REPORT FOR COL215 ASSIGNMENT-3

Objective:

The objective of the assignment is to write a python program containing a function that takes arguments as two lists, namely func_TRUE and func_DC. It then returns regions with least number of literals, by deleting redundant terms from the cells after expanding each term maximally.

Approach:

The objective can be achieved by breaking the process into five steps. Initially, we create a class 'Term' in order to increase the efficiency of the program. The class is used to represent a region containing one or more cells. Now, we group the input terms depending on the number of 1s they contain. We then obtain the maximally expanded regions. These initially expanded regions are what we call prime implicants. We then have to delete the redundant regions to obtain the final output. To do this, we first create a dictionary and then perform certain operations to get the final output.

Step-1:

We initially implement a class called 'Term'. This contains three attributes. It contains list of 0s and 1s,representing regions, namely 'boollist'. Further, we also store the minterms (Eg. 0, 2, 5) which are combined to create the region. This allows us to greatly optimize as we already know the individual cells which make up the given region and do not need to find them. Lastly, it also contains a attribute 'used', and a corresponding method use( ) which helps us keep track of whether or not this term can be expanded.

Step-2:

The process of finding the prime implicants is achieved by use of a helper function. The helper function first creates a 2-D list called 'groups' which keeps track of regions based on the number of 1's. Observe that two regions can be combined only if difference of number of 1s (group number) is exactly one as this implies that they differ by only one bit. This step is not necessary but helps reduce the constant factor in the time taken for computation.

Step-3:

We first check if ‘groups’ has a size of one in which case we can directly return its first term. If not, then we loop over ‘groups’ to compare consecutive groups that is those with a difference of one in the number of 1s. For the terms in these two groups, we check if combination is possible. If it is, then we mark the terms as ‘used’ and append the combined term to ‘new_groups’. We now store all the terms which remain unused in yet another list. We can recurse with ‘new_groups’ as input to see if further combination is possible. The recursion will ultimately append all the final prime implicants into ‘unused’.

Step-4:

Let us now begin removing the redundant regions from the obtained prime implicants. We create a python dictionary for this purpose. It maps the minlist numbers or cells (Eg. 1, 4, 7) to prime implicants containing these cells. This allows us to immediately access all the regions associated with any given cell which is used below.

Step-5:

Once created, we loop over minterms of each of the prime implicants. If a particular minterm has only one region containing it and it is not a ‘DC’ term, then we can call it essential and store it. Observe that if not a single minterm corresponding to the current prime implicant is essential then we can remove this region from consideration as it will be covered in some other suitable regions. Thus, we use a variable ‘indicator’ to tell us the case in which we can remove the prime implicant from the dictionary using the minterms. This process repeats for each prime implicant finally giving us the needed regions.

Time Complexity:

Let us analyse the time complexity of the above algorithm in terms of 'n' which is the number of variables and 'L' which is the length of 'func_TRUE'. We first observe the time complexity of a few helper functions used.

(1) Combine:

This function compares two lists of length n. It then loops over a list of length n performing constant operations in each iteration. Therefore, the net time complexity is $O(n+n) = O(n)$.

(2) Lit_to_bool:

We create a list of length n. We then loop over a string whose length is at most 2n when all the literals are complements. We also loop over the initially created list. Hence, the net time complexity is $O(n+2n+n) = O(n)$.

(3) Bool_to_lit:

We loop over a list of length n. In each iteration, we use string concatenation. Thus, the overall time complexity is $O(n*n) = O(n^2)$.

(4) Initial_grouping:

We create a 2-D list containing n+1 lists. We loop over the true terms that is a list of length L. In each iteration, we loop over another list of length n. Hence, net time complexity is $O(n+nL) = O(nL)$.

(5) Get_prime_implicants:

We call the above function (4) once. We then create a new 2-D list of size at most n. We then have nested for loops of size of the order of n, L and L. In the innermost level of these loops we call combine and use 'in' operator on a list of length L. Lastly, we use a 2-nested for loop with 'in' operator inside. The maximum recursion depth is n. Thus, the net time complexity is $O(n*(nL + n + n*L*L*(n+L) + n*L*L)) = O(n^3L^2+n^2L^3)$.

Opt_function_reduce:

We initially have three separate loops of size n, L and L with a call to (2) after which (5) is called. We then have a 2-nested for loop over prime implicants. Lastly, we have another such nested loop with an “in” operator and a call to (3).

Thus, the overall time complexity of the algorithm is

$O(n+ L*n + L*n + n^3L^2+n^2L^3 + L*L + L*L*L*n^2 + L)$
$= O(n^3L^2+n^2L^3)$

Testcases:

1. Testcase-1 (no cell is present in only a single prime implicant)

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

func_DC=[ ]

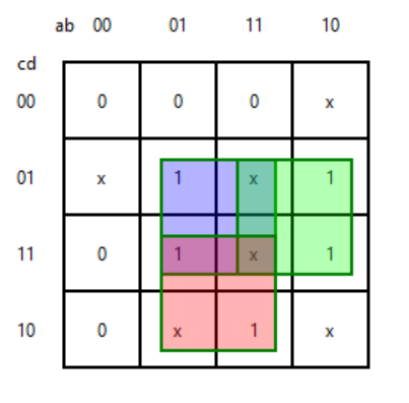

```
270  print(opt_function_reduce(["a'b'c'd'","a'b'c'd","a'b'cd'","a'bc'd",
271  "a'bcd","a'bcd'","abc'd'","abc'd","abcd","ab'c'd'","ab'cd","ab'cd'"],
272  []))

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  JUPYTER

PS C:\Users\santh\Desktop\All courses\Sem3\COL215\Assignments\Assignment-3> py
.\2021CS10564_2021CS10561_assignment_3.py
Deleted term: a'b'c'
Deleted term: a'cd'
Deleted term: ac'd'
Deleted term: ab'c
Deleted term: bd
["a'c'd", "a'bc", "abc'", 'acd', "b'd'"]
```

The selection of this case is mainly to emphasize the fact that the algorithm works even when there are no essential prime implicants in the beginning . Even if we take the middle square, we still have to select remaining four 1*2(or 2*1) and 2*2(“b’d”) terms, so the middle one is redundant here. The algorithm takes care of this as shown above.

2. Testcase-2 to emphasize on ‘DC’ terms:

Func_TRUE = ["a'bc'd","a'bcd","abcd'","ab'c'd","ab'cd"]

Func_DC = ["a'b'c'd","a'bcd'","abc'd","abcd","ab'c'd'","ab'cd'"]



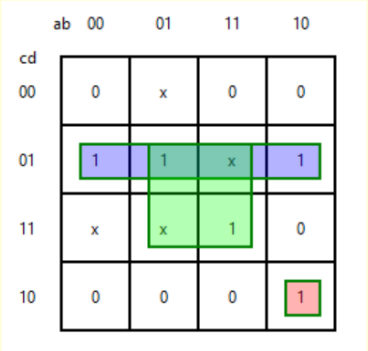

```
268   print(opt_function_reduce(["a'bc'd","a'bcd","abcd'","ab'c'd","ab'cd"],
269   ["a'b'c'd","a'bcd'","abc'd","abcd","ab'c'd'","ab'cd'"]))

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  JUPYTER

PS C:\Users\santh\Desktop\All courses\Sem3\COL215\Assignments\Assignment-3> py
.\2021CS10564_2021CS10561_assignment_3.py
Deleted term: c'd
Deleted term: ab'
Deleted term: bc
['bd', 'ad', 'ac']
```

This testcase was chosen because it clearly shows us that the algorithm gives us the optimum solution even if ‘DC’ terms are present. Observe that the bottom-right ‘x’ must be considered as a 1 while finding the prime implicants. However, the algorithm eventually rules it out. Thus, don’t care terms are used only when needed.

3. Testcase-3

Func_TRUE = ["a'b'c'd","a'bc'd","ab'c'd","abcd","ab'cd'"]

Func_DC = ["a'bc'd'","abc'd","a'b'cd","a'bcd"]




```
268   print(opt_function_reduce(["a'b'c'd","a'bc'd","ab'c'd","abcd","ab'cd'"],
269   ["a'bc'd'","abc'd","a'b'cd","a'bcd"]))

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  JUPYTER

PS C:\Users\santh\Desktop\All courses\Sem3\COL215\Assignments\Assignment-3> py .\2
021CS10564_2021CS10561_assignment_3.py
Deleted term: a'bc'
Deleted term: a'd
["ab'cd'", "c'd", 'bd']
```

This testcase clearly shows that the algorithm works in case of isolated 1s. It also does not unnecessarily include any ‘DC’ terms (such as a’b’cd) as we can see in the above kmap.