Indian Institute of Technology Delhi

COL 215 - Digital Logic and System Design

Assignment 2

Maximize the region occupied by the minterms

Aryan Sharma, Garv Nagori
2021CS10553, 2021CS10549

Contents

1 Introduction **3**

2 Algorithm **3**

2.1 General Idea 3

2.2 Data Representation 3

2.3 Functions 5

3 Test Cases **10**

4 Conclusion **11**

1 Introduction

The objective of this assignment was to maximise the region occupied by the minterms by combining terms to get rid of extra variables. By combining the terms we can represent the expression in a simpler sum of terms. We have to combine all the 1's given in the K map. There is no such condition to include x's in the terms but to maximise some 1's we have to include some x's. Also, here we generalise to n-variable K maps.

2 Algorithm

2.1 General Idea

The main principle behind our algorithm is that two terms can only combine if the position of their "None"s are same and in the remaining bits, there is only one bit difference between the two.
Using this fact we can optimise our algorithm to only try to combine such terms. We sort the terms in groups according to the number of 1's in their binary representation and then try to combine adjacent groups. After each iteration we minimise the terms by one variable and the ones which didn't combine are added to a list. Lastly, we sort the list by the maximum regions and take the max region for any minterm.

2.2 Data Representation

We represent any term in the form of a Tuple,
The first element of the Tuple:

This stores an integer,all the active bits in a term are represented by 1 and the ones that are missing(None bit) are also represented by 1, the in-active bit is represented by a 0.

For example, if we are dealing with 5 variables,then, abc'de' will be 11010 which is 26, ac'd would be 11011 which is 27,here we place a 1 for active bit on a, as b is missing it is represented by a 1,so on, the representation will become 11011 which is 27.

The second element of the Tuple:

This stores an integer,all the bits are 0 except for the None bits, the None bits are represented by 1.

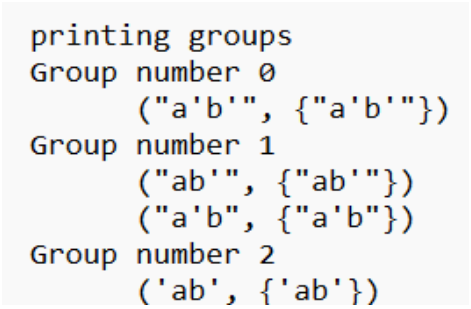
For example, if we are dealing with 5 variables,then,

1. abc'de' will be 00000 which is 0 as there are no None bits in abc'de'
2. ac'd would be 01001 which is 9 as the bits at b and e are None, all other bits are either 0 or 1.

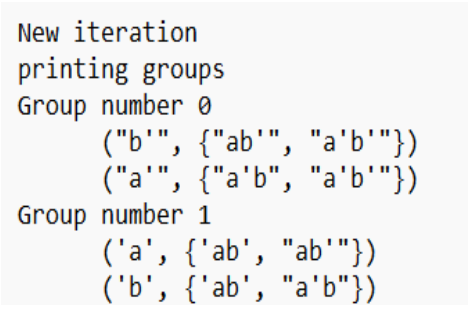
Thus in the end, the complete representation of abc'de' will be the Tuple (26,0) and for ac'd it would be (27,9).
Similarly, another example for a 6 term case,abd'e would be (69,9).

When we know the number of variables that we are dealing with, this representation is a bijection between the set of terms and the corresponding tuple, it is necessary for the representation to be a bijection as we encode the terms into this representation and then later we decode, so we shouldn't be getting terms which we did not intend to get.

In order to implement our idea of grouping the terms into groups having the same number of active bits(1 bit), We use a list of list containing Tuples. The list is named as groups, the list groups[i] contains the list of terms which have i active bits, each Tuple of the list contains three elements(i.e. it is a three-Tuple) the first two elements contains the representation of the term that this grouping represents and the third term is a set containing minterms which combine in order to give this term(these minterms are stored only as an integer, whose binary expansion represents this minterm).
For example, we print the groups before and after the first iteration for the case, $func\_True$ = ["ab'","ab"] and $func\_DC$ = ["a'b","a'b'"]



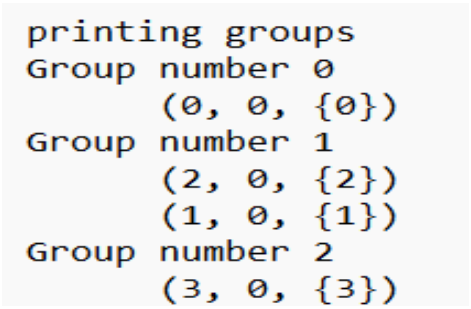
(a) Before Iteration



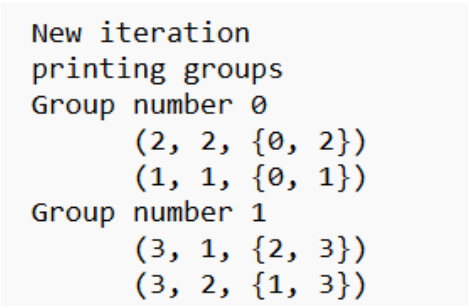
(b) After Iteration

Figure 1: What the groups mean

The way the group would be represented while the program runs after each iteration is shown below.



(a) Before iteration



(b) After iteration

Figure 2: Representation in the program

We have another list for storing the terms which could not be expanded in an iteration through the list groups,these terms are appended into a list called ans, this list also contains the Tuples which were explained above, these terms are the ones which cannot be expanded further.

2.3 Functions

1. **toBinary(str)**
   This function helps us to convert the input given in string to its binary representation. This function returns a number which is binary representation of the minterm passed in the function.

For example, if the minterm is " abc'd' ", its binary representation is 1100 therefore the function returns 12.

2. **combine(group,ans,n)**
   This function takes groups as input and keeps reducing groups by combining terms from consecutive groups and adding terms which cannot be combined to the list ans, this reduction in the number of groups keeps happening until all the groups are empty.
   We run a loop through all the different lists in group, this is the outer most loop in the program, inside this loop we run two nested loops, when we are iterating through the i^{th} list in group, we chose an element in the i^{th} list and check if it can be combined with some element of the $(i+1)^{th}$ list.
   We do the following operations in this function:

   (a) Maintain two sets, set1 and set2,set2 maintains the terms of $(i+1)^{th}$ list which have been combined in this iteration over i, set1 maintains the terms of list i which were combined in the previous iteration over i.

   (b) We check if two terms can be combined by first checking if the second element of the Tuple for both the terms are equal and then we check if the bitwise XOR of the first term of the two Tuples is a power of 2.Taking bitwise XOR of two integers takes O(1) time, the terms which can be combined are only the one's which have one different bit, thus bitwise XOR would have 0 at all positions except for the one at which a bit differs(thus,it will be a power of two).

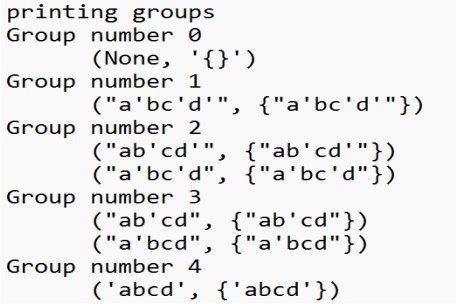
   (c) If two terms can be combined then we take the union of the third element(i.e. sets) and add it into the new groups, we add the bitwise XOR of the two terms to the second term of the new Tuple.

   (d) We maintain a boolean variable check which checks if an element combines in the i^{th} iteration, then we add this element into set2, the elements which do not combine are then added into answer.

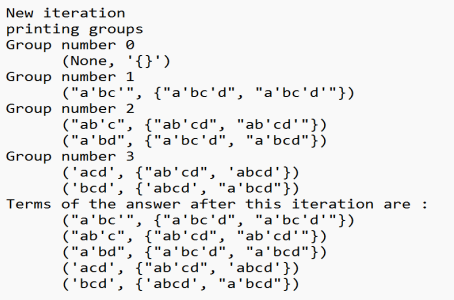
   (e) Our method for combining two terms written in our representation and such that the new term is also in our desired

representation, We replace the second element in the Tuple by the value of the second Tuple in the terms we were adding + bitwise XOR of the first element of the two terms we intend to combine,The first element is replaced by the maximum of the two terms we are combining, the third term is replaced by the union of the third element of the terms we are combining.

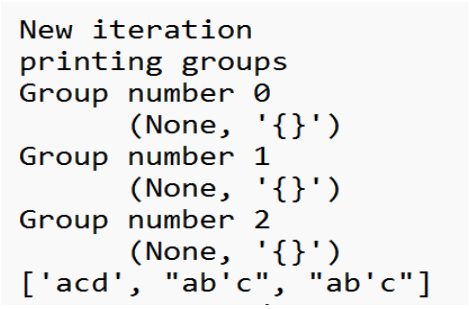
Below through print statement in the operation of this function we show how exactly combination of terms takes place through iterations for the input, func_True = ['abcd', "ab'cd'", "ab'cd"] and func_DC = ["a'bcd", "a'bc'd", "a'bc'd'"].



(a) First group created from func_True and func_DC



(b) After First Iteration



(c) After Second Iteration

Figure 3: Process of reducing groups

(a) Group 0 is empty so Group 0 after the iteration is also empty

(b) a'bc'd'in Group 1 combines with a'bc'd in Group 2 forming a'bc' in Group 1 after iteration

(c) ab'cd' in Group 2 combines with ab'cd of Group 3 to form ab'c, a'bc'd of Group 2 combine with a'bcd of Group 3.

(d) ab'cd of Group 3 combines with abcd of Group 4, a'bcd of Group 3 combines with abcd of Group 4

So on, combinations keep happening until the list groups becomes empty in the end.

3. **fromBinary(nbin,dbin,n)**
   This function converts the integers corresponding to the term and None to the string of the term.

   (a) We convert the integers to its binary representation in string using the *bin* function an removing the initial *0b* in the string.

   (b) Since the length of the string can be less than the number of variables in K map, we iterate from the last. The length of *dbins* is always less than *bins*.

   (c) If we encounter a '1' in the *dbins*, it represents a None therefore we just decrease the counter. If there is a '0' in *dbins* we check the value in *bins* and if it is '0' we append corresponding variable and an apostrophe in a list. If it is '1' we just append the corresponding variable.

   (d) There is just one slight issue with this. The length of *bins* may be less than the number of variables in K map. For the remaining *n-len(bins)* characters we append with an apostrophe since it is '0'.

   (e) After this, we reverse the list since we were iterating from the last. Finally we convert it to a string and return it.

4. **customPrint:**
   This function prints the list of list of Tuples groups and list of Tuples in a terms of strings and terms, which is useful for debugging and see the exact functioning of our program(especially the function *combine*), the pictures used in explanations in this report are the outputs produced from this function.

5. **comb_function_expansion(func_True,func_DC)**
   This is the main function which is called and all other functions are called by this. It operates in the following way:

   (a) At the very start we calculate the number of variables. Then, for each minterm in *func_TRUE* and *func_DC* we convert to its binary representation using the *toBinary* function. We count the number of active bits in it and append a list to its corresponding element in *groups*.

   (b) The list contains the binary representation of the minterms, another number which will represent the None bits(as mentioned in Data representation section) and a set which represents the minterms corresponding to that term(these three together). Additionally, we maintain a dictionary which stores the index of the term. The key is the binary representation of minterm and its value is the index of this minterm in *func_TRUE*.

   (c) We then call the function *combine* to minimize the minterms by expanding them to largest region. The *combine* function calls it recursively and finally returns a list *ans*. That list contains tuples comprising same three parameters - two integers to decode the term and the set of minterms which correspond to the given term.

   (d) Since for each minterm in *func_TRUE* we have to take the ***maximum*** legal region, we sort the *ans* list by the length of the set(third element in the Tuple) in an element. We declare a set *completeset* which consists of terms which have already been expanded or the don't care terms and initialize it to contain all the Don't Care terms.

   (e) For each tuple in *ans* we subtract the set *completeset* from the set in the tuple to get the terms which have not been added to the *output* list. We then convert the term from its two integer form to string using the *fromBinary* function. For each element in the *nonset* we check its index using the dictionary and change the value in *output* list to the string. After all the elements in *nonset* have been completed we include the *nonset* in the *completeset*.

(f) After iterating over the entire *ans* set, the *output* list is the final answer and we return it.

3 Test Cases

After making our algorithm and passing a few initial checks we run it against some test cases of our own making.

1. 4 Variable Test Case 1
   func_TRUE = ['abcd', "ab'cd'", "ab'cd"]
   func_DC = ["a'bcd", "a'bc'd", "a'bc'd'"]
   Output = ['acd', "ab'c", "ab'c"]

2. 4 Variable Test Case 2
   func_TRUE = ['abcd', "ab'cd'", "ab'cd"]
   func_DC = ["a'bcd", "a'bc'd", "a'bc'd'"]
   Output = ['acd', "ab'c", "ab'c"]

3. 6 Variable Test Case 1
   func_TRUE = ["abc'def", 'abcdef']
   func_DC = ["a'bcdef", "a'bc'd'e'f'", "a'bc'd'ef", "a'bc'def", "a'bc'd'e'f"]
   Output = ['bdef', 'bdef']

4. 6 Variable Test Case 2
   func_TRUE = ["abcd'e'f'", 'abcdef', "abcd'e'f", "abcd'ef"]
   func_DC = ["a'bcdef", "a'bc'de'f", "a'bc'def"]
   Output = ["abcd'e'", 'abcef', "abcd'e'", "abcd'f"]

5. 8 Variable Test Case
   func_TRUE = ["a'bc'defgh", "a'bc'd'efgh", "abc'd'efgh", "abc'defgh", "a'bc'def'gh", "a'bc'd'ef'gh", "abc'd'ef'gh", "abc'def'gh"]
   func_DC = []
   Output = ["bc'egh", "bc'egh", "bc'egh", "bc'egh", "bc'egh", "bc'egh", "bc'egh", "bc'egh"]

4 Conclusion

In this assignment we wrote an algorithm which combines minterms systematically and expands the legal region. we used data structures such as lists, tuples, sets and dictionaries to make groups and combine minterms to minimize the number of operations.

Q. Do all expansions result in an identical set of terms
A. All expansions may not result in an identical set of terms. There are multiple possible expansions possible for a minterm depending on the other minterms which it can combine with. For example, minterm "abcd" could expand into "ab" if "abcd"", "abc'd" and "abc'd"" are present. Similarly it can expand into "bc" if "a'bcd", "abcd' and" "a'bcd"" are present. Therefore it can combine to both if all the terms are given.
Q. Are all expansions equally good, assuming that our objective is to maximally expand each term? Explain.
A. All expansions are not equaly good since one expanded term may result in a bigger term, i.e. it can be expanded further whereas the other one may not. For example, "abcd" may expand into both "abc" and "bcd" but suppose "bcd" can further expand into "bc" then the expansion to "abc" is not good since that limits us to 2 cells, whereas the expansion to "bcd" gives us opportunity to grow more.

We try to expand into all possible regions but only take the best possible expansions for a term.