COL 215 SW3

Rajat Bhardwaj

2020CS50436

Geetansh Juneja

2020CS50649

Explanation of our algorithm

First we calculated the set of largest legal regions.
We first found a set that represents all the legal regions. We did that as follows. We inserted a term(from True + dont care) in a set, before inserting we checked whether there exists a term in the set already which can combine with this term to double the region. If yes we remove both terms from the set and add the double region term in the list so that it can be checked and added in the future. If there doesnot exist any term in the set which can help in doubling the area we simple insert that term in the set.
Basically we found a string with one change in the set (example for ”abcd’e”’ if ”ab’cd’e”’ is present in the set already , we remove both ”abcd’e”’ and ”ab’cd’e”’ and we add ”acd’e”’ in a list which has to be checked in future.
We added the set to an array and we sorted all the elements of the array such that the elements with less literals lies before the elements with more literals. Now for each term in the True region, we iterate in the array, if we find that the term lies in the region of the element of the array, we return this region (this region will be maximal since the array is sorted in increasing size of the literals i.e. decreasing size of the region) and break the iteration.
Hence we find all the maximal regions.

Now we need to find the minimum number of areas that will cover all the terms. For this we used quine mccluskey method.
First we found out the essential prime implicants by checking if there exist any term in the input for which there is only one legal region possible.
Now we have all the prime implicants. For the remaining terms for which the essential prime implicants do not cover them, we followed hit and trial method as solving this problem via petrick's method is exponential(NP hard)
Thus for the cost of optimality we made a polynomial time algorithm for it.

We selected any area of any term which is not yet covered by our answer, we inserted that area and deleted all the terms that were coverd by this area, we repeated untill no terms were left. Thus we end up with a suboptimal solution. All the helper functions are same as previous assignment, you can look at the report of that for the explanation of the previous functions.

Explanation of functions

find_essentialPI(term_red)

This function finds out the essential prime implicants form a dictionary term_red, mapping the input terms with the maximal areas.

demo(EssentialPI ,term_reg)

This function is used to print the terms for which regions are to be deleted, the deleted region and the region covering that term

Time complexity

First we call the insert(regionset , F , legal_regions) function.
for the inner ForLoop, length of potential terms can be equal to the size of term, therefore number of iteration of inner loop is = max size of term
for outer loop, the iterations is of the order of number of input terms
Thus its complexity = O(nk)
where k = size of input list
n = number of literals in a term
Now size of final list will also be of the order k
Now for mapping the terms of the input to the area is of the order O(k^2)
Finding EPI is of the order O(k)
We do hit and trial in the end which O(k^2)

Thus the overall complexity is O(nk + k^2)

Test cases

Test case 1

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"]
In the above test case we can test all the essential prime implicants, since all the terms are coverd by EPIs

Test case 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"]
output = ["ac'", "b'd", "bc'"]
In the above test case, there were some terms that were not covered by the EPIs therefore we test our stretagy for assigning regions to the remaining terms here

Test case 3

func_TRUE = ["a'b'c", "a'bc", "a'bc'", "ab'c'"]
func_DC = ["abc'"]
output = ["a'c", "ac'", "a'b"]

Test case 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 = []
Output:
["c'd'e'", "abd", "bc", "bde", "cd'e"]
output = ["c'd'e'", 'bc', 'bde', "cd'e", 'abd']

Larger test case to check the complexity

Test case 5

func_TRUE = ["a'b'c'd'e'fg", "a'bc'd'e'fg'", "abc'd'efg'", "ab'c'd'efg'", "abc'defg'", "abcdefg", "a'bcdefg'", "a'bcd'ef'g'", "abcd'e'fg", "a'bc'defg", "abc'defg", "abcdefg", "a'bcdefg", "a'bcd'efg", "abcd'efg", "a'b'cd'efg", "ab'cd'efg", "abcdef'g", "a'bcdef'g", "a'bcd'ef'g", "abcd'ef'g", "a'b'cd'efg'", "ab'cd'efg'"]
func_DC = []
output = ["a'b'c'd'e'fg", "a'bc'd'e'fg'", 'abdef', "a'bcdefg'", "a'bcd'ef'", "abcd'fg", 'bdefg', 'bceg', "b'cd'ef", "abc'efg'", "ac'd'efg'"]
Larger test case to check the complexity