Indian Institute of Technology Delhi

COL 215 - Digital Logic and System Design

Assignment 3

Removing the redundant prime implicants

Aryan Sharma, Garv Nagori
2021CS10553, 2021CS10549

Contents

1 Introduction

In this assignment, we build upon our previous assignment by minimising the number of terms of the boolean expression. In our last assignment we maximized the region occupied by each minterm. In this assignment, we remove those terms that can be covered by some combination of other terms. Those prime implicants which are necessary to cover some minterm which cannot be covered by any other term are called *essential prime implicants*. Those terms which can be covered by some combination of other terms are called *redundant prime implicants*. So we can divide our algorithm into two parts:

1. Taking all the *essential prime implicants*
2. Removing the *redundant prime implicants*

2 Algorithm

In this final function we call *comb_function_expansion* function with the inputs *func_True* and *func_DC*. The return value are the *prime implicants*. We have created a list *dcbin* which contains the binary representation of the Don't Care terms.

We now create a dictionary *dict1* with keys as the *minterms* which are not in *func_DC* and values as the list of all *prime implicants* that minterm is a part of. If the length of any list is 1, it is an *essential prime implicant* so we take it and delete all the minterms covered by that. We repeat the process until all such *essential prime implicants* have been covered.

If the dictionary becomes empty, it means that all the *essential prime implicants* have covered all the legal regions. In this case we simply use the function *fromBinary* to return the final answer.

We now apply ***Petrick's method*** to minimise the *non-essential prime implicants* if they are remaining. For each remaining minterm, we write it as a OR of the *prime implicants* which it is a part of. We then multiply all such terms to create kind of a POS form since we want each OR expression to be true. Expanding all the terms out we create a SOP form and then choose the AND expression with the least number of terms.

To implement this in Python, we again take the help of ***Bitmasking***. We can represent each *prime implicant* as a power of 2. Thus each AND expression can be represented as a binary number. The positions of "1" gives us which *prime implicants* to take. We create yet another dictionary *dict2* with keys as powers of 2 and values as prime implicants. We also create a list *petrick*, containing lists with prime implicants corresponding to a minterm.

Now we iterate over petrick and combine all the groups one by one. Here, we use the cleverness of using *bits*. To combine two OR expressions we simply take the OR of the terms which are combining.

After the last iteration we have somewhat of a SOP form with each product as a number and all the numbers in *petrick[len(petrick)-1]*. We find the number with the *minimum* number of 1's in it, since that will have the minimum number of *prime implicants* in it. Combining this with the list of *essential prime implicants* and then using *fromBinary* to convert to string, we finally get the ***required list***.

3 Time Complexity

m: Number of Variables in the K-Map
n: Combined length of the lists *func_TRUE* and *func_DC*

3.1 *toBinary*

It is a simple function which iterates over the minterm represented as string only once and returns its binary representation. The time complexity of this function is *O(n)*.

3.2 *fromBinary*

This function takes two terms. *nbin* represents the number corresponding to the term in binary. *dbin* represents the position of the "dashes" or the NONE terms. We convert them to strings and iterate over them to form a string corresponding to the term. Reversing the list and then converting to a string is linear. Thus the time complexity of this function is *O(n)*.

3.3 *combine*

This is the main function which combines the minterms to reduce them to prime implicants.

Here, we combine the minterms which have been separated into groups based on the number of "ON" bits. Thus the maximum number of terms in each groups is n in the worst case.

To check every term in one group with the next group, we need n^2 comparisons at maximum. Since, we have cleverly used bit manipulation, each comparison can be processed in *O(n)* time complexity to take union of two sets. There are m pairs of groups to be compared, therefore the time complexity of one iteration is *O(n^3m)*.

Each iteration combines terms and drops one literal from the combining terms and we recursively call this function. Maximum literals that can be dropped is m and due to this the overall time complexity of this function is *O(n^3m^2)*

However, this is a very crude analysis of this function, missing many of the finer details. Each group will not have n terms in the worst case and if there are there will be no other terms to combine with. There are two opposing factors at play here, the number of terms in one group and the number of comparisons. If one factor is increasing the other factor decreases.

If we compare each one term in the list with any other, we would still need n^2 comparisons. The upper bound of n^2 comparisons in our algorithm hides the cleverness of separating the minterms into groups. The optimizations done in this function make it much faster than the analysed time complexity. However, an amortized analysis is very difficult and the upper bound of *O(n^3m^2)* is correct.

3.4 *comb_function_expansion*

This is the precursor function to finding prime implicants. We calculate m first. Then we iterate over the lists and form the groups. After this we call the function *combine* to minimise the groups. This function's time complexity is *O(n^3m^2)* so this function call is the main bottleneck of this function.Sorting takes *O(n log(n))* time.After this we iterate through the output of combine function and remove the regions which are repeated and also the ones which are totally due to regions consisting only of X's,this takes *O(n)* time. Thus the time complexity of this function is in the worst case $O(n^3m^2) + O(n\ log(n)) + O(n)$. This is $O(n^3m^2)$

3.5 ***opt_function_reduce***

As explained in our algorithm, "Petrick's method" is extremely inefficient. It can blow up exponentially in the number of columns. Since the number of columns can be at max n, the time complexity is $O(n^n)$. This is the biggest bottleneck in our code which causes the time complexity of the entire program to be $O(n^n)$. However, in reality, the number of columns can never be n and if the number of columns increase so does the number of *essential prime implicants* which in turn decrease the number of columns left for Petrick's method.

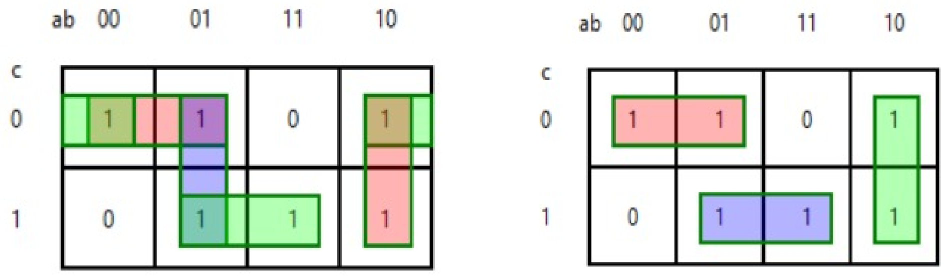
4 Test Cases

We ran our code against many test cases to validate its correctness.

1. func_true = ["ab'c'de","ab'cde","abcde"]
   func_DC = ["a'bcd'e'","a'bcde'","a'bcd'e","a'bcde","abc'de"]
   Output(Petrick's Method) = ["ade"]
   Output(Efficient but Sub-Optimal Algorithm) = ["ade","a'bc"]

2. func_True = ["ab'c'de'fgh'i'j'", "ab'c'de'fgh'ij", "ab'c'de'fgh'ij'", "ab'c'de'fghij", "ab'c'de'fghi'j", "a'b'c'def'gh'i'j'", "a'b'cdef'gh'i'j'", "a'bcd'ef'g'hij", "abc'd'e'fgh'ij", "abc'de'fgh'ij'"]
   func_DC = ["ab'c'de'fgh'i'j", "ab'c'de'fghij'", "ab'c'de'fghi'j'", "abc'de'fgh'ij", "abc'd'e'fgh'ij'"]
   Output(Petrick's Method) = ["ab'c'de'fg", "abc'e'fgh'i", "a'b'def'gh'i'j'","a'bcd'ef'g'hij"]
   Output(Efficient but Sub-Optimal Algorithm) = ["a'b'def'gh'i'j'", "a'bcd'ef'g'hij", "abc'e'fgh'i", "ab'c'de'fg"]

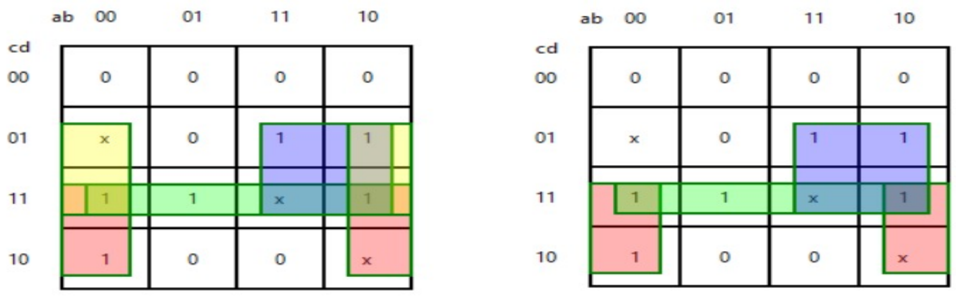
For the two assignments we made, one was efficient but sub-optimal and this one is inefficient but always gives the optimal results.
The efficient one included extra terms which were not necessary to cover the K Map.



(a) Efficient Algorithm but Sub-Optimal (b) Petrick's Method and Optimal

Figure 1: Difference between the two Algorithms

As you can see, the efficient one has 2 extra regions which can further be reduced. Petrick's method gives the best possible output but it is slow for very large input files. Another example of the difference:



(a) Efficient Algorithm but Sub-Optimal (b) Petrick's Method and Optimal

Figure 2: What the groups mean

Q. Why is this set good enough to validate the implementation?
A. The generic test cases we ran were not good enough to conclude that our answers were optimal. The few test cases which have the GUI (and many more like that) were the deciding factor. These test cases have some non-essential prime implicants. Therefore, the first code we created was not correct and we had to look for another way. Then we decided to implement "Petrick's method". This set of test cases containing non essential prime implicants is what determined our final algorithm and made us think outside the box.

5 Conclusion

In this assignment, we have actually made two programs to solve the problem, one of them is efficient but gives sub-optimal over some inputs answers while the other gives the optimal answer every time but takes exponential time. The assignment and piazza discussion had not clarified which one to submit thus we have made both of them, and commented one code in our submission

Github Link for our other Assignment