Course project
EC444: Synthesis and Optimization of Digital Circuits

Submitted by

Adithya Jayan | 181EC102

Under the Guidance of

Prof. M S Bhat

Department of Electronics and Communication Engineering,

NITK Surathkal

Date of Submission: 2-12-2021

National Institute of Technology Karnataka,

Surathkal

Contents

1 Introduction

The aim of the project was as follows:

1. Find the cofactors of the function with respect to any given variable or a set of variables (a, ab ab' etc) and display the results.
2. For a given variable ordering, find and display the ROBDD of the function

The project implementation was carried out using the *python3* language.

2 Implementation: Task 1

Find the cofactors of the function with respect to any given variable or a set of variables (a, ab ab' etc) and display the results.

The implementation was broken down into parts. A *Parse_String* class was created to hold and represent the Boolean function. Other functions such as for generating cofactors, were defined inside the class and is called using the objects of this class. The functions defined in the class are described below.

2.1 The Parse_String constructor

This function initializes a *Parse_String* object and stores the expression in the required format for easy use.
Function call example:

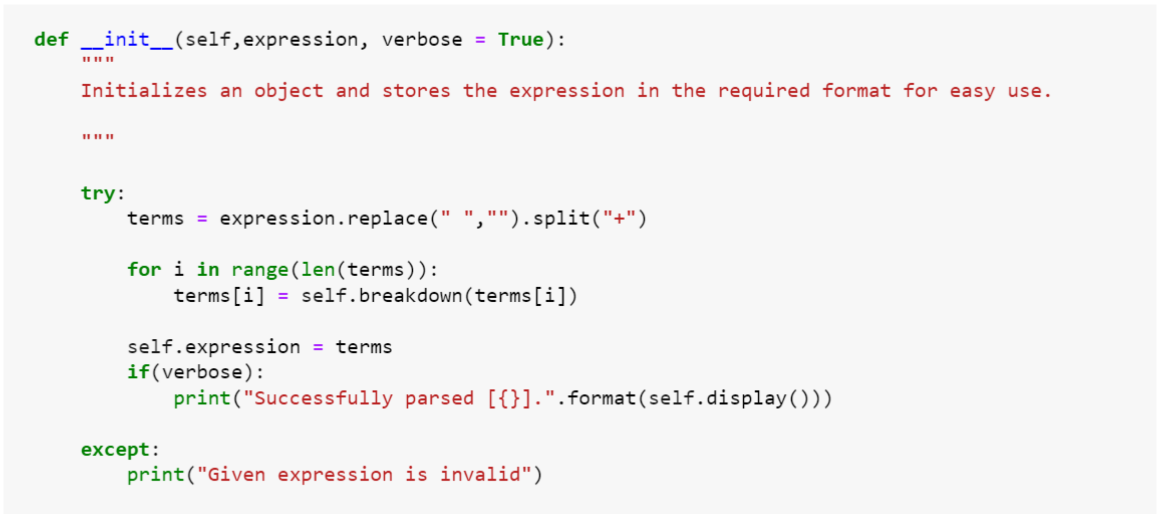
```
SoP = "AC + A'BC' + AB' + DAB' + AEFG' + G + EC" #Input string given by the user
Boolean_expression = Parse_String(SoP) #Create Boolean expression variable
```

The constructor takes the Boolean function input as a string. This string is parsed and stored in a more easy to format. It makes use of the *breakdown* function to break each cube (in string format) into a list of individual literals. The final output is a list of cubes, where each cube is itself a list of literals i.e., an object containing a list of lists.
Therefore,

```
SoP = "AC + A'BC' + AB' + DAB' + AEFG' + G + EC"
```

will get converted to

```
[[A,C],
[A’,B,C’],
[A,B’],
[D,A,B’],
[A,E,F,G’],
[G],
[E,C]]
```

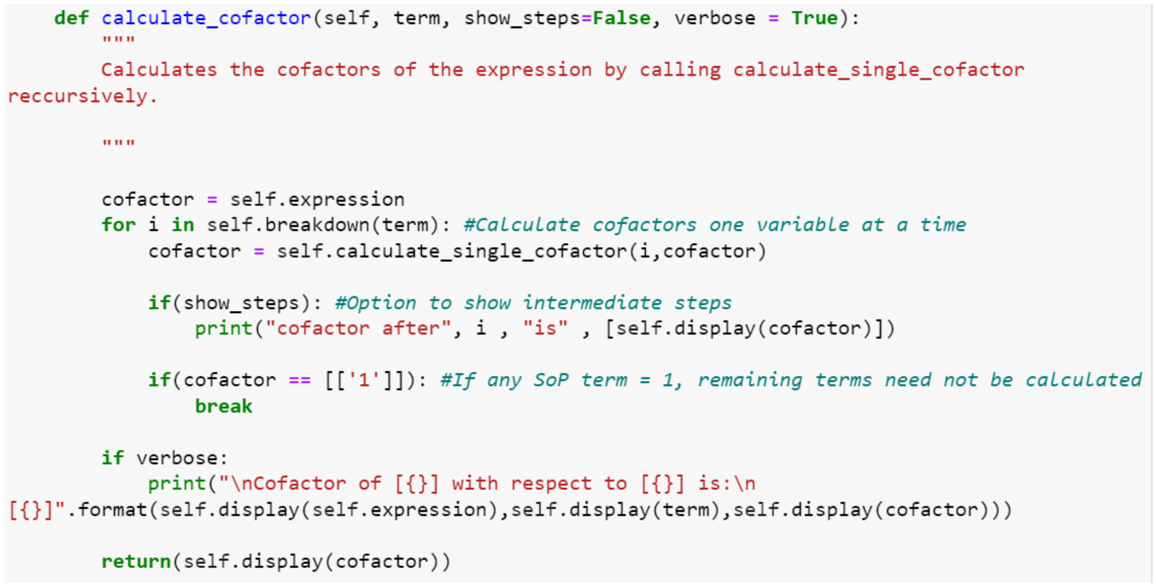


```python
def __init__(self,expression, verbose = True):
    """
    Initializes an object and stores the expression in the required format for easy use.

    """

    try:
        terms = expression.replace(" ","").split("+")

        for i in range(len(terms)):
            terms[i] = self.breakdown(terms[i])

        self.expression = terms
        if(verbose):
            print("Successfully parsed [{}].".format(self.display()))

    except:
        print("Given expression is invalid")
```

Figure 2.1: The Parse_String constructor definition

2.2 The calculate_cofactor function

Calculates the cofactors of the expression by calling *calculate_single_cofactor* reccursively.



```python
    def calculate_cofactor(self, term, show_steps=False, verbose = True):
        """
        Calculates the cofactors of the expression by calling calculate_single_cofactor
reccursively.

        """

        cofactor = self.expression
        for i in self.breakdown(term): #Calculate cofactors one variable at a time
            cofactor = self.calculate_single_cofactor(i,cofactor)

            if(show_steps): #Option to show intermediate steps
                print("cofactor after", i , "is" , [self.display(cofactor)])

            if(cofactor == [['1']]): #If any SoP term = 1, remaining terms need not be calculated
                break

        if verbose:
            print("\nCofactor of [{}] with respect to [{}] is:\n
[{}]".format(self.display(self.expression),self.display(term),self.display(cofactor)))

        return(self.display(cofactor))
```

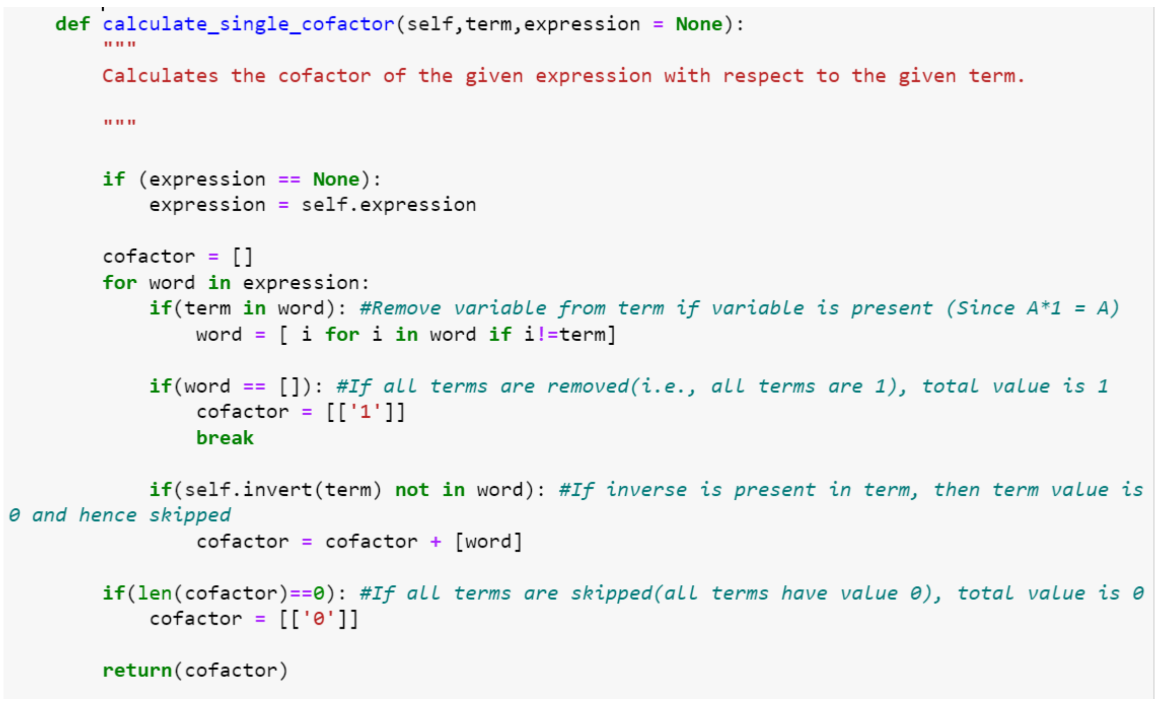
Figure 2.2: The calculate cofactor function

If we wish to calculate cofactor with respect to 'AC', then it first calculates cofactor with respect to A, and then with respect to C by calling *calculate_single_cofactor* each time.

2.3 The calculate_single_cofactor function

This function calculates the cofactor of the given expression with respect to a single given literal. It is used by the *calculate_cofactor* function. It does the following to calculate the cofactor:

- Drops the literal from cube if the literal is present in the cube.
- If the inverse of the literal is present in the cube, then cube value is 0. Therefore the entire cube is dropped.
- If there are no literals remaining in a cube (After the dropping step), it means the cube is 1 and hence the cofactor itself becomes 1.



```
    def calculate_single_cofactor(self,term,expression = None):
        """
        Calculates the cofactor of the given expression with respect to the given term.

        """

        if (expression == None):
            expression = self.expression

        cofactor = []
        for word in expression:
            if(term in word): #Remove variable from term if variable is present (Since A*1 = A)
                word = [ i for i in word if i!=term]

            if(word == []): #If all terms are removed(i.e., all terms are 1), total value is 1
                cofactor = [['1']]
                break

            if(self.invert(term) not in word): #If inverse is present in term, then term value is
0 and hence skipped
                cofactor = cofactor + [word]

        if(len(cofactor)==0): #If all terms are skipped(all terms have value 0), total value is 0
            cofactor = [['0']]

        return(cofactor)
```

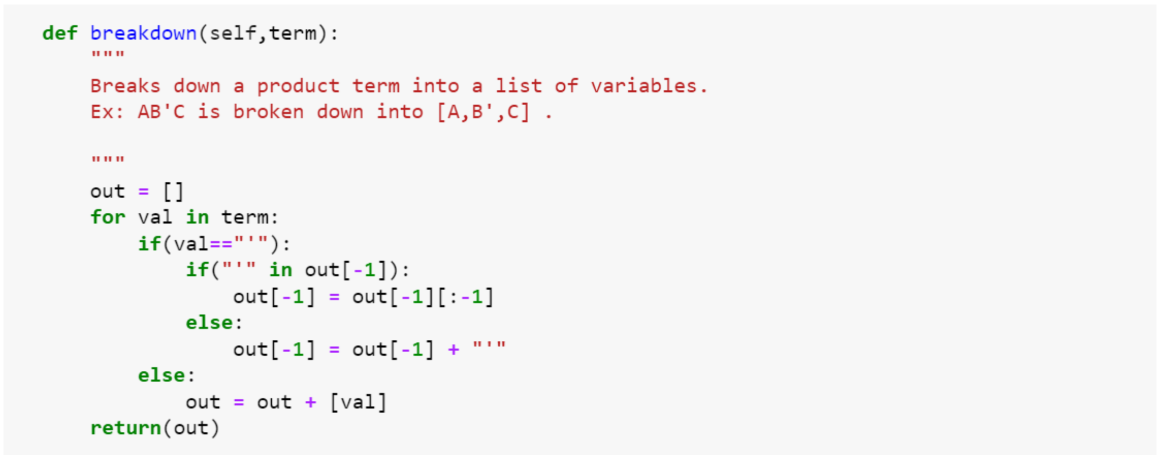
Figure 2.3: The calculate single cofactor function

2.4 Other utility functions

2.4.1 The breakdown function

Breaks down a product term into a list of variables. It is used by the *parse_string* constructor.

`Ex: AB’C is broken down into [A,B’,C] .`



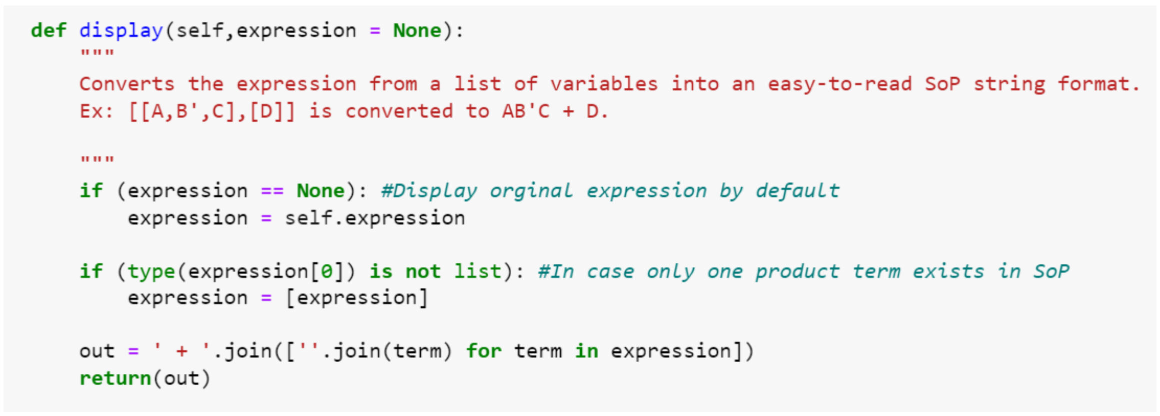
```
def breakdown(self,term):
    """
    Breaks down a product term into a list of variables.
    Ex: AB'C is broken down into [A,B',C] .

    """
    out = []
    for val in term:
        if(val=="'"):
            if("'" in out[-1]):
                out[-1] = out[-1][:-1]
            else:
                out[-1] = out[-1] + "'"
        else:
            out = out + [val]
    return(out)
```

Figure 2.4: The breakdown function used by the constructor

2.4.2 The display function

Converts the expression from a list of cubes (Which is itself a list of literals) into an easy-to-read SoP string format. It is used to display expressions in a readable way.

`Ex: [[A,B’,C],[D]] is converted to AB’C + D.`



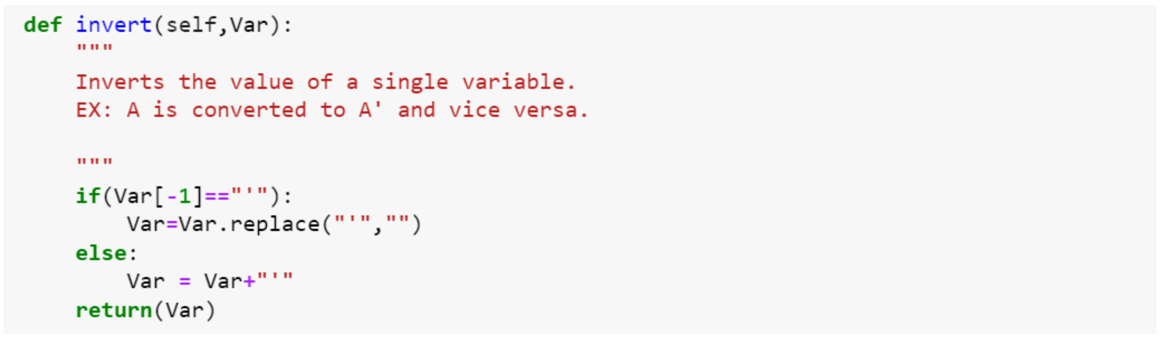
```
def display(self,expression = None):
    """
    Converts the expression from a list of variables into an easy-to-read SoP string format.
    Ex: [[A,B',C],[D]] is converted to AB'C + D.

    """
    if (expression == None): #Display orginal expression by default
        expression = self.expression

    if (type(expression[0]) is not list): #In case only one product term exists in SoP
        expression = [expression]

    out = ' + '.join([''.join(term) for term in expression])
    return(out)
```

Figure 2.5: The display function

2.4.3 The invert function

Inverts the value of a single literal.

`EX: A is converted to A’ and vice versa.`

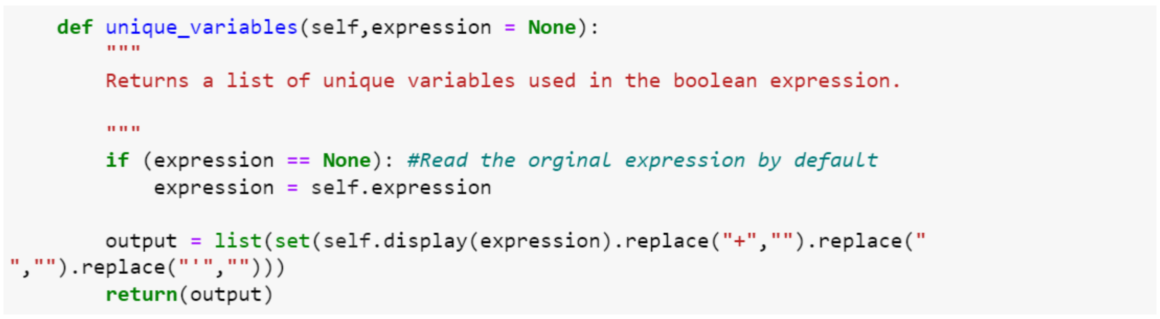


```
def invert(self,Var):
    """
    Inverts the value of a single variable.
    EX: A is converted to A' and vice versa.

    """
    if(Var[-1]=="'"):
        Var=Var.replace("'","")
    else:
        Var = Var+"'"
    return(Var)
```

Figure 2.6: The invert function

2.4.4 The unique_variables function

Returns a list of unique variables used in the Boolean expression. This function is used during the construction of the reduced ordered binary decision diagram of the Boolean expression.

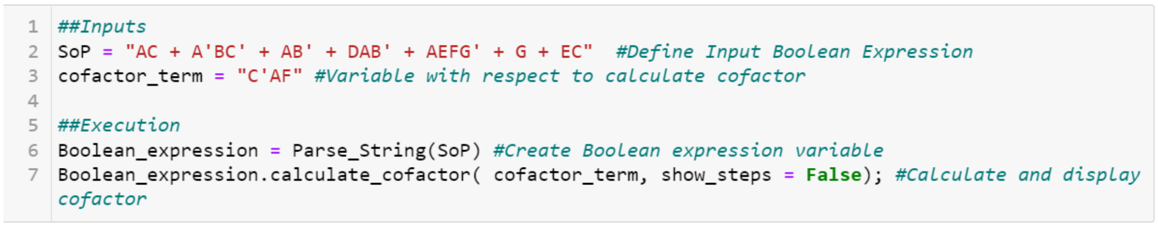


```
    def unique_variables(self,expression = None):
        """
        Returns a list of unique variables used in the boolean expression.

        """
        if (expression == None): #Read the orginal expression by default
            expression = self.expression

        output = list(set(self.display(expression).replace("+","").replace("
","").replace("'","")))
        return(output)
```

Figure 2.7: The unique_variables function

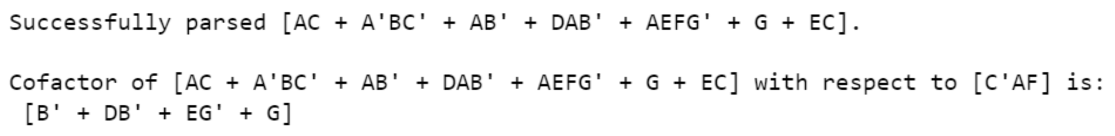
2.5 Final Result

The final function call and outputs are given in figures 2.8 and 2.9.



```
##Inputs
SoP = "AC + A'BC' + AB' + DAB' + AEFG' + G + EC"  #Define Input Boolean Expression
cofactor_term = "C'AF" #Variable with respect to calculate cofactor

##Execution
Boolean_expression = Parse_String(SoP) #Create Boolean expression variable
Boolean_expression.calculate_cofactor( cofactor_term, show_steps = False); #Calculate and display cofactor
```

Figure 2.8: Task 1 Code



```
Successfully parsed [AC + A'BC' + AB' + DAB' + AEFG' + G + EC].

Cofactor of [AC + A'BC' + AB' + DAB' + AEFG' + G + EC] with respect to [C'AF] is:
 [B' + DB' + EG' + G]
```

Figure 2.9: Task 1 output

3 Implementation: Task 2

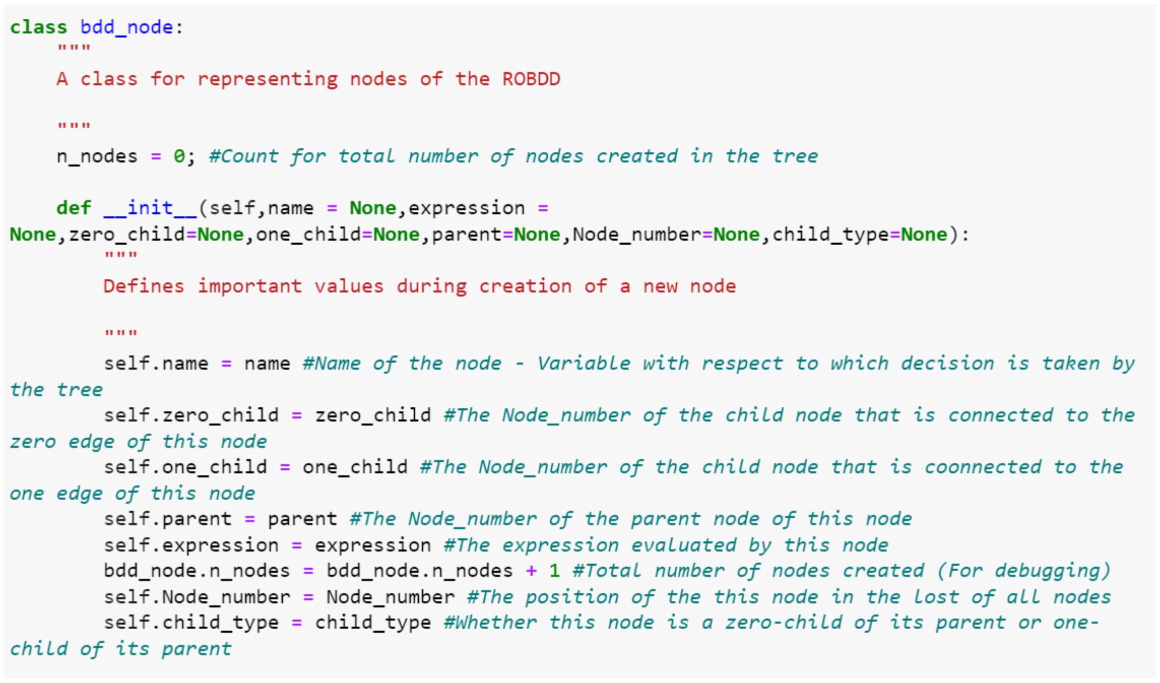
For a given variable ordering, find and display the ROBDD of the function.

The ROBDD is implemented using the *Generate_ROBDD* function. This function uses the *bdd_node* class to define each node. Each node contains all the data related to the node such as name, level, children and parents bundled together into one object.

Unlike languages such as C or C++, **Python does not support the use of pointers**. Thus representing relations between parents and children becomes more complex. For this reason, I have attempted to 'replicate' the functionality of pointers by storing all the nodes in a list, and using it's position value in the list as the pointer to it. [This can be implemented more efficiently in other languages by using actual pointers instead]

3.1 The bdd_node class

A class for representing nodes of the ROBDD.



```
class bdd_node:
    """
    A class for representing nodes of the ROBDD

    """
    n_nodes = 0; #Count for total number of nodes created in the tree

    def __init__(self,name = None,expression =
None,zero_child=None,one_child=None,parent=None,Node_number=None,child_type=None):
        """
        Defines important values during creation of a new node

        """
        self.name = name #Name of the node - Variable with respect to which decision is taken by
the tree
        self.zero_child = zero_child #The Node_number of the child node that is connected to the
zero edge of this node
        self.one_child = one_child #The Node_number of the child node that is coonnected to the
one edge of this node
        self.parent = parent #The Node_number of the parent node of this node
        self.expression = expression #The expression evaluated by this node
        bdd_node.n_nodes = bdd_node.n_nodes + 1 #Total number of nodes created (For debugging)
        self.Node_number = Node_number #The position of the this node in the lost of all nodes
        self.child_type = child_type #Whether this node is a zero-child of its parent or one-
child of its parent
```

Figure 3.1: Constructor for the BDD Node class

This class allows us to store all the required values pertaining to a node such as:

- Name of the node - Variable with respect to which decision is taken by the tree

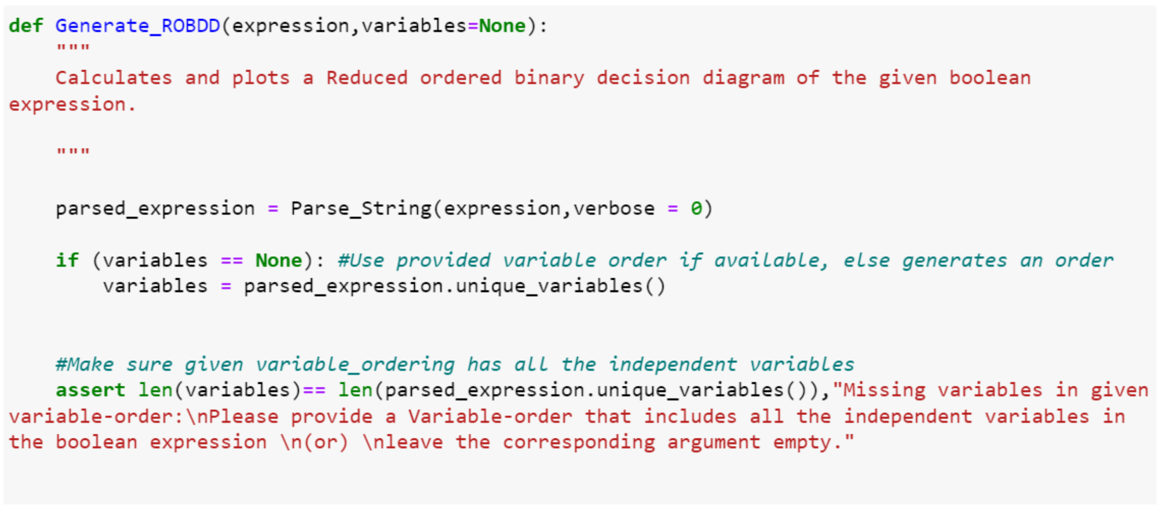
- Children: The Node_numbers(i.e pointers) of the child node that is connected to the zero and one edge of this node
- Parent: The Node_number(i.e pointer) of the parent node of this node
- Evaluated sub-expression: The expression evaluated by this node
- Pointer value: The position of the this node in the list of all nodes (i.e, pointer to itself)
- Child-type: Whether this node is a zero-child of its parent or one-child of its parent

3.2 The Generate_ROBDD function

This function handles the entire calculation and plotting of the ROBDD using the provided expression. It can be broken down into parts for easy understanding.

3.2.1 Reading and validating inputs

Here, the input Boolean expression and variable ordering is read. The expression string is then parsed into a easy-to-use format using the *Parse_String* class defined in Task 1.



```
def Generate_ROBDD(expression,variables=None):
    """
    Calculates and plots a Reduced ordered binary decision diagram of the given boolean
expression.

    """

    parsed_expression = Parse_String(expression,verbose = 0)

    if (variables == None): #Use provided variable order if available, else generates an order
        variables = parsed_expression.unique_variables()


    #Make sure given variable_ordering has all the independent variables
    assert len(variables)== len(parsed_expression.unique_variables()),"Missing variables in given
variable-order:\nPlease provide a Variable-order that includes all the independent variables in
the boolean expression \n(or) \nleave the corresponding argument empty."
```

Figure 3.2: Input reading and validation (ROBDD part 1)

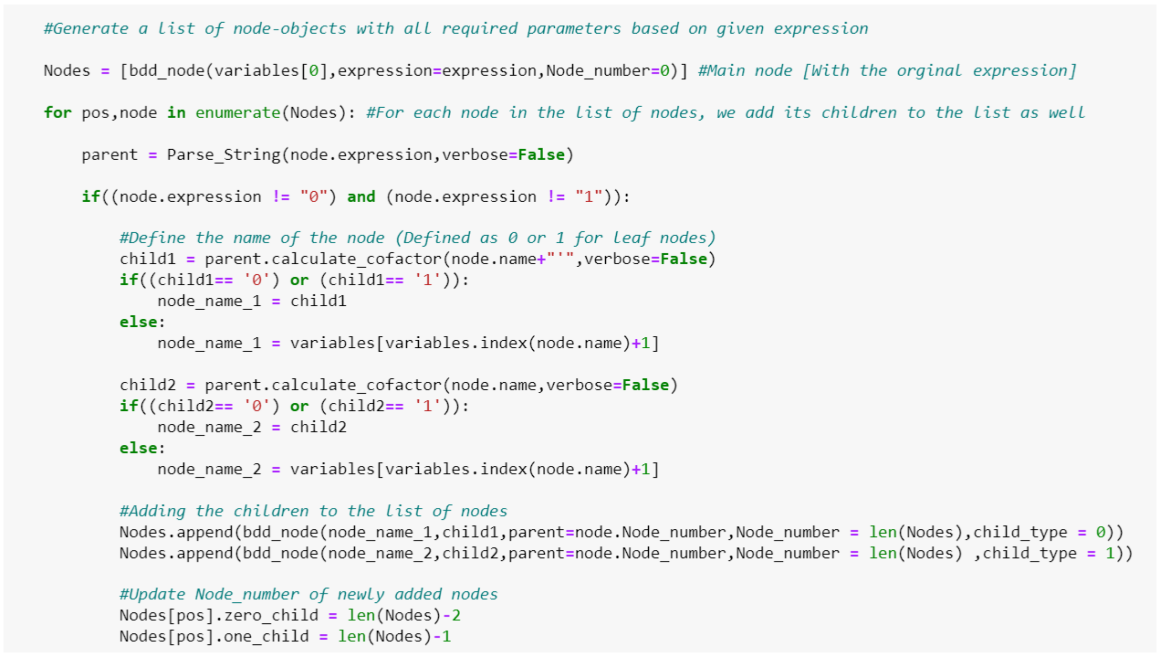
The *unique_variables* function (Also defined earlier) is used to verify if all the literals present in the expression are given in the 'variable order' sting. If not, an error is thrown.

If an input variable order is not given, then a random order is generated using the *unique_variables* function.

3.2.2 Creating list of nodes

We create a list of nodes from the expression recursively. This is done so we can use the list to emulate 'pointers' using the position of the node in the list, since **Python does not support pointers**. This is effectively a non-reduced ordered-BDD.

The values of the nodes such as parents and children are also set during this step. The nodes are calculated by taking the cofactor of the expression with respect to the terms in the given/generated variable order using the previously defined *calculate_cofactor* function.



```python
#Generate a list of node-objects with all required parameters based on given expression

Nodes = [bdd_node(variables[0],expression=expression,Node_number=0)] #Main node [With the orginal expression]

for pos,node in enumerate(Nodes): #For each node in the list of nodes, we add its children to the list as well

    parent = Parse_String(node.expression,verbose=False)

    if((node.expression != "0") and (node.expression != "1")):

        #Define the name of the node (Defined as 0 or 1 for leaf nodes)
        child1 = parent.calculate_cofactor(node.name+"'",verbose=False)
        if((child1== '0') or (child1== '1')):
            node_name_1 = child1
        else:
            node_name_1 = variables[variables.index(node.name)+1]

        child2 = parent.calculate_cofactor(node.name,verbose=False)
        if((child2== '0') or (child2== '1')):
            node_name_2 = child2
        else:
            node_name_2 = variables[variables.index(node.name)+1]

        #Adding the children to the list of nodes
        Nodes.append(bdd_node(node_name_1,child1,parent=node.Node_number,Node_number = len(Nodes),child_type = 0))
        Nodes.append(bdd_node(node_name_2,child2,parent=node.Node_number,Node_number = len(Nodes) ,child_type = 1))

        #Update Node_number of newly added nodes
        Nodes[pos].zero_child = len(Nodes)-2
        Nodes[pos].one_child = len(Nodes)-1
```

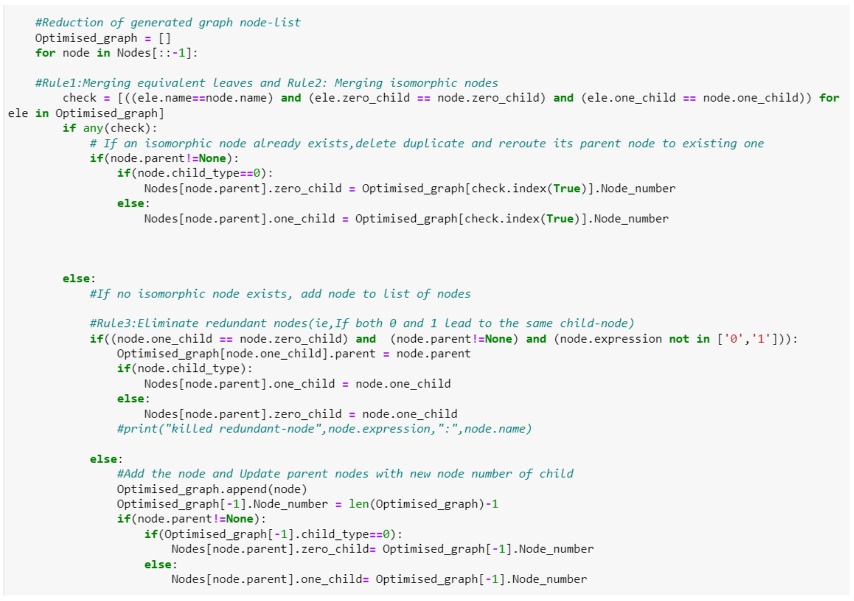
Figure 3.3: Generating list of nodes (ROBDD part 2)

3.2.3 Reducing the BDD nodes

The three rules of reduction are carried out in an optimized manner to obtain the Reduced Ordered Binary Decision Diagram.

- Rule1: Merging equivalent leaves
- Rule2: Merging isomorphic nodes
- Rule3: Eliminate redundant nodes (i.e. If both 0 and 1 lead to the same child-node)

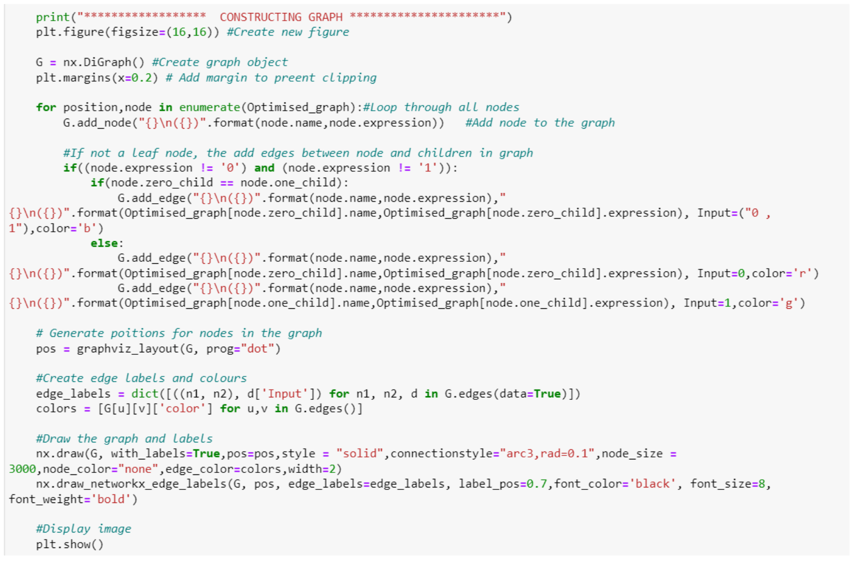
Due to the optimization, only a single pass is required to generate the entire ROBDD.



```python
    #Reduction of generated graph node-list
    Optimised_graph = []
    for node in Nodes[::-1]:

    #Rule1:Merging equivalent leaves and Rule2: Merging isomorphic nodes
        check = [((ele.name==node.name) and (ele.zero_child == node.zero_child) and (ele.one_child == node.one_child)) for 
ele in Optimised_graph]
        if any(check):
            # If an isomorphic node already exists,delete duplicate and reroute its parent node to existing one
            if(node.parent!=None):
                if(node.child_type==0):
                    Nodes[node.parent].zero_child = Optimised_graph[check.index(True)].Node_number
                else:
                    Nodes[node.parent].one_child = Optimised_graph[check.index(True)].Node_number



        else:
            #If no isomorphic node exists, add node to list of nodes

            #Rule3:Eliminate redundant nodes(ie,If both 0 and 1 lead to the same child-node)
            if((node.one_child == node.zero_child) and  (node.parent!=None) and (node.expression not in ['0','1'])):
                Optimised_graph[node.one_child].parent = node.parent
                if(node.child_type):
                    Nodes[node.parent].one_child = node.one_child
                else:
                    Nodes[node.parent].zero_child = node.one_child
                #print("killed redundant-node",node.expression,":",node.name)

            else:
                #Add the node and Update parent nodes with new node number of child
                Optimised_graph.append(node)
                Optimised_graph[-1].Node_number = len(Optimised_graph)-1
                if(node.parent!=None):
                    if(Optimised_graph[-1].child_type==0):
                        Nodes[node.parent].zero_child= Optimised_graph[-1].Node_number
                    else:
                        Nodes[node.parent].one_child= Optimised_graph[-1].Node_number
```

Figure 3.4: Reducing the generated BDD (ROBDD part 3)

3.2.4 Displaying the generated ROBDD



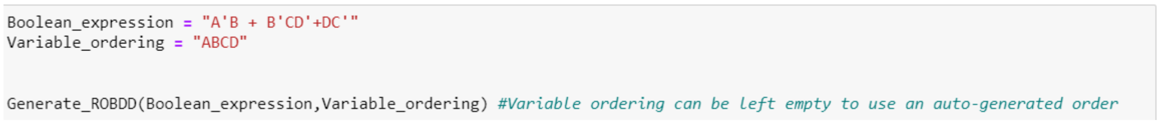
```python
    print("******************  CONSTRUCTING GRAPH **********************")
    plt.figure(figsize=(16,16)) #Create new figure

    G = nx.DiGraph() #Create graph object
    plt.margins(x=0.2) # Add margin to preent clipping

    for position,node in enumerate(Optimised_graph):#Loop through all nodes
        G.add_node("{}\n({})".format(node.name,node.expression))   #Add node to the graph

        #If not a leaf node, the add edges between node and children in graph
        if((node.expression != '0') and (node.expression != '1')):
            if(node.zero_child == node.one_child):
                G.add_edge("{}\n({})".format(node.name,node.expression),"
{}\n({})".format(Optimised_graph[node.zero_child].name,Optimised_graph[node.zero_child].expression), Input=("0 ,
1"),color='b')
            else:
                G.add_edge("{}\n({})".format(node.name,node.expression),"
{}\n({})".format(Optimised_graph[node.zero_child].name,Optimised_graph[node.zero_child].expression), Input=0,color='r')
                G.add_edge("{}\n({})".format(node.name,node.expression),"
{}\n({})".format(Optimised_graph[node.one_child].name,Optimised_graph[node.one_child].expression), Input=1,color='g')

    # Generate poitions for nodes in the graph
    pos = graphviz_layout(G, prog="dot")

    #Create edge labels and colours
    edge_labels = dict([((n1, n2), d['Input']) for n1, n2, d in G.edges(data=True)])
    colors = [G[u][v]['color'] for u,v in G.edges()]

    #Draw the graph and labels
    nx.draw(G, with_labels=True,pos=pos,style = "solid",connectionstyle="arc3,rad=0.1",node_size = 
3000,node_color="none",edge_color=colors,width=2)
    nx.draw_networkx_edge_labels(G, pos, edge_labels=edge_labels, label_pos=0.7,font_color='black', font_size=8, 
font_weight='bold')

    #Display image
    plt.show()
```

Figure 3.5: Displaying the ROBDD (ROBDD part 4)

The generated ROBDD is then parsed and displayed in a graphical form using the matplotlib, networkx and graphviz libraries. Zero and one edges are color coded, and appropriate node labels and expressions are also added.

3.3 Final Result

The final function call and generated result is given in figures 3.6 and 3.7.



```
Boolean_expression = "A'B + B'CD'+DC'"
Variable_ordering = "ABCD"


Generate_ROBDD(Boolean_expression,Variable_ordering) #Variable ordering can be left empty to use an auto-generated order
```

Figure 3.6: Task 2 Code

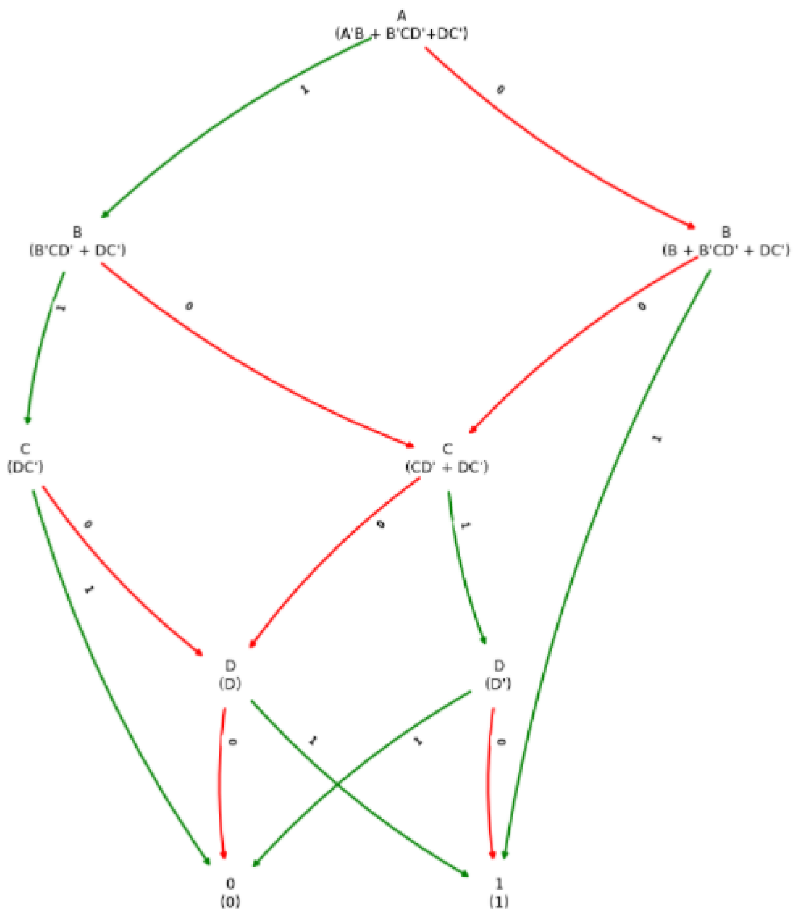
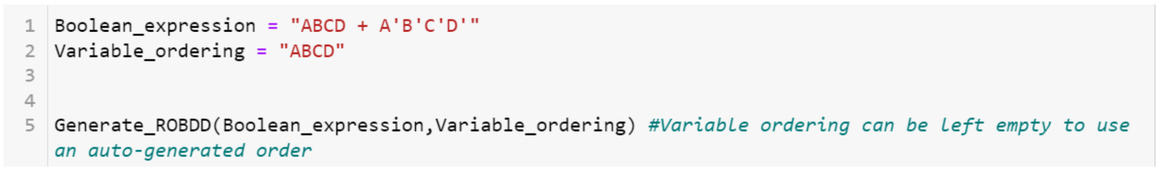


Figure 3.7: Task 2 output (Final ROBDD diagram)

Some more examples and results are given below.



```
Boolean_expression = "ABCD + A'B'C'D'"
Variable_ordering = "ABCD"


Generate_ROBDD(Boolean_expression,Variable_ordering) #Variable ordering can be left empty to use
an auto-generated order
```

Figure 3.8: Task 2 Code (Example 2)

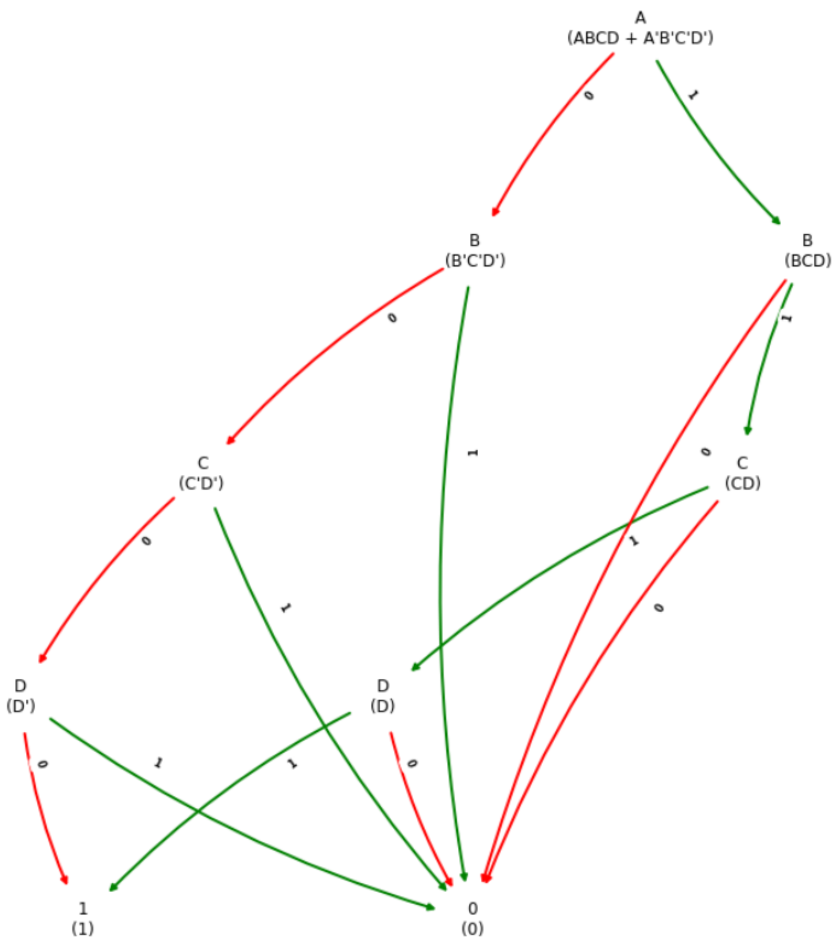
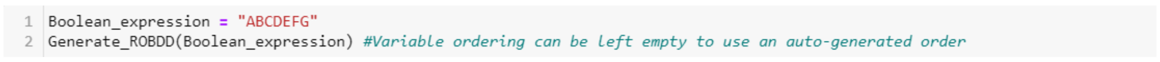


Figure 3.9: Task 2 output (Final ROBDD diagram) (Example 2)

An example with auto generated variable order.



```
Boolean_expression = "ABCDEFG"
Generate_ROBDD(Boolean_expression) #Variable ordering can be left empty to use an auto-generated order
```

Figure 3.10: Task 2 Code (Example 3)

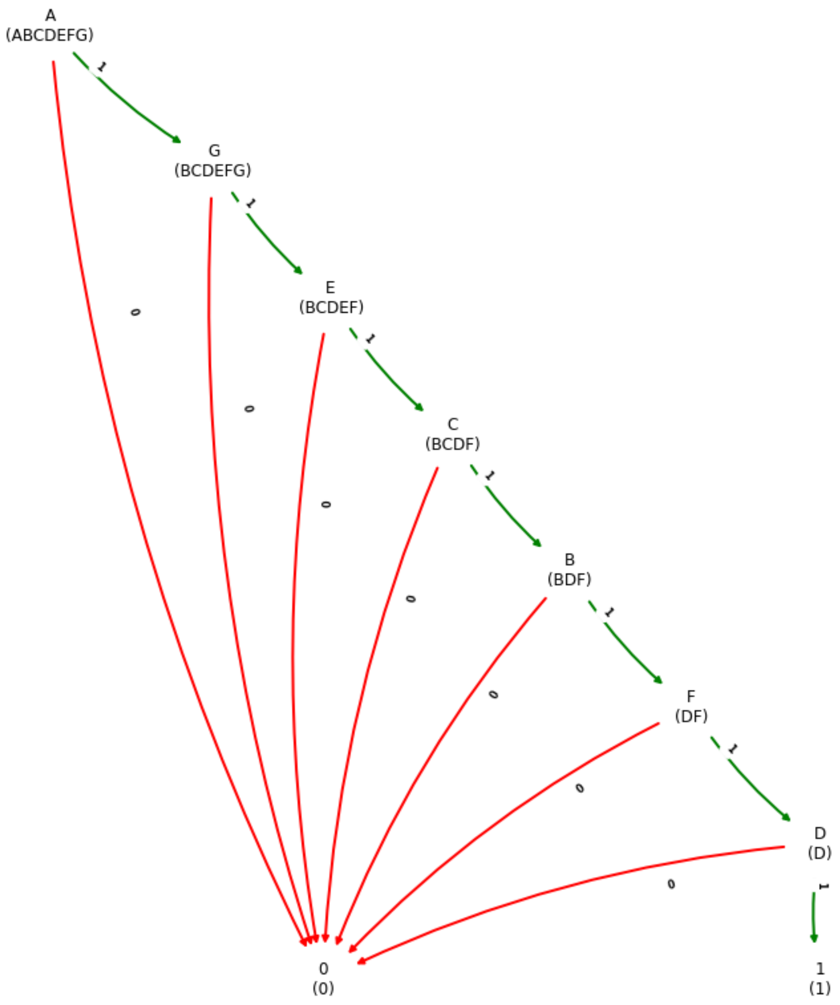


Figure 3.11: Task 2 output (Final ROBDD diagram) (Example 3)