**COL215 Assignment 3**

By Adithya Bijoy and Anish Banerjee

# Our Algorithm:

We initialized a list (l) to which we will add the final terms of the expansion.

## Step1:

We start with the list (s) of minterms in the union of func\_TRUE and func\_DC. Using the MarkRegion() function, we pair up the possible terms in s and add the pairs into a list (m).

To make pairs, we create a set with all the elements in the list. A set is implemented using hash tables. So, we can add, remove or search for elements in constant time.

We traversed through the list s and checked if its neighbours are in the set. If the neighbours are in the list, we make a pair and add them to the list m. If none of the element’s neighbours are in the set, we add the element to the list l. We set s=m and repeat step 2 until we cover all possible expansions. In this way, our list l contains all the valid maximally expanded regions.

## Step2:

We remove the redundant terms in list l. For this, we first count the number of times each minterm appears in all the terms of list l combined. Then we traverse through the list l and remove the terms in which all the minterms involved have a count greater than 1 and reduce the count of those minterms by 1. The list l after removing all the redundancies will be the list required.

# Complexity analysis:

Complexity of our algorithm:

(Where n is the number of minterms in the union of func\_TRUE and func\_DC and N is the total number of variables)

**Analysis:**

**Step1:** Let mi ­be the number of terms in the list *Array* at the starting of ithrecursion. Each recursion takes *O(m2N2)*time (as the time taken to find the neighbours of a term is *O(N2)*); so, the time taken by the MarkRegion() function is:

Now, , that is, the number of possible regions of size i multiplied by the probability of the region being valid. So, the sum becomes:

Thus, complexity of this step is

**Step2:** Here the size of our list is and we traverse through all the minterms contained in each term of the list. Hence complexity becomes , which is less than that of the second step. Hence the complexity of step 1 dominates.

## Test Cases

|  |  |  |
| --- | --- | --- |
| S.no. | Test Case | Output |
| 1 | func\_TRUE = ["a'bc'd'", "abc'd'", "a'b'c'd", "a'bc'd", "a'b'cd"]  func\_DC = ["abc'd"] | ["a'b'd", "bc'"] |
| 2 | func\_TRUE = ["a'b'c'd", "a'b'cd", "a'bc'd", "abc'd", "abc'd'", "ab'c'd'", "ab'cd"]  func\_DC = ["a'bc'd'", "a'bcd", "ab'c'd"] | ["b'd", "bc'", "ac'"] |
| 3 | func\_TRUE = ["a'b'c", "a'bc", "a'bc'", "ab'c'"]  func\_DC = ["abc'"] | ["a'c", "ac'", "a'b"] |
| 4 | func\_TRUE = ["a'b'c'd'e'", "a'bc'd'e'", "abc'd'e'", "ab'c'd'e'", "abc'de'", "abcde'",  "a'bcde'", "a'bcd'e'", "abcd'e'", "a'bc'de", "abc'de", "abcde",  "a'bcde", "a'bcd'e", "abcd'e", "a'b'cd'e", "ab'cd'e"]  func\_DC = [] | ['bde', "c'd'e'", 'abd', "cd'e", 'bc'] |
| 5 | func\_TRUE = ["ab", "ab'", "a'b","a'b'"]  func\_DC = [] | [None] |
| 6 | func\_TRUE = ["a'b'c'd'", "a'b'cd", "a'bc'd", "a'bcd'", "abc'd'", "abcd", "ab'c'd", "ab'cd'"]  func\_DC = [] | ["a'b'c'd'", "abc'd'", "a'bc'd", "a'bcd'", "ab'c'd", "ab'cd'", "a'b'cd", 'abcd'] |
| 7 | func\_TRUE=all 211 combinations of terms  func\_DC=[] | [None] |
| 8 | func\_TRUE=["a'b'c'd","a'bcd","abc'd'","a'bc'd","a'bcd'","abc'd","abcd","ab'cd"]  func\_DC=[] | ['acd', "a'bc", "a'c'd", "abc'"] |

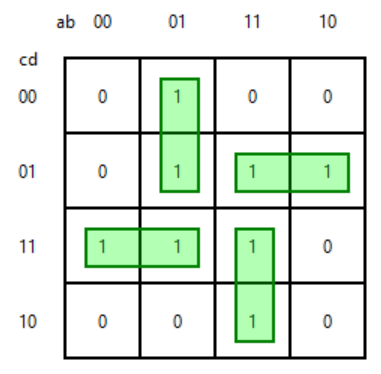
# How do these test cases validate our implementation?

The first 4 test cases are the samples provided in the problem statement.

The fifth one is an edge case when the Boolean function becomes a tautology.

In the sixth case, the function forms a ‘checkerboard’ on the KMap, i.e., no further optimization is possible.

The seventh one is a big test case in which the function is a tautology for 11 Boolean variables. This case ran in around 5-6 seconds on our devices.

In test case 8 the largest region is not present in the optimal solution:

After running all these tests, we are convinced that our algorithm is correct and efficient.

Test Case 8