Indian Institute of Technology, Delhi

Department of Computer Science and Engineering

COL215P: Digital Logic & System Design

Semester l, 2022-2023

Assignment 3: Deleting Unnecessary Terms

Submitted by:

Ishita Chaudhary 2019CS10360

Kshitiz Bansal 2019CS50438

Date: *11/11/22*

Problem Description

Given a Karnauph map of "n" variables (no boundation on 'n'), for each cell with value "1", output the minimized literal count such that they cover all 1s. Input is given as two lists: func_TRUE: one with the positions where the K-map has value one, and func_DC: the other where it has value as "x"/do not care. From the expanded function, delete those terms whose corresponding K-map regions are fully contained within one or more other terms/regions.

Approach

Building further on the expand function used in the previous assignment. We get a list of maximally expanded regions for each corresponding true term in the func_TRUE list. From this, we construct a set of maximally expanded regions (each element in this is unique).

The idea is to see if a true term is covered by one or many of these expanded regions. If a true term is covered by only one of these expanded regions, then the corresponding expanded region has to be included in our final output.

Hence, we create a frequency dictionary, which will hold all true terms as keys and a list of expanded regions that cover the corresponding true term as value. The length of this list will give us the frequency of expanded regions that overlap the respective true term. We use the function "*isSubset()*", which returns true if an expanded region covers a true term, by comparing the values of literals in their binary form (0/1/None).

Now we start iterating on the regions in the set of maximally expanded regions. For each region, we iterate through the frequency dictionary and keep track of the minimum length of the value-list such that the corresponding region is present in the list of expanded regions that cover a true term. We can bifurcate our further implementation into two cases:

Case 1; If the minimum frequency is one, there exists at least one true term, which is only covered by the current expanded region. Hence, we add this region to our solution set. We again iterate through the frequency dictionary and set the value-list of all the true terms to null in which the current expanded region is present, as we have already added this to our solution set. The true terms with this region in their value set have also been covered now, and we need not iterate their lists again.

Case 2: If the minimum frequency is greater than one, it implies that all the true terms covered by the current expanded region are also being covered by at least one more expanded region. Hence, this region can be deleted. We iterate through the frequency dictionary and delete this region from the value-list of all terms.

Finally, we return the solution set.

We have also implemented a print function that prints all the regions we discarded. For each such region, we indicate all the true terms that the region covered and alternate expanded regions (after minimizing the literal count) which cover the corresponding true term.

Time Complexity

Let the number of terms in the func_TRUE list be n, and the K-map has p variables/literals. Given the list of maximally expanded terms, we will find the time complexity of the above-described algorithm. Note that the list of maximally expanded regions will also have n terms, a region for each true term.

1. Construction of a set of the maximally expanded regions: two nested iterations, one for each term and the other to check if the term is already present. Hence, $O(n^2)$.
In the worst case, if no two true terms combined to form an expanded region, the set of regions will also have n terms.

2. Construction of frequency dictionary: first loop iterating over all true terms; n times. The next nested for loop iterating over maximally expanded region set; n times. The next nested statement checks if a true term is a subset of the expanded region. For this, we need to convert both to binary form first. Binary conversion is O(p) because it iterates over the number of literals. Furthermore, checking subset is also O(p), as it compares each variable. Hence, the time complexity of this step is $O(n*n*(p+p+p)) \Rightarrow O(n^2p)$.
The size of this dictionary is also n, as true terms are keys. The maximum length of value-list is n, when a true term might be covered by every expanded region (maximum overlap).

3. Finding the solution set: first loop iterates over maximally expanded region sets; n times.
To find the minimum frequency of each region, we iterate in two nested for loops, one in frequency dictionary and the other in all value lists, both n times.
Irrespective of the minimum frequency value, we iterate over the frequency dictionary and value lists again; both n times. Hence, we get $O(n(n^2+n^2)) \Rightarrow O(n^3)$.

Combining everything, we get $O(n^2+n^2p+n^3)$.
For p variables, the maximum number of true terms is $\log_2 n$ (each variable has two choices:0/1, implying $n_{max}=2^p$, if all terms become 1). Substituting we get, $O(n^2+n^2\log_2 n+n^3)$ which is ***$O(n^3)$***.

This is in addition to the complexity of finding the maximally expanded region, which is again polynomial in terms of n, as described in the previous assignment. We can conclude the overall complexity to be polynomial in terms of the number of input variables.

Testing

We tested the above-described implementation for terms with up to 15 variables. We covered the edge cases where multiple expansions are possible for a term and then checked if our implementation outputs the minimum number of terms covering the required region.
For all the test cases and their corresponding maximal terms, we have tested for significant test cases with nine literals and more than 20 terms. Please check them in the above links. It is much neater there. We have added one big case on the next page for convenience.

Why is this set good enough to validate the implementation?
This set seems good enough because we have covered all the cases that could occur in general. At least for 4 variables, we can visualize all the corner cases (wrapping around, very high overlaps, very low overlaps, dense 1 map, scarce 1 map, dense DC map, scarce DC map, etc.) and thus check our implementation on each. We also test our implementation on larger test cases which contain ~ 15 variables and 20 terms. This can be considered as stress testing as bigger cases may require higher memory. Please see testcases and solutions for a list of all the test cases we tried.
Here are some of the test cases:

1. func_TRUE = ["a'bc'd'", "abc'd'", "a'b'c'd", "a'bc'd", "a'b'cd"]
   func_DC = []
   output = ["bc'd'", "a'c'd", "a'b'd"]
   This case checks correctness when there are no don't cares. Only 1s and 0s exist in the K-map. Output is as expected.

2. func_TRUE = []
   func_DC = ["a'bc'd'", "abc'd'", "a'b'c'd", "a'bc'd", "a'b'cd"]
   output = []
   This case checks correctness when there are no 1s. Only don't cares, and 0s exist in the K-map. Output is as expected. (because we need to expand true terms. There is no requirement to expand don't cares)

3. func_TRUE = ["a'b'c'd'", "abc'd'"]
   func_DC = ["a'bc'd'","ab'c'd'","a'b'c'd","a'b'cd'","a'b'cd","abc'd","abcd'","abcd"]
   output = ["c'd'"]
   This is the correct output. Here we see that when there are multiple possibilities for the expansion of a term, our algorithm correctly chooses the one which results in overlap with others and, thus, fewer terms in the final SOP form. This is the test case that has been widely discussed on Piazza.

4. func_TRUE=["a'b'c'd'", "ab'c'd'", "a'b'cd'","ab'cd'"]
   func_DC=["a'bc'd'","abc'd'", "a'bcd'","abcd'" ,"a'b'c'd","a'b'cd","ab'c'd","ab'cd"]
   output = [" d' "]

In this test case, there are 1s on all 4 corners. It checks if the algorithm works fine when there is a wrapping of the required legal region. The output is in agreement with the expected result.

5. 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 "]
   output = ["b'd ", "bc' ", "ac' "]
   This case is to check the working of high overlap. The output shows that our implementation can handle such cases correctly.

6. func_TRUE = ["abcdef'g'h'", "a'bcdefgh", "a'bc'd'e'f'gh", "a'b'cd'ef'g'h'", "abc'de'f'gh", "a'b'c'd'ef'g'h", "a'bc'd'efgh", "a'bc'defgh", "abcd'e'fg'h", "a'b'cd'e'fgh'", "a'b'cde'f'g'h'", "a'bc'd'efg'h", "a'b'cd'e'fgh", "ab'cde'f'gh'", "ab'cd'e'f'gh'", "ab'cd'e'fg'h", "a'bc'd'e'fg'h'", 'abcdefgh', "a'b'c'd'efgh", "a'b'cdef'g'h'", "a'b'c'def'g'h'", "abc'de'f'g'h", "abc'd'e'f'g'h'", "abcd'e'fgh'", "ab'cd'e'fgh'", "abc'd'efg'h'", "a'bcdefg'h", "a'bc'de'fgh'", "ab'cd'ef'gh'", "ab'c'de'f'g'h", "abc'de'fg'h", "a'bcdef'gh'", "abc'd'ef'gh'", "ab'c'defgh", "abcd'e'f'gh'", "ab'cd'efg'h", "ab'cde'fg'h'", "a'bc'de'fg'h", "a'b'cde'fgh'", "a'bcde'fg'h'", "abc'd'e'f'g'h", "a'bcdef'gh", "a'b'c'de'f'g'h'", "ab'c'd'efgh", "a'b'cdefgh", "a'bcd'e'f'g'h'", "ab'c'd'ef'gh", "a'bcd'e'f'g'h", "a'b'c'def'g'h", "a'b'cd'e'fg'h", "ab'cde'f'g'h", "a'b'c'def'gh'", "ab'c'de'fg'h'", "abcdefg'h", "abcd'ef'gh'", "a'b'cd'ef'g'h", "a'b'c'de'fg'h'", "a'b'c'd'ef'g'h'", "ab'c'def'g'h'", "a'b'c'd'ef'gh", "abc'def'g'h'", "abcdef'gh", "a'b'c'd'e'fgh'", "a'bc'def'gh", "abcde'fg'h", "a'b'cdefgh'", "ab'cdefgh", "a'bcde'f'gh", "ab'c'def'gh", "ab'cdefg'h", "a'bc'de'f'g'h", "abc'defg'h", "ab'c'de'fg'h", "ab'c'd'e'f'gh", "ab'c'd'e'fg'h"]
   func_DC = ["a'bcdefg'h'", "abcde'fgh'", "a'b'cde'fgh", "a'b'cdef'gh", "a'b'cde'f'gh'", "a'bc'd'e'f'g'h", "ab'cdefg'h'", "a'bc'de'fgh", "abcd'efg'h", "abcdefgh'", "ab'c'd'ef'gh'", "ab'c'defg'h'", "abcdef'gh'", "abcd'efgh'", "a'bc'de'f'gh", "a'bc'd'ef'gh'", "a'b'cde'fg'h", "a'b'c'd'e'f'gh", "a'bcd'e'fgh", "ab'c'de'f'gh", "abcd'efgh", "a'b'cdef'gh'", "a'b'cde'f'g'h", "abc'de'fgh", "a'bcd'ef'gh", "abc'd'e'fg'h'", "ab'c'd'e'fg'h'", "a'bc'def'gh'", "a'b'c'd'efg'h", "ab'c'defgh'", "abc'def'gh", "a'bc'de'f'g'h'", "a'bc'def'g'h", "abc'd'ef'gh", "abcdef'g'h", "a'bcd'ef'g'h'", "abc'defgh'", "a'b'c'defg'h'", "a'b'c'de'fg'h", "a'b'c'd'e'f'g'h'", "a'bc'd'ef'g'h", "a'b'cd'efg'h'", "abc'de'f'g'h'", "a'bcdefgh'", "a'bc'd'efg'h'", "abc'defg'h'", "a'bcde'fgh", "a'b'c'd'e'f'g'h", "a'b'cd'efgh'", "a'b'c'de'fgh'", "a'b'cd'e'fg'h'", "a'b'cd'ef'gh'", "ab'cd'ef'g'h", "a'bcd'e'fgh'", "ab'c'd'ef'g'h", "a'b'c'de'fgh", "abc'd'efgh'", "ab'cdef'g'h", "a'bc'd'e'f'g'h'", "a'b'cdefg'h", "abcde'f'gh", "ab'cd'e'fg'h'", "a'bc'defg'h", "a'b'cdef'g'h", "a'bc'd'ef'g'h'", "a'b'c'de'f'gh", "a'bc'd'e'fgh'", "a'b'c'd'e'f'gh'", "abcde'f'g'h'", "a'b'cd'efg'h", "ab'cd'ef'gh", "abc'def'gh'", "a'b'c'd'e'fg'h'", "abc'd'e'fgh'", "ab'c'defg'h", "abcde'f'gh'", "abc'defgh", "abcd'e'f'g'h'", "a'bcd'e'fg'h", "a'bc'd'ef'gh", "ab'c'de'f'gh'", "a'bcdef'g'h", "ab'cd'ef'g'h'", "a'b'c'defg'h", "a'b'c'd'e'fg'h", "abc'd'ef'g'h'"]
   output = ["abdf'g'h'", "a'c'd'f'h", "bdf'gh", "a'c'd'eh", 'bdegh', "cd'e'fg'h", "cd'e'fgh'", "a'b'df'g'h'", "a'ce'fgh", "ace'f'gh'", "bc'd'g'h'", "bc'e'f'g'", "a'c'e'fgh'", "b'cd'ef'h'", "ac'de'g'h", "c'dfg'h", 'adefh', "b'ceg'h", "ab'e'fg'h'", "a'b'e'fgh'", "a'bcdfg'h'", "b'c'd'egh", "a'bd'e'f'g'", "a'deg'h", "b'cdf'g'h", "a'def'gh'", "b'c'e'fg'", "abegh'", "a'c'd'f'g'", "b'c'deg'h'", "abdfg'h", "a'cdeg", "ac'ef'gh", "b'c'e'f'gh"]

This is one of the bigger testcases that we tested our implementation on. Comparing our output with that of other students, we see that the number of terms is minimal.

7. func_TRUE = ["abc'd'ef'", "a'b'c'de'f'", "a'b'cde'f'", "abcde'f", "ab'c'd'ef'", "a'bc'd'ef'", "abcde'f'", "abcd'e'f", "abc'def'", "abcd'ef'", "a'bcd'e'f"]
   func_DC = ["a'b'cd'ef'", "a'bc'de'f", "ab'cd'ef", "a'bcdef", "a'bc'def'", "ab'cde'f'", "abcd'ef", "abc'de'f'", "abcdef'", "a'b'c'd'e'f", "abc'de'f", "a'bcde'f"]
   output = ["bc'ef'", "a'b'de'f'", "ac'd'ef'", "abdf'", "bce'f", "abef'"]

8. func_TRUE = ["abcde'fgh'", "a'b'c'd'e'fg'h", "a'bcde'fgh'", "a'bcde'f'g'h'", "a'b'c'd'ef'gh'", "a'b'c'def'g'h", "a'b'c'd'ef'gh", "a'bcd'ef'gh'", "abcdef'gh", "a'bc'd'e'f'g'h'", "ab'cde'fg'h'", "a'b'c'd'efgh'", "a'bcd'ef'g'h'", "a'bc'd'e'fg'h", "abc'de'fg'h'", "a'bc'd'e'fg'h'", "a'bc'defgh", "abcd'e'f'gh", "abc'de'fgh", "abcd'efgh", "ab'cde'f'g'h'", "a'bcde'f'gh", "abcde'fg'h'"]
   func_DC = ["abc'd'e'f'g'h'", "a'bcdef'g'h", "a'bc'de'f'g'h'", "a'bcd'efg'h", "abc'd'ef'g'h'", "a'b'cde'fgh", "a'b'c'defg'h", "ab'c'd'efg'h'", "ab'c'd'e'fgh'", "abcd'e'f'g'h'", "ab'cdefgh'", "a'bcde'f'g'h", "a'bc'd'ef'gh'", "a'bcde'f'gh'", "ab'cd'e'f'g'h", "a'bc'defg'h", "a'bc'd'e'fgh'", "ab'c'de'fg'h", "a'bc'd'e'f'gh'", "abc'd'e'fg'h", "a'bc'defg'h'", "ab'cd'efgh", "abc'defg'h'"]

   output = ["bcde'fgh'", "a'c'd'e'fg'h", "a'bcde'f'", "a'b'c'deg'h", "a'b'c'd'ef'g", "abcdef'gh", "a'bc'd'e'h'", "a'b'c'd'egh'", "a'bcd'ef'h'", "abde'fg'h'", "a'bc'defh", "abcd'e'f'gh", "abc'de'fgh", "acd'efgh", "ab'cde'g'h'"]
9. func_TRUE = ["a'bc'd'", "abc'd'", "a'b'c'd", "a'bc'd", "a'b'cd"]
   func_DC = ["abc'd"]
   output = ["bc'", "a'b'd"]