COL215 Assignment 2

- Report by Piya Bundela(2021CS10118) and Aastha Rajani(2021CS10093)

Assignment 2 of col215 makes us use knowledge of K-maps. Here we are provided two lists, one which contains the cells which have 1 in the k map and other which contains the cells which have 'x' in the k map. We are asked to output the maximum expanded region corresponding to each cell containing 1.

Here, input is in form of two list, fun_true- that has cells containing 1 and fun_dc – that contains cells with x.

We have defined several helper functions

1- reverse_conversion- This function changes the complemented literals in a string to their corresponding uppercase literals. This function has been defined In order to deal the inputs with a greater ease.
2- conversion- This function changes the uppercase literals to their complemented lower case literals.
3- xor- This function outputs the xor(exclusively or) of two binary strings. It has been defined to find the strings which differ only in one literal.
4- str_to_num- This function converts the uppercase literals to 0 and lowercase literals to 1.
5- del_hash- This function removes the hash from a given string.

Firstly, we merge both the input lists into one list- 'lis' after letting it pass through reverse_conversion to bring them in usable forms. We then convert them to binary form using str_to_num function and append it to 'x'(list). We now iteratively take xor of every element of x with every other element of 'x' to identify terms that differ by only one literal. The xor of such strings will contain only one '1' and rest '0's. After finding such strings we drop that only differing literal and replace it with '#' and append it to 'fin'(list). Also, we have created another xor_list_sec which stores all such strings whose xor output have already been stored in 'fin'. If any string neither satisfies the above conditions nor is present in the xor_list_sec we append as it is to our list 'fin'.

We then pass this one-time simplified list to another function 'simplify'. This function first checks if the string contains '#' or not. If "#' is not there, we append it to new list 'val', else we find any other string which contains '#' in the same position as in our above mentioned string. If no such string is found, we append it to 'val', else we calculate the xor of such strings by converting them to binary form x' to identify terms that differ by only one literal. The xor of such strings will contain only one '1' and rest '0's. We then replace that only differing literal by '#' . Also, we have created another xor_list which stores all such strings whose xor output have already been stored in 'val'. If any string neither satisfies the above conditions nor is present in the xor_list we append as it is to our list 'val'.

Now, we recursively call this simplify function to further simplify the list. The function terminates when the newly simplified list is exactly the same as the list passed as input.

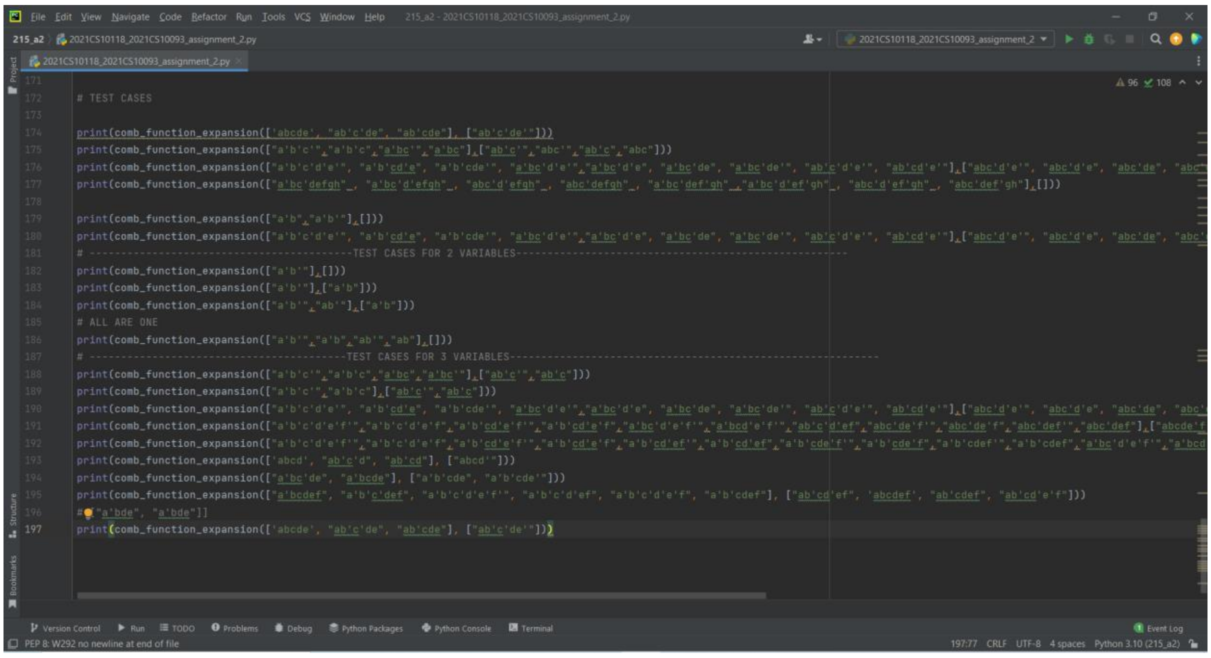
We then deleted the hash from every element and appended this to a new list del_hash_list.

In this most simplified list we find all such output string which are common to the a single input string and returns that output string which has minimum length(maximum expanded region).

This is repeated for all the input strings .

We check for each element in func_TRUE to see if it belongs to the expanded regions we have obtained in del_hash_list (or fin), and make a list of such possible regions. Then we find the largest of such regions, and append it to the answer_list and in the end we return the converted form of answer_list.

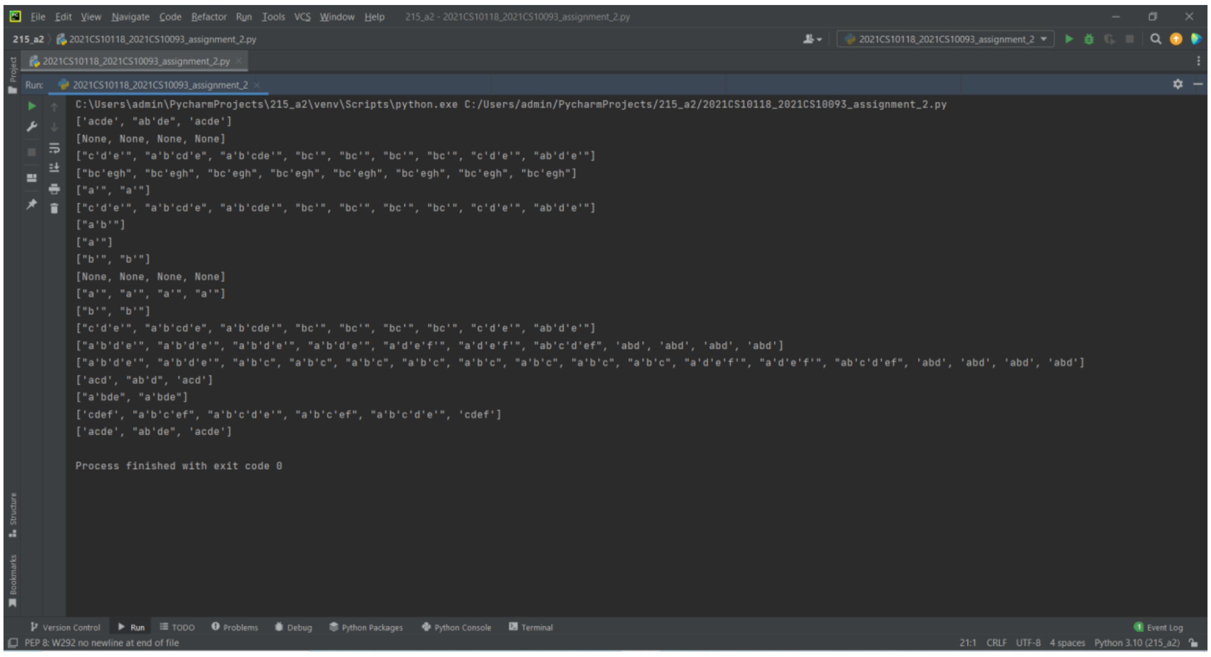
Test Cases:



```
# TEST CASES

print(comb_function_expansion(['abcde', "ab'c'de", "ab'cde"], ["ab'c'de'"]))
print(comb_function_expansion(["a'b'c'","a'b'c","a'bc'","a'bc"],["ab'c'","abc'","ab'c","abc"]))
print(comb_function_expansion(["a'b'c'd'e'", "a'b'cd'e", "a'b'cde'", "a'bc'd'e'","a'bc'd'e", "a'bc'de", "a'bc'de'", "ab'c'd'e'", "ab'cd'e'"],["abc'd'e'", "abc'd'e", "abc'de", "abc'
print(comb_function_expansion(["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"],[]))

print(comb_function_expansion(["a'b","a'b'"],[]))
print(comb_function_expansion(["a'b'c'd'e'", "a'b'cd'e", "a'b'cde'", "a'bc'd'e'","a'bc'd'e", "a'bc'de", "a'bc'de'", "ab'c'd'e'", "ab'cd'e'"],["abc'd'e'", "abc'd'e", "abc'de", "abc'
# ----------------------------------------TEST CASES FOR 2 VARIABLES------------------------------------------------------
print(comb_function_expansion(["a'b'"],[]))
print(comb_function_expansion(["a'b'"],["a'b"]))
print(comb_function_expansion(["a'b'","ab'"],["a'b"]))
# ALL ARE ONE
print(comb_function_expansion(["a'b'","a'b","ab'","ab"],[]))
# ---------------------------------------TEST CASES FOR 3 VARIABLES------------------------------------------------------
print(comb_function_expansion(["a'b'c'","a'b'c","a'bc","a'bc'"],["ab'c'","ab'c"]))
print(comb_function_expansion(["a'b'c'","a'b'c"],["ab'c'","ab'c"]))
print(comb_function_expansion(["a'b'c'd'e'", "a'b'cd'e", "a'b'cde'", "a'bc'd'e'","a'bc'd'e", "a'bc'de", "a'bc'de'", "ab'c'd'e'", "ab'cd'e'"],["abc'd'e'", "abc'd'e", "abc'de", "abc'
print(comb_function_expansion(["a'b'c'd'e'f'","a'b'c'd'e'f","a'b'cd'e'f'","a'b'cd'e'f","a'bc'd'e'f'","a'bcd'e'f'","ab'c'd'ef","abc'de'f'","abc'de'f","abc'def'","abc'def"],["abcde'f
print(comb_function_expansion(["a'b'c'd'e'f'","a'b'c'd'e'f","a'b'cd'e'f'","a'b'cd'e'f","a'b'cd'ef'","a'b'cd'ef","a'b'cde'f'","a'b'cde'f","a'b'cdef'","a'b'cdef","a'bc'd'e'f'","a'bcd
print(comb_function_expansion(['abcd', "ab'c'd", "ab'cd"], ["abcd'"]))
print(comb_function_expansion(["a'bc'de", "a'bcde"], ["a'b'cde", "a'b'cde'"]))
print(comb_function_expansion(["a'bcdef", "a'b'c'def", "a'b'c'd'e'f'", "a'b'c'd'ef", "a'b'c'd'e'f", "a'b'cdef"], ["ab'cd'ef", 'abcdef', "ab'cdef", "ab'cd'e'f"]))
#["a'bde", "a'bde"]]
print(comb_function_expansion(['abcde', "ab'c'de", "ab'cde"], ["ab'c'de'"]))
```



```
C:\Users\admin\PycharmProjects\215_a2\venv\Scripts\python.exe C:/Users/admin/PycharmProjects/215_a2/2021CS10118_2021CS10093_assignment_2.py
['acde', "ab'de", 'acde']
[None, None, None, None]
["c'd'e'", "a'b'cd'e", "a'b'cde'", "bc'", "bc'", "bc'", "bc'", "c'd'e'", "ab'd'e'"]
["bc'egh", "bc'egh", "bc'egh", "bc'egh", "bc'egh", "bc'egh", "bc'egh", "bc'egh"]
["a'", "a'"]
["c'd'e'", "a'b'cd'e", "a'b'cde'", "bc'", "bc'", "bc'", "bc'", "c'd'e'", "ab'd'e'"]
["a'b'"]
["a'"]
["b'", "b'"]
[None, None, None, None]
["a'", "a'", "a'", "a'"]
["b'", "b'"]
["c'd'e'", "a'b'cd'e", "a'b'cde'", "bc'", "bc'", "bc'", "bc'", "c'd'e'", "ab'd'e'"]
["a'b'd'e'", "a'b'd'e'", "a'b'd'e'", "a'b'd'e'", "a'd'e'f'", "a'd'e'f'", "ab'c'd'ef", 'abd', 'abd', 'abd', 'abd']
["a'b'd'e'", "a'b'd'e'", "a'b'c", "a'b'c", "a'b'c", "a'b'c", "a'b'c", "a'b'c", "a'b'c", "a'b'c", "a'd'e'f'", "a'd'e'f'", "ab'c'd'ef", 'abd', 'abd', 'abd', 'abd']
['acd', "ab'd", 'acd']
["a'bde", "a'bde"]
['cdef', "a'b'c'ef", "a'b'c'd'e'", "a'b'c'ef", "a'b'c'd'e'", 'cdef']
['acde', "ab'de", 'acde']

Process finished with exit code 0
```

Q. Do all expansions result in an identical set of terms?

Ans. No, terms wouldn't be identical as we are expanding in different directions in different step. But these terms would eventually combine and form maximal region.

Q. Are all expansions equally good, assuming that our objective is to maximally expand each term? Explain.

Ans. In our algorithm, in the intermediate, expansions are taken from different directions which eventually returns the maximum possible expansion. Therefore we can say that all expansions are equally good.