이름: Dela Rosa, Joshua
학번: 2019142208

디지털논리회로 프로젝트
제출: 2020.12.19

Code Description

Objective: To explain how the KMAP solver is written and explain what algorithms and data structures were implemented to write the kmap.py file.

Language: Python

The minimum sum of products is written by using 3 classes namely: `Minterm, Group,` and `kmap`. `Minterm` and `Group` are both Node structures which contains useful information that will be utilized during the running of the kmap solver program.

Minterm Class

The class is a node type of data structure that holds important information which relates to the minterms entered in to the KMAP. It contains a `self.key`, which corresponds to the minterm value it represents; a `self.bcd`, which is the binary form of the number defined by the number of variables in the kmap. `self.split` allows access to the minterm by interpreting it as a list, which is made possible by overloading the `__getitem__` function. `self.selected` refers to the state if the minterm is selected and is part of a group; and self.isIncluded is used in determining essential prime implicants.

```
3  class Minterm:
4      def __init__(self, key): # input needs to be an integer
5          self.key = int(key)
6          self.bcd = None
7          self.split = None
8          self.selected = False
9          self.isIncluded = False
10
```

Group Class

The class is also a node type of data structure that refers to the prime implicants in the k-map. This class contains `self.members` which contains the minterms gathered by this group; `self.tier`, which is the number of minterms it has; `self.split`, allows the access to the data carried by the group; and `self.dominated`, refers if the group is a subset of another bigger group. The group class has an `addMinterm()` function which allows to add minterms to the group.

```
26 class Group:
27     def __init__(self, key): # input key is str
28         self.members = []
29         self.tier = 0
30         self.key = str(key)
31         self.split = [i for i in key]
32         self.dominated = False
```

kmap Class

The class contains the methods to solve for 3 variables or 4 variable maps. The appropriate map size, variables, and assignments are determined by the `variables` argument. Under the class, it contains useful characteristics such as `self.nodes`, which contains the noded version of the minterms; `self.groups`, which contains all of the groups that is currently processed to solve for the minimum sum of products; and `self.bcd`, which determines what binary assignments to be assigned to each minterms. The `self.variables`, will determine how the kmap will run. Depending whether the variables are 3 or 4, the map size and bcd assignments will differ. At the very beginning, the appropriate bcd assignments are assigned to each minterm and `self.solve()` is automatically run at initiation.

```
51  class kmap:
52      def __init__(self, variables, minterms):
53          self.variables = int(variables)
54          self.minterms = minterms # this is a list
55          self.nodes = [] # minterm nodes
56          self.groups = [] # group nodes
57          self.bcd = {}
```

The following methods under the kmap class are explained in detail as follows:

- `solve ()`:

Depending on the number of variables, the way the kmap run is different. However, the same algorithm is used on both occasions.

First, the `self.groups` is reset to avoid any duplication from a previous input. A dictionary of all the possible combinations are then defined via a dictionary. The dictionary has keys that refers to the prime implicants for all possible inputs and the item assigned with it is the node form of that group. Using a for loop, all of the prime implicants of the k-map for the given input is assigned to manually and added to the specific group it belongs in the dictionary.

```
223          for minterm in self.nodes:
224              a = minterm[0]
225              b = minterm[1]
226              c = minterm[2]
227              v1k_a = str(a) + "**"
228              v1k_b = "*" + str(b) + "*"
229              v1k_c = "**" + str(c)
230              v2k_ab = str(a) + str(b) + "*"
231              v2k_ac = str(a) + "*" + str(c)
232              v2k_bc = "*" + str(b) + str(c)
233              v1 = [v1k_a, v1k_b,v1k_c]
234              v2 = [v2k_ab, v2k_ac, v2k_bc]
235
236              for v1_str in v1:
237                  groupdict[v1_str].addMinterm(minterm)
238              for v2_str in v2:
239                  groupdict[v2_str].addMinterm(minterm)
```

Next, the most dominant groups are searched and added into the `self.groups`. 'Dominant' only refers to prime implicants with variable assignments, not if all of the minterms are present and minimum output is 1. The group is only added to the list if and only if it is in full capacity, which depends on the number of variables. For 3 variables, the biggest is a 4 member group and for 4 variables, an 8 member group is the most dominant; both are represented only by 1 variable. The reason being is that if a most dominant group exist, it will dominate over the smaller prime implicants. Additionally, it is immediately added to the list as it is definitely going to be an essential prime implicant. All other groups are temporarily held in another list for further processing.

If there exists a 1 variable group in the `self.groups` list, all of the items held temporarily are examined whether it is dominated by the 1 variable group. This is done by examining the key of the groups. For instance, X_1 has a key of "1∗∗∗". X_1X_2 with the key "11∗∗", is dominated by X_1 as it is the subset of the latter. This is confirmed because the first digits are both the same and the number of asterisks are less than X_1. The subsets are also set as dominated. For the 4 variable kmap, the same process is done twice: (1) for the 1 variable 8 member group; and (1) for the 2 variable 4 member group.

```
251         # see if 2 vars are dominated:
252         limit_list = ["0**", "1**","*0*", "*1*","**0", "**1"]
253         for limit in limit_list:
254             if groupdict[limit] in self.groups:
255                 for code in groupdict[limit].split:
256                     if code != "*":
257                         ind = groupdict[limit].split.index(code)
258                         for i in hold:
259                             if i.split[ind] != groupdict[limit].split[ind]:
260                                 self.groups.append(i)
261                             if i.split[ind] == groupdict[limit].split[ind]:
262                                 i.dominated = True
```

After determining the subsets, all of the subsets that are not dominated by any other group is added to the `self.groups` list. Then, all of the groups that are on their own are then converted to groups and added to the list, after all of the comparisons. This is to help realize a more systematic approach on solving the kmap.

After all of the prime implicants are determined, essential prime implicants are determined by using the `self.isIncluded` characteristic of the Minterm node. To begin, every group is compared to all of the other groups. If the group's members is not subset of all of the other groups' members, it is considered as an essential prime implicant. This is because it holds 1 or more minterms that are not covered by any other group in the list. There would be instances where there may be multiple essential prime implicant combinations. To do properly but randomly select the proper prime implicants, a for loop is run for all of the nodes in the minterms list. If a minterm is not yet included in any of the essential prime implicants, the a group is extracted from the list of overlapping prime implicants. This picked prime implicant's members are all set to included. This prevents picking a new prime implicant that includes the same minterm unless there are no other options but to pick it again.

For cases where all of the minterms of the k-map are present, the code resets the group list and appends a group with key “1” that includes every minterm and appends the group; if empty, a group with “0” is appended instead.

- Other functions

The kmap class is packed with all of the other methods that are helpful such as printing the table for visualization as well as for printing the Minimal Sum of Products. All of the other functions included are automatically set to run after calling the kmap class assuming that the number of variables and minterms are defined.