SW Assignment-1

Aryan Gaurav 2020CS10327 Atul Jeph 2020CS10329

October 2022

1 Approach

Precisely, our approach was to first find the region corresponding to the term, then try to see whether it is a $full$ region or a $wrap-around$ and find the top left and bottom right coordinates respectively, and finally check legality of the term by checking value at each coordinate in the *kmap* region we found above.

2 Code

The **Code** consists of three parts, namely, **region finding**, **coordinate finding** and **legality check**. Since, the $legality - check$ needs to be generic for n variables, the $region - finding$ also has to be generic, only the $coordinate - finding$ part is hard-coded for n=2,3 and 4 variables.

We have defined two functions, one is the given **is-legal-region** function for $coordinate - finding$ and $legality - check$ and the other, **region**, for finding the region corresponding to the given *term*.

3 is-legal-region function

Firstly, we define an initial empty list of tuples (coordinates) *reglist* and then, fill it up with all the coordinate values of the *kmap*; this serves as the $universe$ of all coordinates possible for the given *kmap* function. This list is further converted into a set called *reg*. Then, we call the **region** function to get the region corresponding to the given *term*; this region is represented as a set of coordinates (tuples).

Now, we need to find the top-left and bottom-right coordinates, corresponding to the region found. We first find the **actual** top-left (min) and bottom-right (max) coordinates in the region *reg*. Then, we need to find whether the region is a $full$ / normal region or a $wrap-around$. For that, we define a variable $full$, which is equal to the total number of coordinates, had there been a normal/ **NOT** a wrap-around, square/rectangular region in the *kmap* matrix, starting at *min* and ending at *max* coordinate. We also have a variable *size* denoting the size of the set reg. Now, if full != size, it means we have wrap-around and we need to alter our originally found top-left and bottom-right coordinates.

Now, for full != size, we alter the top-left (min) and bottom-right (max) coordinates using a bunch of conditions for special cases and other conditions for generic cases, finally giving us the two tuples (coordinates) that we need to return. The conditions have been explained through $one - liner - comments$ in the code itself. It all boils down to swapping the x or y or both coordinates of the min and max tuples, under different conditions.

Finally, the **legality** check for the given $term$ is generic, i.e., could be extended to any number of variables. There, we simply check the values in the $kmap$ corresponding to the coordinates listed in the set reg; if any of them is 0, we return $false$, else $true$.

4 region function

The **region** function is generic over the length of the term, i.e., it would give the region for any (n) number of variables. We have followed the **gray code** for binary number increment.

We have taken 2 variables k1 and k2, corresponding to the number of column and row variables respectively; and *col* and *row* for number of columns and rows in the *kmap* matrix. The for loop runs for $k1 + k2$ iterations, first k1 for finding the variable and it's complement's regions, of the column variables and next k2 iterations for the row variables.

Here, for each iteration we define x which is the partition length corresponding to the *variable* under consideration (e.g- a,b,c,d,e,f,g etc). The value of a *variable* changes for the first time after $one - partition - length$ number of columns/rows and successive changes occur after every $2 * partition - length$ number of columns/rows. For example, in a 7-variable *kmap*, a changes after 8 columns, c changes for the first time after 2 columns, and successive changes occur after every 4 columns and so on. Similarly, for the row variables, the same concept applies. So, we make use of this concept to find the regions corresponding to a variable and it's complement and store it all in a dictionary of dictionaries *dict*.

Finally, we just traverse through the list *term* and for each $literal$, we take the intersection of the "so-far" achieved *region* with the region corresponding to the *literal* and store it in the variable reg. reg was initialized as the set of all possible coordinates for the given number of variables in the **is-legal-region** function.

A detailed description of the function is provided in the code itself, with the help of comments.

5 Test Cases

We used the $K - map - gui - tk$ utility for visualization and test-case checking purposes. For the test cases ($kmap - function$ and $term$), we displayed the corresponding region on the $kmap$ and cross-checked the marked region with the $term$ used as input. Some of the test-cases (in the form of ($kmap - function$, $term$)) used were:

- [[1,'x'],[0,1]],[None,None]
- [1,'x'],[0,1]],[1,1]
- [1,'x'],[0,1]],[0,None]
- [[1,'x',0,1],['x',1,0,'x']],[None,0,None]
- [[1,'x',0,1],['x',1,0,'x']],[None,None,None]
- [[1,'x',0,1],['x',1,0,'x']],[0,1,0]
- [[1,0,'x',0],[1,1,1,0],[1,'x','x',0],[1,0,1,0]],[None,0,None,0]
- [[1,0,'x',0],[1,1,1,0],[1,'x','x',0],[1,0,1,0]],[None,None,None,0]
- [[1,0,'x',0],[1,1,1,0],[1,'x','x',0],[1,0,1,0]],[1,1,0,0]
- [[1,0,'x',0],[1,1,1,0],[1,'x','x',0],[1,0,1,0]],[None,None,None,None]

And the corresponding outputs were:

- ((0, 0),(1, 1), False)
- ((1, 1),(1, 1), True)
- ((0, 0),(1, 0), False)
- ((0, 3),(1, 0), True)
- ((0, 0),(1, 3), False)
- ((0, 1),(0, 1), True)
- ((3, 3),(0, 0), False)
- ((3, 0),(0, 3), False)
- ((0, 2),(0, 2), True)
- ((0, 0),(3, 3), False)

6 Test Case for generic region find

Since we did not find the coordinates for $n > 4$, we weren't able to make use of the utility provided, therefore, we provided 5 and 7 variable kmap-functions and terms to manually check the correctness of our $region{-}find$ and $legality{-}check$ functions. It worked well.

Test Cases:

- [[1,0,1,1,'x','x',0,1], [1,0,1,1,'x','x',0,1], [1,0,1,1,'x','x',0,1], [1,0,1,1,'x','x',0,1]], [None,0,0,None,1]
- [[1,0,1,1,'x','x',0,1], [1,0,1,1,'x','x',0,1], [1,0,1,1,'x','x',0,1], [1,0,1,1,'x','x',0,1]], [None,1,None,None,0]
- [[1,0,0,1,1,1,1,0,'x',0,'x',1,'x',0,1,1],
  [1,0,0,1,1,1,1,0,'x',0,'x',1,'x',0,1,1],
  [1,0,0,1,1,1,1,0,'x',0,'x',1,'x',0,1,1],
  [1,0,0,1,1,1,1,0,'x',0,'x',1,'x',0,1,1],
  [1,0,0,1,1,1,1,0,'x',0,'x',1,'x',0,1,1],
  [1,0,0,1,1,1,1,0,'x',0,'x',1,'x',0,1,1],
  [1,0,0,1,1,1,1,0,'x',0,'x',1,'x',0,1,1],
  [1,0,0,1,1,1,1,0,'x',0,'x',1,'x',0,1,1]]
  , [None,None,None,0,None,None,0]

Output:

- [(1, 0), (1, 7), (2, 0), (2, 7)] , True
- [(0, 2), (0, 3), (0, 4), (0, 5), (3, 2), (3, 3), (3, 4), (3, 5)], True
- [(0, 0), (0, 3), (0, 4), (0, 7), (0, 8), (0, 11), (0, 12), (0, 15), (3, 0), (3, 3), (3, 4), (3, 7), (3, 8), (3, 11), (3, 12), (3, 15), (4, 0), (4, 3), (4, 4), (4, 7), (4, 8), (4, 11), (4, 12), (4, 15), (7, 0), (7, 3), (7, 4), (7, 7), (7, 8), (7, 11), (7, 12), (7, 15)] , False