Indian Institute of Technology, Delhi

Department of Computer Science and Engineering

COL215P: Digital Logic & System Design

Semester l, 2022-2023

Assignment 2: Boolean Function Expansion

Submitted by:

Ishita Chaudhary 2019CS10360

Kshitiz Bansal 2019CS50438

Date: *26/10/22*

Problem Description

Given a Karnauph map of "n" variables (no boundation on 'n'), for each cell with value "1", output the maximum legal region. A legal region can consist of 1s and x's but cannot contain any 0s. 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.

Approach

We use the data structure Binary Search Tree to solve this problem. The node of this tree contains four fields: key, which contains the term in a binary list form (with 0s, 1s, and None), a pointer to the left node, a pointer to the right node, and a tuple of the form (list, int) which stores maximum expanded term for the corresponding node and the count of how many times the term has been expanded respectively. In each expansion, the area of the legal region becomes twice, and we drop a variable from the term. For example, a'b + ab -> b.
We use a dictionary {0:0, 1:1, None:2} to compare two terms and insert a new node in the BST.
Insertion and searching in BST: we compare each index of the term (in 1s, 0s, "None"s form) using the above-mentioned dictionary and use the BST property (higher key value to the right and lower key value to the left) to insert/ find a node.

We use the following helper functions:
1. toLiteral: this function takes a term as an input in the form of a list of 0s, 1s, and "None"s and returns it into its literal string form. For example, input: [1, 0, None, 0], output: ab'd'
2. toBinaryList: this function takes input a list of terms in their literal forms and outputs a list of terms converted in their binary format. Each string term is now a list of 0s, 1s, and "None"s.
3. isLegal: this function returns a boolean true if the given term is legal; otherwise false. If the term contains a None field, it is replaced by 1 and 0, and the status of both these terms is checked by calling the function recursively until there is no "None" field left in the term. If the term does not have any None field, then we check if the term is present in either of the initially given lists (list with 1s and list with Xs). If yes, we store this term in the nextLegal list of terms for the given term.
We can also print the next legal terms for each input term by calling the function printNextLegal.

For each term present in the input list func_TRUE, the term is maximally expanded using the *expand* function, which recursively expands the given term while also adding the intermediate terms into the BST, if not already present. For every variable in the term which is not `None`, we check if the term formed by just negating this variable is legal using the

``isLegal()`` function. If yes, we combine these two terms and change this variable to `None` to form an intermediate-term and recursively call ``expand()`` on this intermediate term. While doing this, we also keep track of the maximum decrement in the number of non `None` variables and the corresponding maximally expanded term. At the end of each intermediate recursive call, we store this intermediate term in the BST along with its maximum decrement and maximally expanded term so that we never have to perform the same calculation again, similar to how we memoize in dynamic programming.

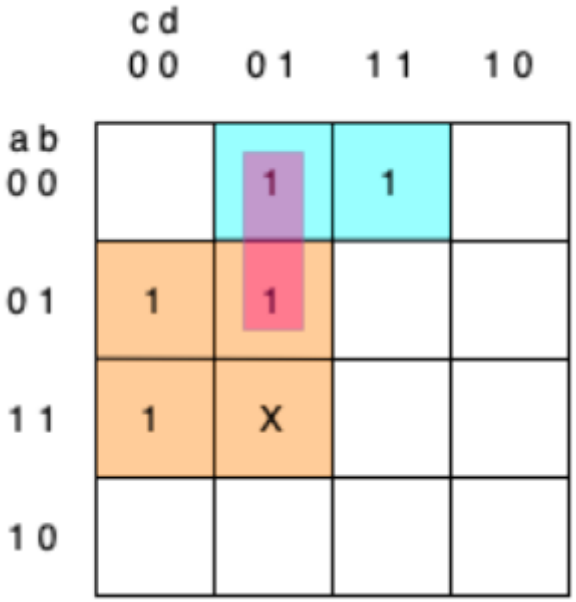
Inferences

Do all expansions result in an identical set of terms?

No, all expansions do not result in the same set of terms. There are numerous possible expansions for a term depending on the terms it combines within its neighborhood.

A term may even have two expansions of the same size but different regions (horizontal and vertical).

Here, if expanding the term a'b'c'd, there are two valid expansions of the same size: [a'c'd] and [a'b'd]



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

No, all expansions are not equally good. Some expansions may reach bigger terms than others. This occurs because after doing one expansion (where there were, say, two options to expand), the resulting term may combine with some already existing term. However, had we done the other expansion, it would not have combined further. Thus, all expansions are not equally good.

We try to choose the expansion based on the number of literals removed, taking the best possible expansion of a term.

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 checked if our implementation outputs the maximal expansion.
For all the test cases and their corresponding maximal terms, please see: testcases and solutions.
We have tested for big 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.
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'", "bc'd'", "a'c'd", "a'c'd", "a'b'd"]
   This case checks correctness when there are no dont 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 1 terms. There is no requirement to expand don't cares)

3. func_TRUE = ["a'b'c'd'", "a'b'c'd", "a'b'cd'", "a'b'cd","a'bc'd'", "a'bc'd", "a'bcd'", "a'bcd", "ab'c'd", "abc'd'", "abc'd", "abcd"]
   func_DC = []
   This covers the case when there are a large number of overlapping regions, and there are multiple options when expanding each term. We see from the final output that it gives the best reachable expansion for each. Here is the output:
   output = [" a' ", " a' ", " a' ", " a' ", " a' ", " a' ", " a' ", " a' ", " c'd "," bc' "," c'd ", " bd "]

4. func_TRUE = ["a'b'c'd'", "a'b'cd'", "a'b'cd","a'bc'd'", "a'bc'd", "a'bcd'", "ab'c'd", "abc'd'", "abc'd", "abcd"]
   func_DC=["a'bcd", "a'b'c'd"]
   This case is to check the working of high overlap and dont cares together. The output shows that our implementation can handle such cases correctly.

output = [" a' ", " a' ", " a' ", " a' ", " a' ", " a' ", " a' ", " a' ", " c'd "," bc' "," c'd ", " bd "]

5. 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'gh", "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'd'efg'h", "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'"]
   This is one of the big test cases that we tested on.
   output = ["abcdef'", 'bcdeg', "a'bc'f'h", "a'b'ef'g'", "bc'de'h", "a'b'ef'g'", "a'c'd'eh", "a'bdgh", "acd'fg'h", "a'b'ce'fg", "a'b'df'g'h'", "a'c'd'eh", "a'b'ce'fg", "ace'f'gh'", "acd'e'gh'", "b'd'e'fg'", "bc'd'g'h'", 'bcdeg', "a'c'd'eh", "a'b'ef'g'", "a'b'ef'g'", "bc'de'h", "bc'e'f'g'", "abcgh'", "acd'e'gh'", "bc'd'g'h'", 'cdefh', "a'c'de'fg", "acd'f'gh'", "ac'de'g'h", "bc'de'h", 'bcdeg', "abegh'", 'adefh', "abcgh'", "b'ceg'h", "ab'dfg'h'", "bc'de'h", "a'b'ce'fg", "a'bcdfg'h'", "bc'e'f'g'", 'bcdeg', "a'b'df'g'h'", "b'c'd'egh", "a'cdeg", "a'bd'e'f'g'", "b'c'd'egh", "a'bd'e'f'g'", "a'b'ef'g'", "b'd'e'fg'", "b'cdf'g'h", "a'b'def'h'", "b'c'dfg'", 'cdefh', "abegh'", "a'b'ef'g'", "b'c'dfg'", "a'b'ef'g'", "b'c'deg'h'", "a'c'd'f'h", "abc'eh'", 'bcdeg', "a'b'e'fgh'", "a'bdgh", "abdfg'h", "a'cdeg", 'cdefh', "a'bdgh", "ab'c'egh", 'cdefh', "bc'de'h", 'bdefh', "b'c'dfg'", "b'c'd'f'gh", "b'd'e'fg'"]
6. func_TRUE = [”a’bc’defgh”, ”abc’d’efgh”, ”a’bc’d’ef’gh”, ”abc’defgh”, ”a’bc’def’gh”, ”abc’d’ef’gh”, ”a’bc’d’efgh”, ”abc’def’gh”]

func DC = []
Output = ["bc'egh", "bc'egh", "bc'egh", "bc'egh", "bc'egh", "bc'egh", "bc'egh", "bc'egh"]

7. func_TRUE = ["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'"]
   func_DC = ["abc'd'e'", "abc'd'e", "abc'de", "abc'de'"]
   output = ["c'd'e'", "a'b'c'd'e'", "a'b'cde'", "bc'", "bc'", "bc'", "bc'", "c'd'e'", "ab'd'e'"]

8. func_TRUE = ["a'bc'd'", "abc'd'", "a'b'c'd", "a'bc'd", "a'b'cd"]
   func_DC = ["abc'd"]
   output = ["bc'", "bc'", "a'c'd", "bc'","a'b'd"]