve the algorithm by challenging themselves to design a node searching strategy on a branch and bound tree. The results of their findings lead to an adaptive node searching algorithm that can be used in branch and bound trees. David, Sheldon, Jason, and Edward[9] further the previous work by being able to use a tree search strategy to enumerate all possible solutions to a given problem at hand where they further elaborate on what can be improved upon the algorithm to get even better results. They explain that there are three algorithmic components that can be improved which are the search strategy, the branching strategy, and the pruning rules where they made a comprehensive study of the current algorithms with respect to each category as it details their strengths and limitations that are imposed on it.

5. Proposed Solution

As mentioned in section 3, the program will be split up into two subprograms. First the Quince-McClusky algorithm will be implemented so all prime implicants can be found. Then this shall be fed through to the Branch and Bound algorithm. A visual representation of this process is shown in Figure 1:

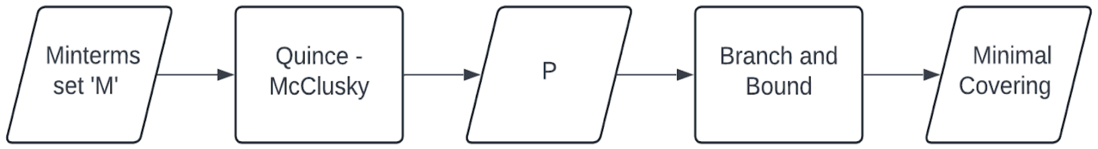


Figure 1. Overview of Digital Logic Minimization Process.

5.1 Quince-McClusky

Now that we've obtained a set of minterms to find prime implicants from, we may now begin our Quince-McClusky algorithm. Our first step is to convert our minterm set, M, into a set of binary strings. Then with these binary forms we're able to begin our grouping process, which is used to iteratively compare members of adjacent groups and 'match' binary strings with a difference of only one character. This gets repeated until no more reductions can be done, giving us our prime implicants. With this information, we can create a Prime Implicant chart and identify our essential prime implicants.

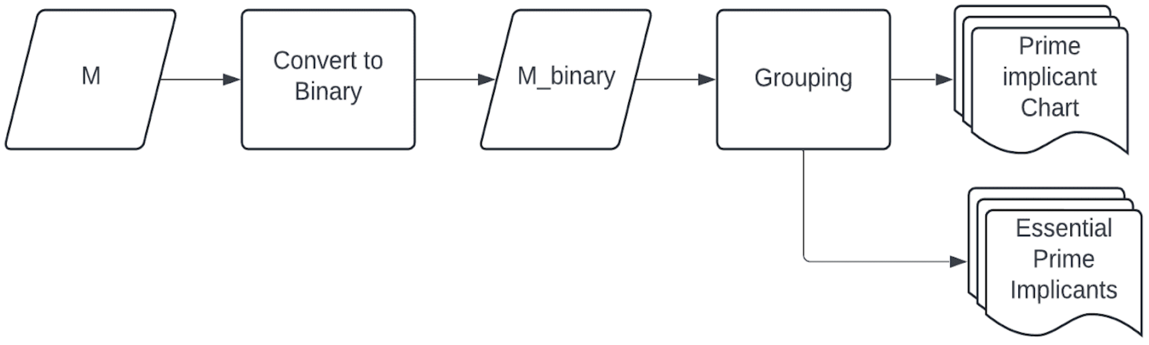


Figure 2. Overview of our implementation of Quince-McClusky

5.1.1 Conversion to Binary

First comes a method to find the maximum amount of bits needed to create all necessary minterms in binary, called findMaxBits(). This method can be found below in figure 3:

```
int findMaxBits(const vector<int>& numbers) {
    if (numbers.empty()) return 0;
    int maxNum = *max_element(numbers.begin(), numbers.end());
    return ceil(log2(maxNum + 1));
}
```

Figure 3. findMaxBits()

So long as there are minterms contained within the set, the max element will be found and stored in our maxNum variable, and the ceil() function from the <cmath> library will be used to find the required amount of bits. The amount of bits needed for an integer is found with logarithm base 2, one is added in the case that if the maxNum is a power of two, we will need an extra bit to represent that, i.e. log2(8) = 3, however 3 bits only covers 0-7. The ceil function is then applied so that if the result is a decimal it is rounded up to the nearest integer. An example of this is that log2(7)~=2.807, therefore 3 bits are appropriate.

With the maximum number of bits needed, a quick loop is called that iterates through the minterm set and creates a conversion for each to binary using the toBinary() function, as seen in figure 4.:

```
string toBinary(int number, int digits) {
    string binary = bitset<32>(number).to_string();
    return binary.substr(32 - digits);  // adjust length as necessary
}
```

Figure 4. toBinary()

Here the <bitset> header is used to convert from an integer to a binary string. First the number is entered into the “bitset<32>(number).to_string()” which will create the corresponding bitset. Then the next line just returns a substring of the binary string so that the appropriate amount of digits are present. The maxBits value found from the findMaxBits() function is used to determine the string length.

5.1.2 Grouping

We used two functions for the process of grouping, groupMinterms() to obtain initial groups, then processGroups() where the algorithm is applied. groupMinterms() is found below in figure 5:

```
map<int, vector<string>> groupMinterms(const vector<string>&
binaryMinterms) {
    map<int, vector<string>> groups;

    for (string minterm : binaryMinterms) {
        int countOnes = count(minterm.begin(), minterm.end(), '1');
        groups[countOnes].push_back(minterm);
    }

    // Debug: print groups for verification
    cout << "Grouping based on count of '1's in binary representation:\n";
    for (const auto& group : groups) {
        cout << "Group " << group.first << " (" << group.second.size() << "
elements): ";
        for (const auto& bin : group.second) {
            cout << bin << " ";
        }
        cout << endl;
    }

    return groups;
}
```

Figure 5. groupMinterms()

The function takes in our previously made vector of minterm binary strings. The output is to be a mapping of the count of 1s to a vector of binary strings, groups. The first for loop iterates through each string in our input vector and obtains a count of the times a '1' appears in the string. This is done with the count() function from the <algorithm> library. This method works in that it takes a beginning and ending iterator value as the first two parameters, and a value for the third parameter. The function will then iterate between the given iterators and return a count for each time the value in the third parameter is found. The rest of the function is so the initial group is printed in an understandable manner.

Once we have our initial map of our groups the processGroups() function is ran as seen in figure 6 below:

```
set<string> processGroups(map<int, vector<string>>& groups) {
    set<string> primeImplicants;
    map<int, vector<string>> newGroups;
    bool progress = false;
    int count = 0;
    while (true) {
        progress = false;
        newGroups.clear();

        for (auto it = groups.begin(); it != groups.end(); ++it) {
            auto next = std::next(it);
            if (next == groups.end()) break;

            for (const auto& current : it->second) {
                for (const auto& nextMinterm : next->second) {
                    auto combineResult = tryCombine(current, nextMinterm);
                    if (combineResult.first) {
                        // Check if this new minterm is already added to
avoid duplicates
                        if (!newGroups[it->first - 1].empty() &&
                            find(newGroups[it->first - 1].begin(),
newGroups[it->first - 1].end(), combineResult.second) ==
newGroups[it->first - 1].end()) {
                            newGroups[it->first -
1].push_back(combineResult.second);
                        }
                        else if (newGroups[it->first - 1].empty()) {
                            newGroups[it->first -
1].push_back(combineResult.second);
                        }
                        progress = true;
                    }
                }
            }
        }

        // If no progress can be made, break and add remaining minterms as
prime implicants
        if (!progress) {
            for (const auto& group : groups) {
                for (const auto& minterm : group.second) {
                    // Ensure no duplicates
                    if (primeImplicants.find(minterm) ==
primeImplicants.end()) {
                        primeImplicants.insert(minterm);
                    }
                }
            }
            break;
        }

        // Set up groups for the next iteration
        count++;
        cout << "Grouping " << count << ": " << endl;
        for (const auto& group : groups) {
            cout << "Group " << group.first << " (Count " <<
group.second.size() << "): ";
            for (const auto& bin : group.second) {
                cout << bin << " ";
            }
            cout << "\n";
        }
        groups = newGroups;
    }

    return primeImplicants;
}
```

Figure 6. processGroups()

To begin we'll start with explaining what our variables mean and the ideas behind them, then get into a more detailed explanation of the function. Our desired output is a set of strings called our prime implicants. As the function carries out and minterms are matched to each other, with an example result as '0-10,' and no more matches can be found, the primeImplicants set will save the remaining resulting strings. The newGroups map will be used to update the state of the grouping after all matchings are found for a set of groups. The progress boolean will be used in the while loop, and will indicate whether or not more matches can be found in the current groupings for each iteration. The count integer is only used for printing purposes.

And so begins our while loop. Progress starts out as false until proven otherwise and newGroups is cleared so it may be filled with potential matchings. From here there is a triply nested loop. What this does is iterate through each group, then within each group will iterate through the members of that group, and for each of these members there is a third for loop that will iterate through each member of the next group so long as there is a next group. For each member of each group, we will compare it to each member of the next group, and if we're able to match it we'll save the resulting string to it's corresponding new group in the newGroups map, i.e. a match from a member in group 1 and group 2 will be placed into newGroups. Also when a resulting match is placed, progress is set to true and the loop will attempt another iteration. After checking for results from all groups, if no progress has been made the remaining prime implicants all groups are iterated through and added to the output set primeImplicants. The rest of the function just prints out the image of groups for each iteration.

To further clarify how we find whether a match can be made, we wrote a function called tryCombine() as seen figure 7:

```
pair<bool, string> tryCombine(const string& a, const string& b) {
    if (a.length() != b.length()) {
        return { false, "" };  // Ensure both strings are of equal length
    }

    int countDifferences = 0;
    string result = a;

    for (size_t i = 0; i < a.length(); ++i) {
        if (a[i] != b[i]) {
            countDifferences++;
            result[i] = '-';
            if (countDifferences > 1) {
                return { false, "" };  // Not combinable if more than one
difference
            }
        }
    }

    return countDifferences == 1 ? make_pair(true, result) :
make_pair(false, "");
}
```

Figure 7. tryCombine() method.

The first if statement is to ensure no errors have occurred for debugging purposes. Then we create a variable to count the amount of differences between the two strings, and a result string to save and return the resulting match if found. The program then iterates through string a, if there is a difference between the strings at the same index, the count is incremented and the resulting string saves a hyphen at the index to show the digits are canceled out. If there is more than one difference then a false bool and an empty string are returned as output to indicate the strings cannot be matched. If there is only one difference, a true boolean paired with the generated result string is returned.

Once we have all our minterms, we can create our prime implicant chart, which will be a mapping of minterms as keys to a vector of prime implicant strings that cover that minterm. The following snippet of code is used in our main() method to achieve this in figure 8:

```
// Create a prime implicant chart
    map<string, vector<string>> chart;
    for (int minterm : allMinterms) {
        string binaryMinterm = toBinary(minterm, maxBits);
        for (const auto& implicant : primeImplicants) {
            if (isCovered(binaryMinterm, implicant)) {
                chart[binaryMinterm].push_back(implicant); //chart consists
of each prime implicant with it's minterms(binary)
            }
        }
    }
```

Figure 8. Code to create prime implicant chart/map.

First we create our map, then we iterate through each minterm in our set of found minterms, ‘allMinterms,’ for each minterm we check if that prime implicant covers our minterm, and if so we’ll add it to the vector. The code for the boolean method isCovered() is found in the next figure, figure 9:

```
bool isCovered(const string& minterm, const string& implicant) {
    // Check each bit in the minterm against the implicant
    for (size_t i = 0; i < minterm.size(); ++i) {
        // If implicant bit is not '-' and does not match minterm bit,
return false
        if (implicant[i] != '-' && implicant[i] != minterm[i]) {
            return false;  // Implicant does not cover the minterm at this
bit
        }
    }
    return true;  // All bits match or are covered by don't care positions
in implicant
}
```

Figure 9. isCovered() function.

The logic behind this is that as the grouping goes matches will represent two different entries from the prior groups. An example being 1100 and 1000 will result in 1-00, meaning if 1-00 is a prime implicant it covers both 1100 and 1000. This also means if there is further matching, maybe an implicant is 1--0, this also encompasses minterms 1100 and 1000 among others. This pretty much does this directly by iterating along the minterm string range, and comparing the values at the indices between the implicant and the minterm so long as the value is not '-.' If there is a difference found, return false, otherwise return true.

Another feature of the Quince-McClusky algorithm is the finding of the essential prime implicants. For this we created the method, findEPI() found in figure 10:

```
vector<string> findEPI(const map<string, vector<string>>& chart) {
    vector<string> essentialPrimeImplicants;
    // Iterate over each minterm's list of implicants
    for (const auto& minterm : chart) {
        if (minterm.second.size() == 1) {  // If exactly one implicant
covers this minterm
            string soleImplicant = minterm.second.front();
            if (find(essentialPrimeImplicants.begin(),
essentialPrimeImplicants.end(), soleImplicant) ==
essentialPrimeImplicants.end()) {
                essentialPrimeImplicants.push_back(soleImplicant);
            }
        }
    }
    return essentialPrimeImplicants;
}
```

Figure 10. findEPI() function.

The logic inspiring the function is what was talked about during lecture, in the case where a minterm is covered by only one prime implicant, that implicant is considered essential. This is easy since we've already made a prime implicant map mapping each minterm to a list of prime implicants. In this case if a vector has a size of 1, then the implicant contained in that vector is essential.

5.2 Branch and Bound

The branch and bound implementation was based on the pseudo code presented in class created by Hatchel and Somenzi [1]. This implementation is made up of three main functions, being the

BCP(), MIS_quick(), and Reduce() functions. I will first go through Reduce and MIS_quick, as they are subfunctions of BCP.

5.2.1 Reduce

When the method is first called, we are in the root node. The first step for this node is to reduce the prime implicant chart using row and column dominance as discussed in class. The theorem to be followed was also explained in class, cited again from Hatchel and Somenzi [1]. I am going to follow the jargon used in class, so each row will describe each prime implicant and which minterms its covers, and each column will represent a minterm and each prime implicant it is covered by. The Reduce() method is shown below in figure 11:

```
map<string, vector<string>> Reduce(const map<string, vector<string>>&
chart) {
    map<string, vector<string>> reducedChart = chart;

    //Col dominance: (if a minterm has same implicants as another and more
it can be removed)
    set<string> keysToRemove;
    for (auto ci : reducedChart) {
        set<string> ciImplicants(ci.second.begin(), ci.second.end());
        for (auto cj : reducedChart) {
            if (ci.first != cj.first) {
                set<string> cjImplicants(cj.second.begin(),
cj.second.end());
                if (keysToRemove.find(cj.first) == keysToRemove.end() &&
!cj.second.empty() && includes(ciImplicants.begin(), ciImplicants.end(),
cjImplicants.begin(), cjImplicants.end())) { //if the implicants of cj are
a subset of ci:
                    keysToRemove.insert(ci.first);
                    break;
                }
            }
        }
    }
    for (string key : keysToRemove) {
        reducedChart.erase(key);
    }
    keysToRemove.clear();

    //Row dominance:
    //first create flipped chart:
```

```
    set<string> pitoremove;
    map<string, vector<string>> fchart = flipChart(reducedChart);
    for (auto ri : fchart) {
        set<string> riMinterms(ri.second.begin(), ri.second.end());
        for (auto rj : fchart) {
            if (ri.first != rj.first) {
                set<string> rjMinterms(rj.second.begin(), rj.second.end());
                if (keysToRemove.find(ri.first) == keysToRemove.end() &&
!ri.second.empty() && includes(riMinterms.begin(), riMinterms.end(),
rjMinterms.begin(), rjMinterms.end())) pitoremove.insert(rj.first); break;
            }
        }
    }
    for (string key : pitoremove) {
        for (auto pair : reducedChart) {
            if (find(pair.second.begin(), pair.second.end(), key) !=
pair.second.end()) {
                keysToRemove.insert(pair.first);
            }
        }
    }
    for (string key : keysToRemove) {
        reducedChart.erase(key);
    }

    return reducedChart;

}
```

Figure 11. Reduce() method.

Within the function there are two sets of nested for loops, and a set of strings that are to be deleted from the prime implicant chart. To reiterate, the input is our prime implicant chart, and is a map that uses a string representation of a minterm as a key, and these keys are mapped to a vector of prime implicants in a string form.

The first nested for loop, as indicated by the comments, is to assert column dominance, since the input map is already fitting for this. To put the nested for loop into simpler terms, for each key/minterm in the chart we shall create a set ‘ciImplicants’ containing all prime implicants mapped to the key/minterm, then for each other column in the map that is not equal to ci, we will see if the same set of implicants mapped to cj are a subset of the set from ci, then the ci key is recorded to be removed. In order to check if the cj set is a subset of the ci set, the includes()

function, included in the C++ Standard Library, is used. This function works with the four parameters, all of which are iterators. The first two parameters are the beginning and ending iterators of the first list, and the third and fourth parameters are the beginning and ending iterators of the second set. The function returns a boolean value indicating whether the range between the third and fourth parameters are included in the range indicated by the first and second parameters. The if conditional statement requires two other conditions to be satisfied besides the set of cj being a subset of the set of ci. These conditions are that the cj set is not empty, and the other is whether the key found is already included in the set. Checking whether the cj set was empty was used for error prevention, and checking whether the element is already included in the set was later found to be redundant, since the set object will not save duplicates of the same value.

This follows the theorem in that if ci has all the 1s of cj, then ci is covered when cj is covered and can be removed. To reiterate the intuition behind this as described in class the following example can be used. If m0 is covered by p0 and p1, and m1 is covered by only p1, then we can rewrite this with the following equations 1 and 2:

$$m_0 : p_0 + p_1 = 1 \tag{1}$$

$$m_1 : p1 \ = \ 1 \tag{2}$$

Looking at these equations we can see that p1 must be present in order to cover m1, therefore the value of p0 doesn't matter since m0 must be covered due to p1. After the column dominance is asserted, and we've saved the keys that are being dominated by other columns, we iterate through the prime implicant chart and remove all keys found in our 'keysToRemove' set. The keysToRemove set is then cleared as we begin asserting our row dominance.

Before we began applying the row dominance rules, our idea was to make the prime implicant chart easier to iterate through so the focus could be on iterating through the prime implicants rather than the minterms, since the minterms are currently being used as keys. To achieve these we created a small function flipChart() to take our minterm map as input, and return a prime implicant map. The function is found below in figure 12:

```
map<string, vector<string>> Reduce(const map<string, vector<string>>&
chart) {
    map<string, vector<string>> reducedChart = chart;

    //Col dominance: (if a minterm has same implicants as another and more
it can be removed)
    set<string> keysToRemove;
    for (auto ci : reducedChart) {
        set<string> ciImplicants(ci.second.begin(), ci.second.end());
        for (auto cj : reducedChart) {
            if (ci.first != cj.first) {
                set<string> cjImplicants(cj.second.begin(),
cj.second.end());
                if (keysToRemove.find(cj.first) == keysToRemove.end() &&
!cj.second.empty() && includes(ciImplicants.begin(), ciImplicants.end(),
cjImplicants.begin(), cjImplicants.end())) { //if the implicants of cj are
a subset of ci:
                    keysToRemove.insert(ci.first);
                    break;
                }
            }
        }
    }
    for (string key : keysToRemove) {
        reducedChart.erase(key);
    }
    keysToRemove.clear();

    //Row dominance:
    //first create flipped chart:
    set<string> pitoremove;
    map<string, vector<string>> fchart = flipChart(reducedChart);
    for (auto ri : fchart) {
        set<string> riMinterms(ri.second.begin(), ri.second.end());
        for (auto rj : fchart) {
            if (ri.first != rj.first) {
                set<string> rjMinterms(rj.second.begin(), rj.second.end());
                if (keysToRemove.find(ri.first) == keysToRemove.end() &&
!ri.second.empty() && includes(riMinterms.begin(), riMinterms.end(),
rjMinterms.begin(), rjMinterms.end())) pitoremove.insert(rj.first); break;
            }
        }
    }
    for (string key : pitoremove) {
        for (auto &pair : reducedChart) {
            auto rem = find(pair.second.begin(), pair.second.end(), key);
            if (rem != pair.second.end()) {
                pair.second.erase(rem);
            }
        }
    }

    return reducedChart;
```

```
}
```

Figure 12. flipChart() function.

First we iterate through each entry in the map, then for each element in the key's corresponding vector of prime implicant strings, the prime implicants are added to the set 'implicantsToAdd.' After we've found this set, we'll iterate again through our minterm map. In this second nested for loop, for each implicant string in our set, iterate through each vector in the original map, if that implicant is included in that vector, then add the vector respective key to the new map under the key that is the prime implicant. Then return the newly created map, which is our flipped chart.

Now that we have our flipped chart we can begin to assert row dominance in a more convenient way. The second nested for loop within the Reduce() function will iterate through each key in the flipped map, or row ri, then iterate through all other keys that are not equal to ri as rj. Then similarly to our column dominance the same conditions are applied, even checking whether the set from rj is a subset of the set from ri. The only difference in this case is that if the conditions are satisfied, then the key at rj will be added to the set to be removed. This is to follow the reasoning from the theorem, that if a prime implicant covers all the same minterms as another prime implicant and more, then why include the prime implicant with less minterms covered, hence rj is dominated by ri. After these prime implicants are identified, they can be iteratively removed from the reducedChart vectors so they aren't included in the final solution.

<u>5.2.2 MIS_quick</u>

The MIS_quick() method will be used to find a lower bound for the branch and bound algorithm. The purpose of the lower bound is to set a minimum cost for the final solution. As an example, if the Quince-McClusky program found that there were three essential prime implicants, then there must be at least three prime implicants included in the solution quite literally because they're essential. The pseudo code as described by Hachtel and Somenzi [1], is a similar process to the column dominance theorem described in the previous section, in that we shall iterate through our input table, find the minterm with the least amount of prime implicants covering it, add this column to our new MIS table, then remove all columns in the original table with common prime implicants. We then repeat this after until all columns have been removed, and return the size of the MIS table. The reason this works as a lower bound is that MIS stands for Maximal Independent Set, and will return the shortest columns that are all independent of each other, since dependent entries are deleted during the process. Our implementation of MIS_quick is found below in figure 13:

```
int MIS_quick(const map<string, vector<string>>& chart) {
    map<string, vector<string>> chartcopy = chart;
    set<string> keys;
    for (auto pair : chartcopy) {
        if (pair.second.empty()) keys.insert(pair.first);
    }
    for (string key : keys) {
        chartcopy.erase(key);
    }
    map<string, vector<string>> MIS;
    do {
        int minimplicants = INT_MAX;
        string key = "";
        for (const auto& pair : chartcopy) {
            if (pair.second.size() < minimplicants) {
                minimplicants = pair.second.size();
                key = pair.first;
            }
        }
        if(chartcopy.size() != 0) MIS[key] = chartcopy[key];
        vector<string> keysToRemove;
        for (string implicant : MIS[key]) {
            for (const auto& pair : chartcopy) {
                if (find(pair.second.begin(), pair.second.end(), implicant)
!= pair.second.end()) {//if a minterm is convered by implicant, remove
                    //chartcopy.erase(pair.first);
                    keysToRemove.push_back(pair.first);
                }
            }
            for (auto key : keysToRemove) {
                chartcopy.erase(key);
            }
        }
    } while (!chartcopy.empty());

    return MIS.size();

}
```

Figure 13. MIS_quick() implementation.

Before our function truly begins, some cleanup is done in that we iterate through our chart, and find any minterms that at this time have an empty vector of prime implicants, this may occur at any point during BCP function since when branches are made from a prime implicant node that prime implicant will be wiped from the chart since it no longer needs to be considered, these features will be further explained in the next section. After this cleanup, we create a do-while loop that checks for whether the chart still contains some elements, since once the chart is empty the functions job is done. Within this do statement, we create the 'minimplicants' and 'key' variables to keep track of the smallest found count in a column and a key of said column respectively. Then the chart is iterated through and this minimum is recorded within these variables. There is then an additional if statement that makes sure that charcopy is not an empty map, and if this condition is satisfied then the original chart entry of the recorded minimum will be copied to the MIS map. Then for each prime implicant included in our new MIS entry, each pair in the original chart map is iterated through, and if it's vector contains the chosen prime implicant, then the key of that map entry will be recorded into keysToRemove set so that it will be remove after the loop is done. The reason this set of keys is created, and then needs to be deleted after the first for loop is that if it is done during the for loop then it will generate a running error since once we delete the entry, the map value the iterator points to will become inaccurate. Then this process is repeated until the original chart is empty, and finally the size of the MIS table is returned.

5.2.3 BCP

The code for BCP() is found in figure 14 below:

```
vector<string> BCP(map<string, vector<string>> &chart, int U,
vector<string> &currentSolution) {
    map<string, vector<string>> rchart = Reduce(chart);

    if (isTerminal(rchart)) { //if terminal case
        if (Cost(currentSolution) < U) {
            U = Cost(currentSolution);
            if (currentSolution.empty()) return { "no solution" };
            else return currentSolution;
        }
        else {
            return { "no solution." };
        }
    }
    int LB = MIS_quick(rchart);
    if (LB >= U) {
        return { "no solution" };
    }

    string pi;
    map<string, vector<string>> rchart1 = rchart;
    map<string, vector<string>> rchart0 = rchart;
    for (auto pair : rchart) {
        if (!pair.second.empty()) {
            pi = pair.second.at(0); //choose a pi
            break;
        }
    }

    //set pi to 1 and branch (any chart entry with pi becomes (1 + pj),
therfore remove these entries completely and add to currentSolution)
    set<string> keys;
    for (auto pair : rchart1) {
        for (string s : pair.second) {
            if (s == pi) keys.insert(pair.first); break;
        }
    }
    for (auto &pair : rchart1) {
        if (keys.find(pair.first) != keys.end()) {
            pair.second.clear();
        }
    }
    keys.clear();
    vector<string> cs1 = currentSolution;
    cs1.push_back(pi);
    auto s1 = BCP(rchart1, U, cs1);
    if (Cost(s1) == LB) return s1;
    //set pi to 0 and branch (remove pi (0 + pj) and DONT add to current
solution)
    for (auto &pair : rchart0) {
        pair.second.erase(remove_if(pair.second.begin(), pair.second.end(),
[&pi](string s) {return s == pi; }), pair.second.end()); //remove pi
    }
    auto s0 = BCP(rchart0, U, currentSolution);
    return BestSolution(s0, s1);
}
```

Figure 14. BCP() function.

The BCP function itself is recursive. The idea is that it emulates a binary decision diagram, which will be identified later in the explanation. Just to clarify, in this case of a binary decision diagram, each node of the graph represents a prime implicant, and this node will branch out to two other nodes based on whether this prime implicant is chosen to be used or not in the final solution. This is what makes the branch and bound algorithm an exhaustive solution.

The trick to implementing this function was translating from the pseudo code presented in class, referencing Hachtel and Somenzi [1]. The inputs in the pseudo code are F, U, and currentSolution. U and currentSolution seemed straightforward, as U keeps track of the minimum seen cost, and currentSolution keeps the running list of prime implicants that have been decided to be included at this point in the branching. F in the pseudo code was described by Hachtel and Somenzi to be the left hand side of the constraint equation, which took us some time to understand the meaning. After some digging in the text, we found that this constraint equation is a product of the minterm prime implicant equation shown in equations 1 and 2. The resulting constraint equation from m0 and m1 in equations 1 and 2 can be found in equation 2.

$$(p_0)(p_0 + p_1) = 1 \tag{3}$$

F in our case is called 'chart,' where we just used the same chart we've been using, since each minterm has a vector of prime implicant, which is an acceptable representation of its corresponding equation in the left hand side of the constraint equation.

The first primary function of the code should be, as described by Hachtel and Somenzi, is that F and current solution should be reduced. We achieve the same by applying our Reduce() function on 'chart.' The second function described should find if the current state/node is a terminal case, and if so it should check if the current cost is less than the current running minimum. If so, update the minimum to match and return the currentSolution. If this condition is untrue, then as described in the third line, return no solution. Our implementation almost directly follows this, the only added feature is that we check if the currentSolution is empty then we must return our definition of no solution, being the vector "{ "no solution" }." This was added as an extra precaution to make sure a cost of 0 couldnt overpower something above the lower bound, however it is not totally necessary since this is the whole purpose of the next line described in the text. Our Cost() function simply returns the size of currentSolution, since this indicates the amount of implicants included in the solution.

The fourth line says to find a lower bound with some LOWER_BOUND() function, which in our case will be MIS_quick. Then if this found lower bound is greater than or equal to the current upper bound, or U, then we must return no solution, since a solution with fewer prime implicants than the lower bound means that not all essential prime implicants could have been included, therefore making it an invalid solution. Our implementation just follows this idea, there are no extra features that were required. This checking of the lower bound introduces a pruning to our binary decision tree so that there is no time wasted exploring branches that won't' provided with a better or valid solution.

The fifth line then is to choose some prime implicant from the remaining available prime implicants. This is achieved in our implementation when we create a string variable ‘pi,’ then iterate through the vectors in the current minterm map, and choose the first available prime implicant.

Then line 6 will make a recursive call, or branch, where this prime implicant is chosen as a 1 where it is included in the currentSolution. In line 7, if the cost of the result from this branch is found to be equal to the lower bound, then this is considered a perfect solution and will be returned with now further branching. In order to create the state that this prime implicant is chosen, a copy of the rchart is created as rchart1, indicating the 1 decision. If a prime implicant is chosen to be included in the currentSolution, then this means that any term, or vector, that includes this prime implicant is set to 1, and no longer need to be included in the equation/map. The reason it no longer needs to be included is that in the case p0 + p1, if p0 is set to 1 then the equation will result in 1 regardless of the value of p1. This removal is done by iterating through each vector in the map, and if the prime implicant is contained in that vector, then the vector is cleared. Since the prime implicant is chosen as a 1 that also means that it is added to the currentSolution. In our program we make a copy of currentSolution, ‘cs1,’ and input into the branch.

If this condition is not satisfied, then another branch will be made where the chosen prime implicant is set to zero and not included in the currentSolution. In this case, each vector is iterated through a different copy of the map rchart0, this time if a prime implicant is found in the vector, then only that prime implicant is removed rather than clearing the entire vector. An example to clarify this is that p1 + p0 where p0 is equal to 0 is equivalent to p1. There is no copy of currentSolution made since we are not adding anything to the currentSolution, since we are choosing to neglect this prime implicant.

After these two branches have been made, we then compare the cost of each branch and return the branch with the better cost using our BestSolution() function. This function can be seen in figure 15 below:

```
vector<string> BestSolution(vector<string> solution1, vector<string>
solution2) {
    if (solution1.at(0) == "no solution") return solution2;
    else if (solution2.at(0) == "no solution") return solution1;
    else return Cost(solution1) >= Cost(solution2) ? solution1 : solution2;
}
```

Figure 15. BestSolution() function.

The main goal of this function is to compare the costs of each solution, and return the solution with the lesser cost. We also had to compensate for when a solution is considered “no solution,” and if one is no solution then return the other, and this works since if both are not a solution, then there must not be a solution.

6. Implementation Issues

There weren't many issues during the implementation of the Quince-McClusky algorithm besides some syntax confusion during the writing of the processGroups() method. Most of the issues came from writing our branch and bound algorithm, as we found it a bit difficult to understand and translate the pseudo code at first. This required us to do some additional reading in the text to gain a full understanding. It also took us some time to understand why the recursion represented the branching in the binary decision diagram. There was also confusion with what the cost calculation should return, which we looked into while debugging.

During some of our debugging, we learned through experience that the branching tries all ‘1’ decisions as possible, and once it reaches the end (or terminal case), it will begin to traverse back up the branch, taking the solution with it. With each traversal back upward into a parent node, the program will venture down the ‘0’ zero decision from that node, generating the branches that should be attached to it, and this repeats until the tree is finished. The figure 16 was generated with a python script generated by chatGPT to aid in the visualization of the order these decision and solutions are found. The subsequent code can also be found in the following figure, figure 17:

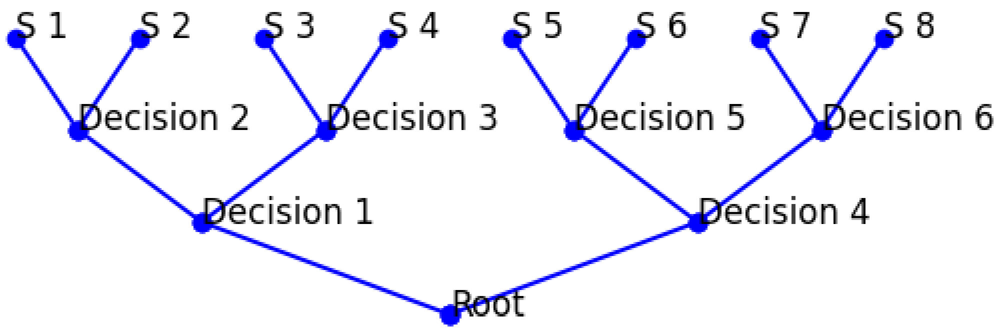


Figure 16. Diagram showing order of decision made and solutions found. In our case, a left branch symbolizes a 1 decision and a right branch symbolizes the 0 decision

```python
import matplotlib.pyplot as plt

class Node:
    def __init__(self, val):
        self.val = val
        self.left = None
        self.right = None

def visualize_tree(node, depth=1, x=0, y=0, prev_x=0, prev_y=0):
    if node is not None:
        plt.plot([prev_x, x], [prev_y, y], marker='o', color='b')
        plt.text(x, y, node.val, fontsize=12)
        if node.left is not None:
            visualize_tree(node.left, depth+1, x-2**(5-depth), y-2, x, y)
        if node.right is not None:
            visualize_tree(node.right, depth+1, x+2**(5-depth), y-2, x, y)

def main():
    global root
    root = Node('Root')
    root.left = Node('Decision 1')
    root.right = Node('Decision 4')
    root.left.left = Node('Decision 2')
    root.left.right = Node('Decision 3')
    root.right.left = Node('Decision 5')
    root.right.right = Node('Decision 6')
    root.left.left.left = Node('S 1')
    root.left.left.right = Node('S 2')
    root.left.right.left = Node('S 3')
    root.left.right.right = Node('S 4')
    root.right.left.left = Node('S 5')
    root.right.left.right = Node('S 6')
    root.right.right.left = Node('S 7')
    root.right.right.right = Node('S 8')

    plt.figure(figsize=(10, 10))
    visualize_tree(root)
    plt.xlim(-32, 32)
    plt.ylim(-8, 8)
    plt.gca().invert_yaxis()
    plt.axis('off')
    plt.show()
```

```
if __name__ == "__main__":
    main()
```

Figure 17. Python code for generation of diagram.

When beginning to write our code, we made the cost as the number of variables present in the current solution. The issue with this however is that it would often cause results that did not include all essential prime implicants due to the structure of the code. The way in which we were calculating cost as the number of variables would be to iterate through each implicant in the current solution, and keep a count of how many indices had a 1 or 0, showing that variable was present in the prime implicant. Our cost function, with the old methodology commented out, is shown below in figure 18:

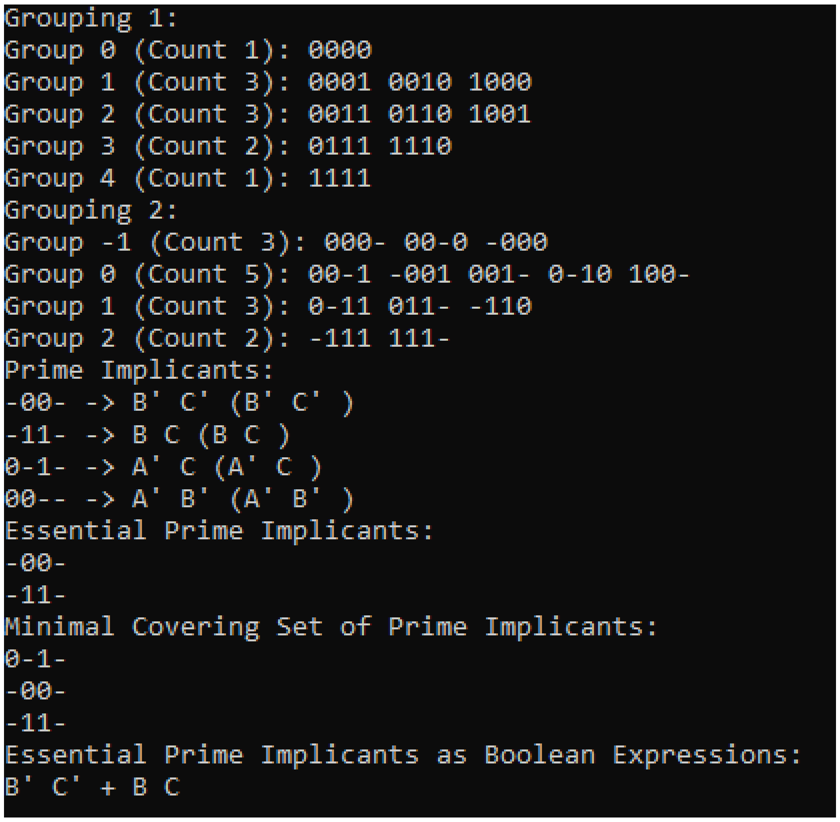
```
int Cost(vector<string> currentSolution) {
    if (currentSolution.empty()) return 0;
    else {
        /*set<int> indicies;
        for (string s : currentSolution) {
            for (int i = 0; i < s.size(); i++) {
                if (s.at(i) == '1' || s.at(i) == '0') indicies.insert(i);
            }
        }
        return indicies.size();
        */
        return currentSolution.size();
    }

}
```

Figure 18. Used Cost() function, with the old method commented out.

The issue in doing it this way is that if we need three essential prime implicants to be included, and a current solution has only two of thos prime implicants, it may have a cost of three, which is equal to the lower bound and is considered a perfect solution. An example to show this is if we have two prime implicants '-10-' and '--10', the old Cost function would calculate this as a cost of 3 although it may be already known there are supposed to be three essential prime implicants. This incited the change to the cost just representing the size of the current solution list.

7. Experimental Results

To show what our raw output data looks like, figure 19 provides a screenshot of the console. The example in this screenshot was not used in This output shows first all the minterms entered in their binary form, then their initial groupings based on the amount of 1s in their binary form. After this the processGroups() function is called, and each iteration prints what the groups look like at this step. After this the prime implicants are printed under "Prime Implicants:." Directly after that the list of essential prime implicants are printed. At this point the branch and bound algorithm is run, and its output is printed under "Minimal Covering Set of Prime Implicants:."



```
Grouping 1:
Group 0 (Count 1): 0000
Group 1 (Count 3): 0001 0010 1000
Group 2 (Count 3): 0011 0110 1001
Group 3 (Count 2): 0111 1110
Group 4 (Count 1): 1111
Grouping 2:
Group -1 (Count 3): 000- 00-0 -000
Group 0 (Count 5): 00-1 -001 001- 0-10 100-
Group 1 (Count 3): 0-11 011- -110
Group 2 (Count 2): -111 111-
Prime Implicants:
-00- -> B' C' (B' C' )
-11- -> B C (B C )
0-1- -> A' C (A' C )
00-- -> A' B' (A' B' )
Essential Prime Implicants:
-00-
-11-
Minimal Covering Set of Prime Implicants:
0-1-
-00-
-11-
Essential Prime Implicants as Boolean Expressions:
B' C' + B C
```

Figure 19. Example console output.

To provide some validation that the group process works correctly, we will go through what is shown in the screenshot, also because there are only two stages to the grouping. This is actually an example that was covered in class, and as we can see both essential prime implicants are included, and in a true answer either one of the other remaining prime implicants could have been selected to provide a maximum covering.

Minterms	Prime Implicants	Minimal Covering Set of Prime Implicants:	Essential Prime Implicants as Boolean Expressions:
0 3 5 6 7 9 10 11 12 13 14 15 18 20 24	0--11 -> A' D E (A' D E) 0-1-1 -> A' C E (A' C E) 0-11- -> A' C D (A' C D) 01--1 -> A' B E (A' B E) 01-1- -> A' B D (A' B D) 011-- -> A' B C (A' B C)	0--11 0-1-1 0-11- 01--1 01-1- 011--	A' D E + A' C E + A' C D + A' B E + A' B D + A' B C
0 3 10 13 15 24 27 30 31 33 42 45 48 51 63	-01010 -> B' C D' E F' (B' C D' E F') -01101 -> B' C D E' F (B' C D E' F) -11111 -> B C D E F (B C D E F) 0-1111 -> A' C D E F (A' C D E F) 0011-1 -> A' B' C D F (A' B' C D F) 011-11 -> A' B C E F (A' B C E F) 01111- -> A' B C D E (A' B C D E)	0-1111 011-11 01111- -01010 -01101 -11111	B' C D' E F' + B' C D E' F + B C D E F + A' B C E F + A' B C D E
0 7 20 27 31 32 42 47 52 55 63 79 85 94 106	-101010 -> B C' D E' F G' (B C' D E' F G') 0-00000 -> A' C' D' E' F' G' (A' C' D' E' F' G') 0-10100 -> A' C D' E F' G' (A' C D' E F' G') 0-11111 -> A' C D E F G (A' C D E F G) 0011-11 -> A' B' C D F G (A' B' C D F G) 01-1111 -> A' B D E F G (A' B D E F G) 011-111 -> A' B C E F G (A' B C E F G)	0011-11 0-00000 01-1111 0-10100 011-111 -101010	B C' D E' F G' + A' C' D' E' F' G' + A' C D' E F' G' + A' B' C D F G + A' B D E F G + A' B C E F G
0 15 42 53 63	0--101010 -> A' D E' F G' H	0--101010	A' D E' F G' H I' + A' D E

85 106 127 170 191 213 234 255 277 319	I' (A' D E' F G' H I') 0--111111 -> A' D E F G H I (A' D E F G H I)	0--111111	F G H I
0 31 85 106 127 213 234 255 341 362 426 447 511 597 638	-001010101 -> B' C' D E' F G' H I' J (B' C' D E' F G' H I' J) 0-01010101 -> A' C' D E' F G' H I' J (A' C' D E' F G' H I' J) 0-01101010 -> A' C' D E F' G H' I J' (A' C' D E F' G H' I J') 0-11111111 -> A' C D E F G H I J (A' C D E F G H I J) 00-1010101 -> A' B' D E' F G' H I' J (A' B' D E' F G' H I' J) 00-1101010 -> A' B' D E F' G H' I J' (A' B' D E F' G H' I J') 00-1111111 -> A' B' D E F G H I J (A' B' D E F G H I J) 011-111111 -> A' B C E F G H I J (A' B C E F G H I J)	00-1111111 00-1010101 00-1101010 0-01010101 0-01101010 011-111111 -001010101	B' C' D E' F G' H I' J + A' C' D E' F G' H I' J + A' C' D E F' G H' I J' + A' B' D E' F G' H I' J + A' B' D E F' G H' I J' + A' B' D E F G H I J + A' B C E F G H I J
0 63 170 213 255 341 426 511 682 767 853 891 934 1021 1022	-010101010 -> B' C D' E F' G H' I J' (B' C D' E F' G H' I J') -011111111 -> B' C D E F G H I J (B' C D E F G H I J) -101010101 -> B C' D E' F G' H I' J (B C' D E' F G' H I' J) 0-10101010 -> A' C D' E F' G H' I J' (A' C D' E F' G H' I J') 0-11111111 -> A' C D E F G H I J (A' C D E F G H I J)	0-10101010 0-11111111 -010101010 -011111111 -101010101	B' C D' E F' G H' I J' + B' C D E F G H I J + B C' D E' F G' H I' J + A' C D' E F' G H' I J' + A' C D E F G H I J
0 2 512 514 768 770 896 898 960 962	-0000000-0 -> B' C' D' E' F' G' H' J' (B' C' D' E' F' G' H' J')	-0000000-0 11-00000-0 11111-00-0	B' C' D' E' F' G' H' J' + A B C D E F H' J' + A B C D E F G H I

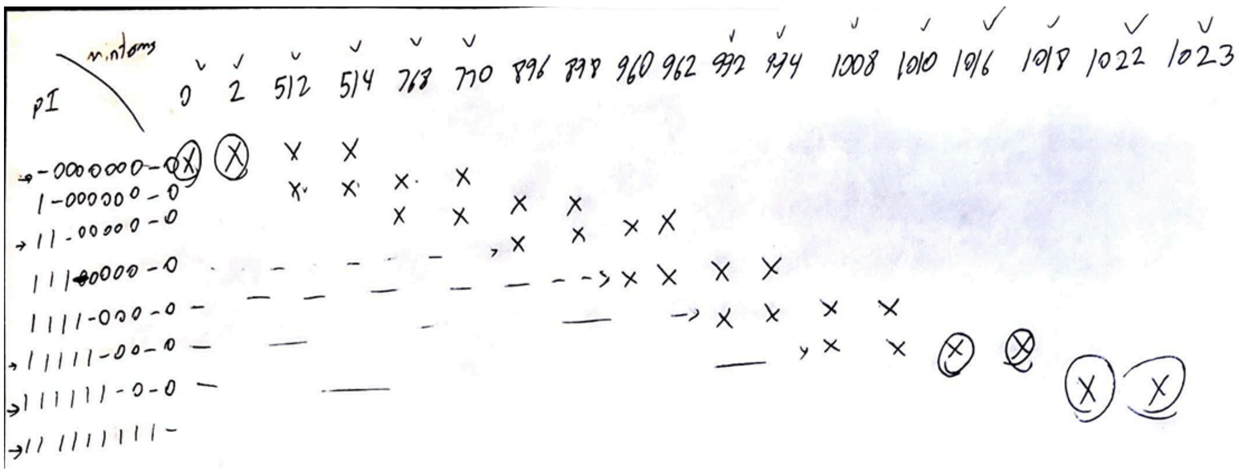
992 994 1008 1010 1016 1018 1022 1023	1-000000-0 -> A C' D' E' F' G' H' J' (A C' D' E' F' G' H' J') 11-00000-0 -> A B D' E' F' G' H' J' (A B D' E' F' G' H' J') 111-0000-0 -> A B C E' F' G' H' J' (A B C E' F' G' H' J') 1111-000-0 -> A B C D F' G' H' J' (A B C D F' G' H' J') 11111-00-0 -> A B C D E G' H' J' (A B C D E G' H' J') 111111-0-0 -> A B C D E F H' J' (A B C D E F H' J')	111111-0-0 111111111-	
0 255 320 380 420 477 480 560 620 690 735 850 902 952 1024	-0000000000 -> B' C' D' E' F' G' H' I' J' K' (B' C' D' E' F' G' H' I' J' K')	-0000000000	B' C' D' E' F' G' H' I' J' K'

Table 1. Results for several sets of minterms

Based on table 1 we can see various results for various sets of minterms. The requirement from the project description is to test 10 different equations with up to 10 variables. This would mean that the minterms must be expressed with a ten digit binary number, meaning minterm decimal values up to 1023, since after this point branch and bound is known to begin to make mistakes.

We also must analyze the results from the table in order to confirm whether our program is working properly. There are several examples where all found prime implicants are essential, then they're all included in the minimal covering set, so let us choose a row where not all prime implicants are essential.

Some of these input equations were generated with chatGPT using the prompt: "give me sets of minterms representing up to 10 variable logic equations," the 9th row represents the set of minterms that represent a 10 variable equation. This example also shows that only two of the prime implicants are found to be essential, making this a worthy challenge to find the minimal covering set. In order to truly check the work, I will do the operations by hand in order to confirm our algorithm properly works.The matching is straightforward enough that we proved it was correct with the screenshot provided in figure 19, and I will move forward to the prime implicants chart, and see if the found essential prime implicants are correct, and that the minimal covering is correct.



PI \ mintoms	0 ✓	2 ✓	512 ✓	514 ✓	768 ✓	770 ✓	896	898	960	962	992 ✓	994 ✓	1008 ✓	1010 ✓	1016 ✓	1018 ✓	1022 ✓	1023 ✓
→ -0000000-0	⊗	⊗	X	X														
1-000000-0			X	X	X	X												
→ 11-00000-0					X	X	X	X										
111-0000-0							X	X	X	X								
1111-000-0									X	X	X	X						
→ 11111-00-0											X	X	X	X				
→ 111111-0-0													X	X	⊗	⊗		
→ 111111111-																	⊗	⊗

Figure 19. Finding essential prime implicants by hand.

There is an issue with this example however that the minimal covering set does not cover all minterms. I take it this comes with the fact that larger amounts of variables do cause branch and bound to make errors as discussed in class. If we look in the example used in the screenshot in figure 18, however, we can see that the minimal covering set easily covers the entire set of minterms.

8. Conclusion

After writing, debugging, and obtaining results from our implementation of branch and bound and Quince-McClusky, they seem like almost opposite algorithms in that Quince-McClusky is a rigid algorithm, in that all input has a definitive output, but branch and bound is not always able to find that output, and doesn't even seem like it tries to. Branch and bound introduces this idea of a lower bound, and then exhaustively testing all possible solution attempts to the minimization, while pruning solutions that are clearly infeasible, having a cost less than the lower bound. What perplexes me is that the algorithm never demands proof, at least in the Hachtel and Somenzi [1] pseudo code, that the solution is a correct one. It is possible that I may be overconfident in my understanding of the proposed program, however there was no mentioned function that would actually validate that the solution is a correct one. The authors speak about how branch and bound can search the solution space exhaustively and find the absolute optimal solution, but I believe that the objective function that the algorithm is trying to solve is not subject to a hard rule that the solution must cover all minterms.

A popular theme we've seen in reading about all the algorithms presented during lecture, are talks about the idea of a solution space, and the goal of the algorithm is to search and find an 'optimal solution' within the space, and it's difficult not to think there is a way to maybe explore this solution space to get a general idea of what it may look like, and with this general knowledge of the space maybe find a function that can transform our input into the true optimal solution in this space. Branch and Bound has gotten the closest to this in that it will explore the entire solution space, and once it's done it'll climb back out with what should be the best solution. These things are easier said than done, however I just find it frustrating that there is still a struggle to find a definitive right function, rather than a 'best' function.

9. Bibliography

[1] Hachtel, Gary D., and Fabio Somenzi. *Logic Synthesis and Verification Algorithms*. Springer, 2006.

[2] T. K. Jain, D. S. Kushwaha and A. K. Misra, "Optimization of the Quine-McCluskey Method for the Minimization of the Boolean Expressions," Fourth International Conference on Autonomic and Autonomous Systems (ICAS'08), Gosier, France, 2008, pp. 165-168, doi: 10.1109/ICAS.2008.11. keywords: {Optimization methods;Minimization methods;Digital circuits;Pattern recognition;Design optimization;DH-HEMTs;Hardware;Cost function;Boolean functions;Boolean Expression;Prime Implicants;Sum of Products;Product of Sum},

[3] Jadhav, Vitthal, and Amar Buchade. "Modified Quine-Mccluskey Method." *arXiv.Org*, 8 Mar. 2012, arxiv.org/abs/1203.2289.

[4] Duşa, Adrian. "Enhancing Quine-McCluskey -." *University of Bucharest*, 2007, compasss.org/wpseries/Dusa2007b.pdf.

[5] J. Safaei and H. Beigy, "Quine-McCluskey Classification," 2007 IEEE/ACS International Conference on Computer Systems and Applications, Amman, Jordan, 2007, pp. 404-411, doi: 10.1109/AICCSA.2007.370913. keywords: {Minimization methods;Boolean functions;Circuits;Java;Training data;Paper technology;Testing;Databases;Zero current switching;Polynomials},

[6] Tomaszewski, Sebastian P, et al. "WWW-Based Boolean Function Minimization." *Montclair State University*, 2003, www.montclair.edu/profilepages/media/318/user/10.1.1.119.3383.pdf.

[7] Boyd , Stephen, and Jacob Mattingley. "Branch and Bound Methods." *Stanford University*, 18 June 2010, citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=5c1f384f2478efb17a83c2f5ade6da059c7dbbea.

[8] He, He, et al. "Learning to Search in Branch and Bound Algorithms." *Advances in Neural Information Processing Systems*, 1 Jan. 1970, proceedings.neurips.cc/paper_files/paper/2014/hash/757f843a169cc678064d9530d12a1881-Abstract.html.

[9] Morrison, David R, et al. "Branch-and-Bound Algorithms: A Survey of Recent Advances in Searching, Branching, and Pruning." *Discrete Optimization*, Elsevier, 16 Feb. 2016, www.sciencedirect.com/science/article/pii/S1572528616000062.

10.Appendix

```
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>
#include <map>
#include <set>
#include <sstream>
#include <cmath>
#include <bitset>
using namespace std;

bool isCovered(const string& minterm, const string& implicant) {
    // Check each bit in the minterm against the implicant
    for (size_t i = 0; i < minterm.size(); ++i) {
        // If implicant bit is not '-' and does not match minterm bit,
return false
        if (implicant[i] != '-' && implicant[i] != minterm[i]) {
            return false;  // Implicant does not cover the minterm at this
bit
        }
    }
    return true;  // All bits match or are covered by don't care positions
in implicant
}

// Converts an integer to a binary string with a specified length
string toBinary(int number, int digits) {
    string binary = bitset<32>(number).to_string();
    return binary.substr(32 - digits);  // adjust length as necessary
}

vector<string> findEPI(const map<string, vector<string>>& chart) {
    vector<string> essentialPrimeImplicants;
    // Iterate over each minterm's list of implicants
    for (const auto& minterm : chart) {
        if (minterm.second.size() == 1) {  // If exactly one implicant
covers this minterm
            string soleImplicant = minterm.second.front();
            if (find(essentialPrimeImplicants.begin(),
essentialPrimeImplicants.end(), soleImplicant) ==
essentialPrimeImplicants.end()) {
                essentialPrimeImplicants.push_back(soleImplicant);
            }
        }
    }
    return essentialPrimeImplicants;
}


vector<string> findVariables(const string& minterm) {
    vector<string> varList;
    for (size_t i = 0; i < minterm.length(); ++i) {
        if (minterm[i] == '0') {
            varList.push_back(string(1, static_cast<char>(i + 'A')) + "'");
        }
        else if (minterm[i] == '1') {
            varList.push_back(string(1, static_cast<char>(i + 'A')));
        }
    }
    return varList;
}

// Function to group minterms by count of 1's in their binary
representation
map<int, vector<string>> groupMinterms(const vector<string>&
binaryMinterms) {
    map<int, vector<string>> groups;

    for (string minterm : binaryMinterms) {
        int countOnes = count(minterm.begin(), minterm.end(), '1');
        groups[countOnes].push_back(minterm);
    }

    // Debug: print groups for verification
    cout << "Grouping based on count of '1's in binary representation:\n";
    for (const auto& group : groups) {
        cout << "Group " << group.first << " (" << group.second.size() << "
elements): ";
```

```
        for (const auto& bin : group.second) {
            cout << bin << " ";
        }
        cout << endl;
    }

    return groups;
}

// Combines two minterms if they differ by exactly one bit, using a dash
'-' to mark the differing bit.
pair<bool, string> tryCombine(const string& a, const string& b) {
    if (a.length() != b.length()) {
        return { false, "" };  // Ensure both strings are of equal length
    }

    int countDifferences = 0;
    string result = a;

    for (size_t i = 0; i < a.length(); ++i) {
        if (a[i] != b[i]) {
            countDifferences++;
            result[i] = '-';
            if (countDifferences > 1) {
                return { false, "" };  // Not combinable if more than one
difference
            }
        }
    }

    return countDifferences == 1 ? make_pair(true, result) :
make_pair(false, "");
}


// Function to process groups to find prime implicants
set<string> processGroups(map<int, vector<string>>& groups) {
    set<string> primeImplicants;
    map<int, vector<string>> newGroups;
    bool progress = false;
    int count = 0;
    while (true) {
        progress = false;
        newGroups.clear();

        for (auto it = groups.begin(); it != groups.end(); ++it) {
            auto next = std::next(it);
            if (next == groups.end()) break;

            for (const auto& current : it->second) {
                for (const auto& nextMinterm : next->second) {
                    auto combineResult = tryCombine(current, nextMinterm);
                    if (combineResult.first) {
                        // Check if this new minterm is already added to
avoid duplicates
                        if (!newGroups[it->first - 1].empty() &&
                            find(newGroups[it->first - 1].begin(),
newGroups[it->first - 1].end(), combineResult.second) ==
newGroups[it->first - 1].end()) {
                            newGroups[it->first -
1].push_back(combineResult.second);
                        }
                        else if (newGroups[it->first - 1].empty()) {
                            newGroups[it->first -
1].push_back(combineResult.second);
                        }
                        progress = true;
                    }
                }
            }
        }

        // If no progress can be made, break and add remaining minterms as
prime implicants
        if (!progress) {
            for (const auto& group : groups) {
                for (const auto& minterm : group.second) {
                    // Ensure no duplicates
                    if (primeImplicants.find(minterm) ==
primeImplicants.end()) {
                        primeImplicants.insert(minterm);
                    }
                }
            }
            break;
        }
        // Set up groups for the next iteration
        count++;
        cout << "Grouping " << count << ": " << endl;
        for (const auto& group : groups) {
            cout << "Group " << group.first << " (Count " <<
group.second.size() << "): ";
            for (const auto& bin : group.second) {
                cout << bin << " ";
            }
            cout << "\n";
        }
        groups = newGroups;
    }

    return primeImplicants;
}

string binaryToExpression(const string& binary) {
    string result;
    bool firstTerm = true;  // Flag to manage spacing and formatting

    for (size_t i = 0; i < binary.size(); ++i) {
        if (binary[i] == '1' || binary[i] == '0') {
            if (!firstTerm) {
                result += " ";  // Add space before each term except the
first
            }
            result += char('A' + i);  // Append the variable
            if (binary[i] == '0') {
                result += "'";  // Append ' for negated terms
            }
            firstTerm = false;  // Update flag after adding the first term
        }
    }

    return result;
}

vector<int> readNumbers(const string& prompt) {
    cout << prompt;
    string line;
    getline(cin, line);
    istringstream iss(line);
    set<int> numbers;
    int num;
    while (iss >> num) {
        if (iss.fail() || num < 0) {
            iss.clear(); // Clear error state
            cout << "Invalid input. Please enter only non-negative
integers.\n";
            getline(cin, line); // Re-prompt input
            iss.str(line);
            continue;
        }
        numbers.insert(num);
    }
    return vector<int>(numbers.begin(), numbers.end());
}

int findMaxBits(const vector<int>& numbers) {
    if (numbers.empty()) return 0;
    int maxNum = *max_element(numbers.begin(), numbers.end());
    return ceil(log2(maxNum + 1));
}

map<string, vector<string>> flipChart(map<string, vector<string>> original)
{
    map<string, vector<string>> fchart;
    set<string> implicantsToAdd;
    for (auto pair : original) {
        for (string imp : pair.second) {
            implicantsToAdd.insert(imp);
        }
    }
    for (string imp : implicantsToAdd) {
        for (auto pair : original) {
            if (find(pair.second.begin(), pair.second.end(), imp) !=
pair.second.end()) fchart[imp].push_back(pair.first);
        }
    }
    return fchart;
}

map<string, vector<string>> Reduce(const map<string, vector<string>>&
chart) {
    map<string, vector<string>> reducedChart = chart;

    //Col dominance: (if a minterm has same implicants as another and more
it can be removed)
    set<string> keysToRemove;
    for (auto ci : reducedChart) {
        set<string> ciImplicants(ci.second.begin(), ci.second.end());
        for (auto cj : reducedChart) {
            if (ci.first != cj.first) {
                set<string> cjImplicants(cj.second.begin(),
cj.second.end());
                if (keysToRemove.find(cj.first) == keysToRemove.end() &&
!cj.second.empty() && includes(ciImplicants.begin(), ciImplicants.end(),
cjImplicants.begin(), cjImplicants.end())) { //if the implicants of cj are
a subset of ci:
                    keysToRemove.insert(ci.first);
                    break;
                }
            }
        }
    }
    for (string key : keysToRemove) {
        reducedChart.erase(key);
    }
    keysToRemove.clear();

    //Row dominance:
    //first create flipped chart:
    set<string> pitoremove;
    map<string, vector<string>> fchart = flipChart(reducedChart);
    for (auto ri : fchart) {
        set<string> riMinterms(ri.second.begin(), ri.second.end());
        for (auto rj : fchart) {
            if (ri.first != rj.first) {
                set<string> rjMinterms(rj.second.begin(), rj.second.end());
                if (keysToRemove.find(ri.first) == keysToRemove.end() &&
!ri.second.empty() && includes(riMinterms.begin(), riMinterms.end(),
rjMinterms.begin(), rjMinterms.end())) pitoremove.insert(rj.first); break;
            }
        }
    }
    for (string key : pitoremove) {
        for (auto &pair : reducedChart) {
            auto rem = find(pair.second.begin(), pair.second.end(), key);
            if (rem != pair.second.end()) {
                pair.second.erase(rem);
            }
        }
    }

    return reducedChart;

}

int MIS_quick(const map<string, vector<string>>& chart) {
    map<string, vector<string>> chartcopy = chart;
    set<string> keys;
    for (auto pair : chartcopy) {
        if (pair.second.empty()) keys.insert(pair.first);
    }
    for (string key : keys) {
        chartcopy.erase(key);
    }
    map<string, vector<string>> MIS;
    do {
        int minimplicants = INT_MAX;
        string key = "";
        for (const auto& pair : chartcopy) {
            if (pair.second.size() < minimplicants) {
                minimplicants = pair.second.size();
                key = pair.first;
            }
        }
        if(chartcopy.size() != 0) MIS[key] = chartcopy[key];
        vector<string> keysToRemove;
        for (string implicant : MIS[key]) {
            for (const auto& pair : chartcopy) {
                if (find(pair.second.begin(), pair.second.end(), implicant)
!= pair.second.end()) {//if a minterm is convered by implicant, remove
                    //chartcopy.erase(pair.first);
                    keysToRemove.push_back(pair.first);
                }
            }
            for (auto key : keysToRemove) {
                chartcopy.erase(key);
            }
        }
    } while (!chartcopy.empty());

    return MIS.size();

}

int Cost(vector<string> currentSolution) {
    if (currentSolution.empty()) return 0;
    else {
        /*set<int> indicies;
        for (string s : currentSolution) {
            for (int i = 0; i < s.size(); i++) {
                if (s.at(i) == '1' || s.at(i) == '0') indicies.insert(i);
            }
        }
        return indicies.size();
        */
        return currentSolution.size();
    }

}

vector<string> BestSolution(vector<string> solution1, vector<string>
solution2) {
    if (solution1.at(0) == "no solution") return solution2;
    else if (solution2.at(0) == "no solution") return solution1;
    else return Cost(solution1) >= Cost(solution2) ? solution1 : solution2;
}

bool isTerminal(map<string, vector<string>> chart) {
    bool flag = true;
    for (auto pair : chart) {
        if (!pair.second.empty()) flag = false;
    }
    return flag;
}

vector<string> BCP(map<string, vector<string>> &chart, int U,
vector<string> &currentSolution) {
    map<string, vector<string>> rchart = Reduce(chart);

    if (isTerminal(rchart)) { //if terminal case
```

```
        if (Cost(currentSolution) < U) {
            U = Cost(currentSolution);
            if (currentSolution.empty()) return { "no solution" };
            else return currentSolution;
        }
        else {
            return { "no solution." };
        }
    }
    int LB = MIS_quick(rchart);
    if (LB >= U) {
        return { "no solution" };
    }

    string pi;
    map<string, vector<string>> rchart1 = rchart;
    map<string, vector<string>> rchart0 = rchart;
    for (auto pair : rchart) {
        if (!pair.second.empty()) {
            pi = pair.second.at(0); //choose a pi
            break;
        }
    }

    //set pi to 1 and branch (any chart entry with pi becomes (1 + pj),
therfore remove these entries completely and add to currentSolution)
    set<string> keys;
    for (auto pair : rchart1) {
        for (string s : pair.second) {
            if (s == pi) keys.insert(pair.first); break;
        }
    }
    for (auto &pair : rchart1) {
        if (keys.find(pair.first) != keys.end()) {
            pair.second.clear();
        }
    }
    keys.clear();
    vector<string> cs1 = currentSolution;
    cs1.push_back(pi);
    auto s1 = BCP(rchart1, U, cs1);
    if (Cost(s1) == LB) return s1;
    //set pi to 0 and branch (remove pi (0 + pj) and DONT add to current
solution)
    for (auto &pair : rchart0) {
        pair.second.erase(remove_if(pair.second.begin(), pair.second.end(),
[&pi](string s) {return s == pi; }), pair.second.end()); //remove pi
    }
    auto s0 = BCP(rchart0, U, currentSolution);
    return BestSolution(s0, s1);
}

int main() {
    vector<int> minterms = readNumbers("Enter the minterms
(space-separated): ");
    //vector<int> dontCares = readNumbers("Enter the don't cares
(space-separated, if any): ");
    vector<int> dontCares;
    // Combine minterms and don't cares into allMinterms
    vector<int> allMinterms(minterms);
    allMinterms.insert(allMinterms.end(), dontCares.begin(),
dontCares.end());

    // Determine the maximum number of bits for binary representation
    int maxBits = findMaxBits(allMinterms);
    vector<string> binaryMinterms;

    // Output binary representations of all minterms
    cout << "All minterms and their binary representations:\n";
    for (int minterm : allMinterms) {
        string binary = toBinary(minterm, maxBits);
        binaryMinterms.push_back(binary);
        cout << minterm << " -> " << binary << endl;
    }

    // Group minterms
    auto groups = groupMinterms(binaryMinterms);

    // Identify prime implicants
    auto primeImplicants = processGroups(groups);
    map<string, vector<string>> varImplicants;
    cout << "Prime Implicants:\n";
    for (const auto& implicant : primeImplicants) {
        vector<string> vars = findVariables(implicant);
        varImplicants[implicant] = vars; // Map binary implicant to its
variable representation
```

```
        cout << implicant << " -> " << binaryToExpression(implicant) << " 
(";
        for (const auto& var : vars) {
            cout << var << " ";
        }
        cout << ")\n";
    }

    // Create a prime implicant chart
    map<string, vector<string>> chart;
    for (int minterm : allMinterms) {
        string binaryMinterm = toBinary(minterm, maxBits);
        for (const auto& implicant : primeImplicants) {
            if (isCovered(binaryMinterm, implicant)) {
                chart[binaryMinterm].push_back(implicant); //chart consists 
of each prime implicant with it's minterms(binary)
            }
        }
    }

    // Extract essential prime implicants
    vector<string> epiVector = findEPI(chart);
    set<string> essentialPrimeImplicants(epiVector.begin(), 
epiVector.end());

    set<string> coveredMinterms; // To track minterms covered by EPIs

    // Initialize allMinterms set for Branch-and-Bound
    set<string> allMintermBins; // Collecting all minterms that need to be 
covered
    for (const auto& pair : chart) {
        allMintermBins.insert(pair.first);
    }

    // Initialize an upper bound U for Branch-and-Bound
    int U = INT_MAX; // Start with the maximum possible cost

    // Call Branch-and-Bound
    vector<string> currentSolution;
    vector<string> best;

    best = BCP(chart, U, currentSolution);
    // Display results
    cout << "Essential Prime Implicants:\n";
    for (const auto& epi : essentialPrimeImplicants) {
        cout << epi << endl;
    }

    cout << "Minimal Covering Set of Prime Implicants:\n";
    for (const auto& implicant : best) {
        cout << implicant << endl;
    }

    // Display results in Boolean expression format
    cout << "Essential Prime Implicants as Boolean Expressions:\n";
    string booleanExpression;
    for (const auto& epi : essentialPrimeImplicants) {
        if (!booleanExpression.empty()) booleanExpression += " + ";
        booleanExpression += binaryToExpression(epi);
    }
    cout << booleanExpression << endl;

    return 0;
}
```