ROBDD CHECKER

Table of Contents

A. Introduction:

Binary Decision Diagrams (BDDs) are data structures used to represent and manipulate Boolean functions efficiently. They provide a graphical way to model logical expressions, where each node represents a variable, and edges represent the possible values (true or false) of that variable. Reduced Ordered Binary Decision Diagrams (ROBDDs) are an optimized form of BDDs that ensure a canonical representation of a Boolean function by applying two key restrictions: variables appear in a fixed order along any path, and redundant nodes or duplicate subgraphs are eliminated. This reduction and ordering make ROBDDs highly compact and efficient, particularly useful in applications such as digital circuit design, verification, and synthesis in Electronic Design Automation (EDA).

B. Gui Input:

Introduction:

The Graphical User Interface (GUI) designed for this project serves as a user-friendly tool to visualize and analyze Boolean functions. The GUI facilitates the input of Boolean functions and their variable orderings, providing a simple yet interactive experience for users to observe the corresponding Binary Decision Diagram (BDD) and its optimized form, the ROBDD.

Additionally, it enables users to perform equivalence checking between Boolean functions, ensuring accuracy and reliability in logical comparisons. The intuitive layout and clear visualizations make it accessible for both educational purposes and practical applications in digital logic design.

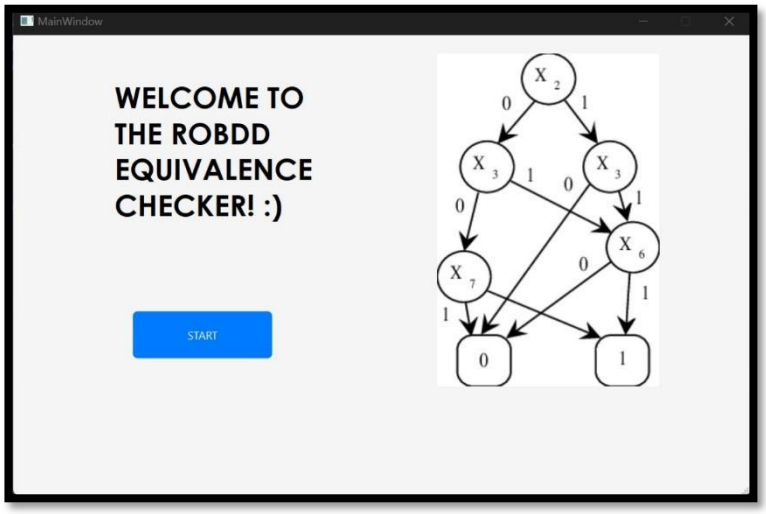
Key Features:

1-Input

2-Error Handling

3-Output

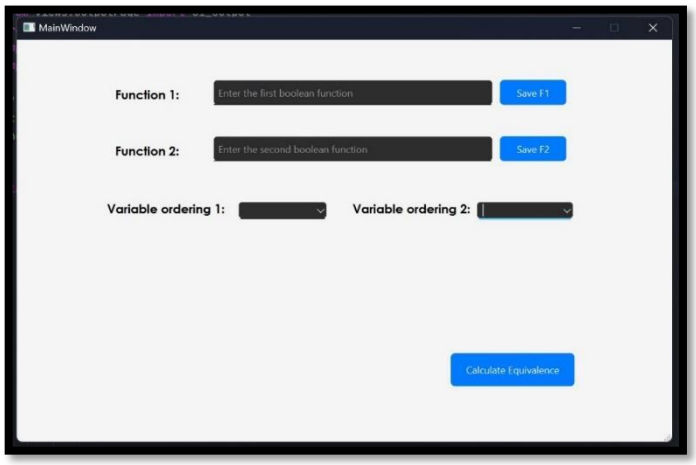
Firstly, when we run the Application, the first page that appears is the welcome page as shown in the next photo:



this is just a simple starting page that welcomes the user. When the user clicked on the "Start" button , it switches the page and routes to the input screen where we will discuss all the key features.

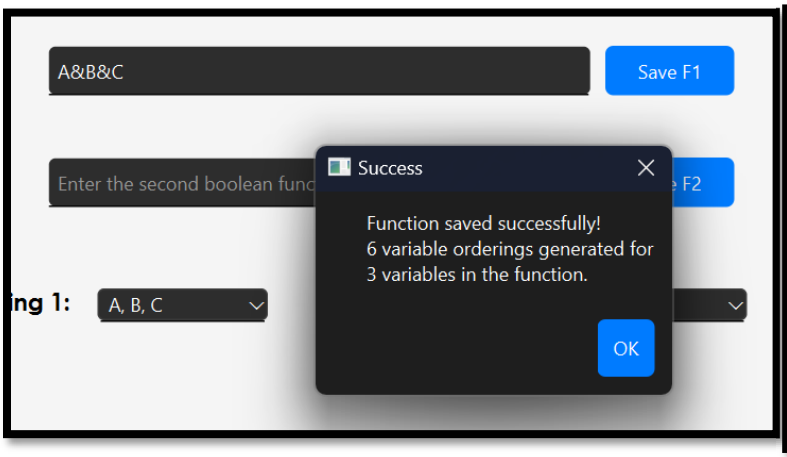
1- Input:

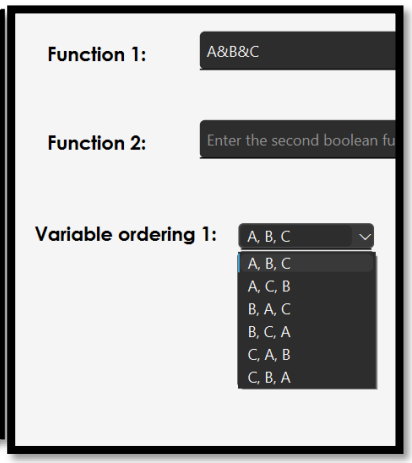
The GUI provides the facility for the user to input to functions in the shown text fields with labels "Function 1:" and "Function 2:" , and also their variable orderings. When the user clicks on the save button, the function is saved and all the existing



variables in it is stored in a set.

Then all the possible variable orderings are stored in the combo-box for the respected function as shown in the following images:





Code Section:

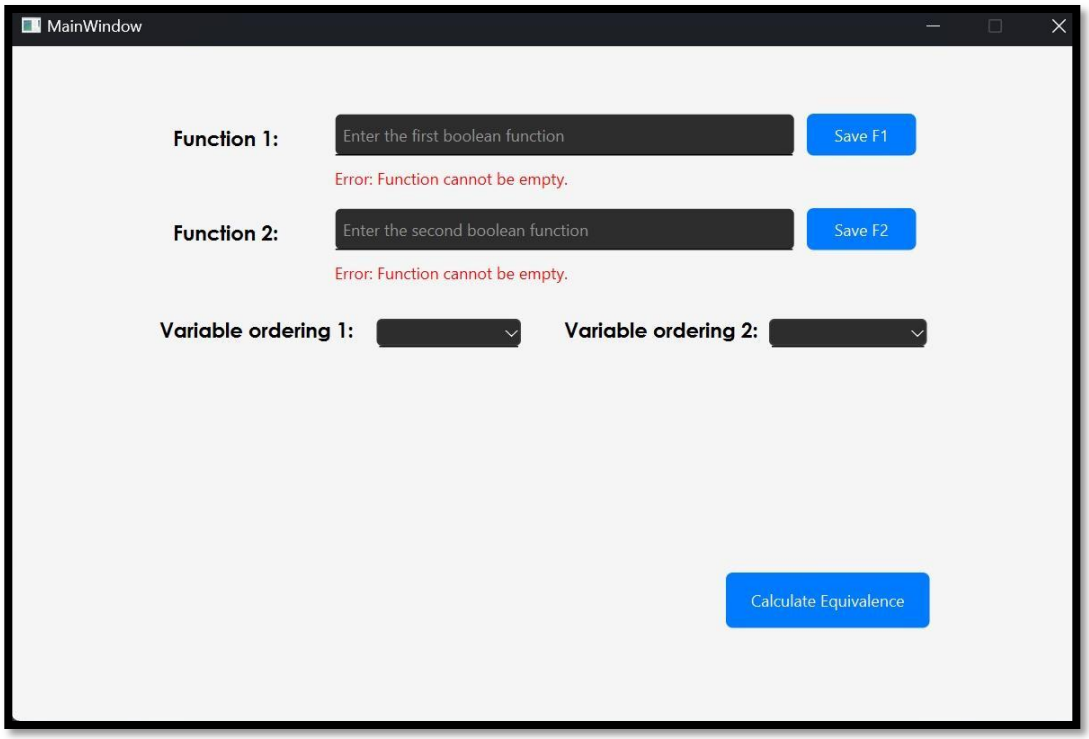
```
# Extract unique variables from the function text
variables = sorted(set(re.findall( pattern: r'[A-Za-z]+', function_text)))

# Generate all permutations of variables (variable orderings)
orderings = list(itertools.permutations(variables))

# Clear the combo box and insert new items
combo_box.clear()
for ordering in orderings:
    combo_box.addItem(", ".join(ordering))

# Optional: Display a message box indicating success
success_box = QMessageBox(self)
success_box.setWindowTitle("Success")
success_box.setText(f"Function saved successfully!\n{len(orderings)} variable orderings generated for \n{len(variables)} variables in the function.")
success_box.setStyleSheet("QLabel{color: white;}")
success_box.exec()
```

2-Error Handling:

In this section we edited the GUI to handle multiple errors after trying various scenarios:

2.1-Empty Fields:

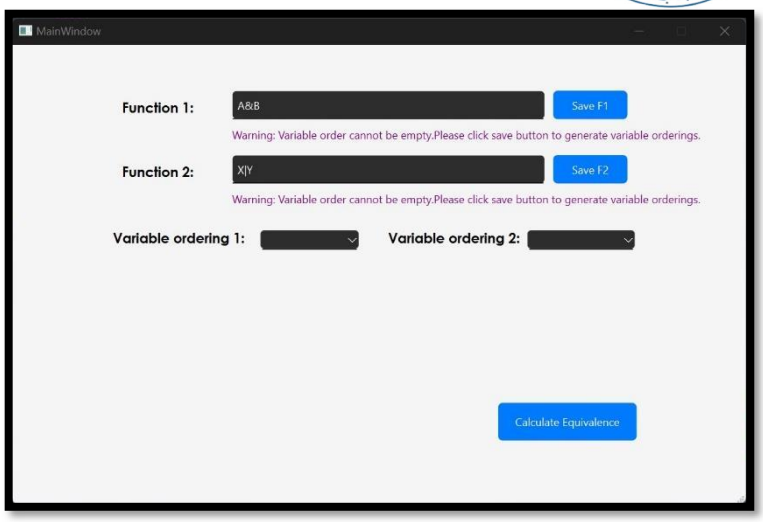
Let's suppose the user clicked on the "Calculate Equivalence" button but the fields does not hold any text. Therefore, we created a check on the fields where we check if any of them are empty then display an error message as shown:



-Code section:

2.2-Function Not Saved:

In order for the variable ordering to be generated, the “Save” button must be clicked. Therefore , we implemented the feature of checking on the current text in the variable orderings combo box if its empty and the user entered any text in the function field. If this check returns true then an error message is displayed asking the user to save the function.



-Code section:

```
if self.ui.Func1.text() == "" or self.ui.Func2.text() == "":
    if self.ui.Func1.text() == "":
        self.ui.Error1.setText("Error: Function cannot be empty.")
        self.ui.Error1.setStyleSheet("color: red;")
        self.ui.Error1.show()

        # Clear the line edit
        self.ui.Func1.clear()

    if self.ui.Func2.text() == "":
        self.ui.Error2.setText("Error: Function cannot be empty.")
        self.ui.Error2.setStyleSheet("color: red;")
        self.ui.Error2.show()

        # Clear the line edit
        self.ui.Func2.clear()
    return
```

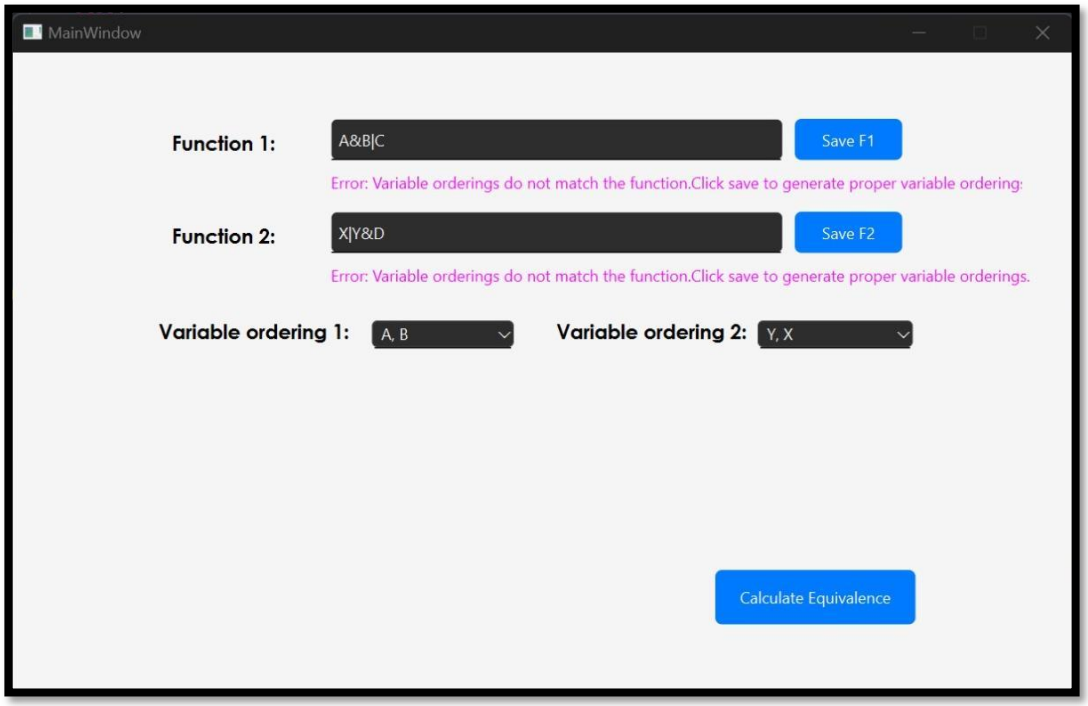
```
elif self.ui.Vorder1.currentText() == "" or self.ui.Vorder2.currentText() == "":

    if self.ui.Vorder1.currentText() == "":
        self.ui.Error1.setText("Warning: Variable order cannot be empty.Please click save button to generate variable orderings.")
        self.ui.Error1.setStyleSheet("color: purple;")
        self.ui.Error1.show()

    if self.ui.Vorder2.currentText() == "":
        self.ui.Error2.setText("Warning: Variable order cannot be empty.Please click save button to generate variable orderings.")
        self.ui.Error2.setStyleSheet("color: purple;")
        self.ui.Error2.show()
    return
```

2.3-Function-Variables Mismatch:

Let's say that after the user entered a function and its ordering, the user then modified the function and added more variables into it. Well that would be a problem because he will be entering a variable order that is different from the function itself. That is why this check was implemented, to avoid code complications in the run.

Here is the shown error handling:

-



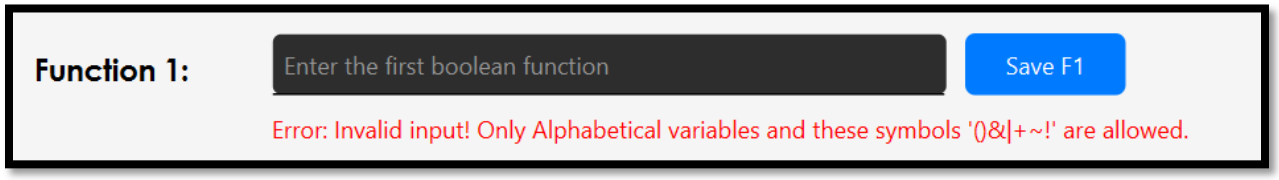
Code Section:

```
if variables_in_Func1 != variables_in_Vorder1 or variables_in_Func2 != variables_in_Vorder2:
    if variables_in_Func1 != variables_in_Vorder1:
        self.ui.Error1.setText("Error: Variable orderings do not match the function.Click save to generate proper variable orderings.")
        self.ui.Error1.setStyleSheet("color: magenta;")
        self.ui.Error1.show()
    if variables_in_Func2 != variables_in_Vorder2:
        self.ui.Error2.setText("Error: Variable orderings do not match the function.Click save to generate proper variable orderings.")
        self.ui.Error2.setStyleSheet("color: magenta;")
        self.ui.Error2.show()
    return
```

2.4-Invalid Operators:

In this type of error , we handle input from user that is not logically correct for the algorithm. For clarity, there is a set of accepted operators that the user cannot enter anything outside its bounds.So, an error message is displayed for the user asking to enter only valid characters and operators.

Then:



-Code section:

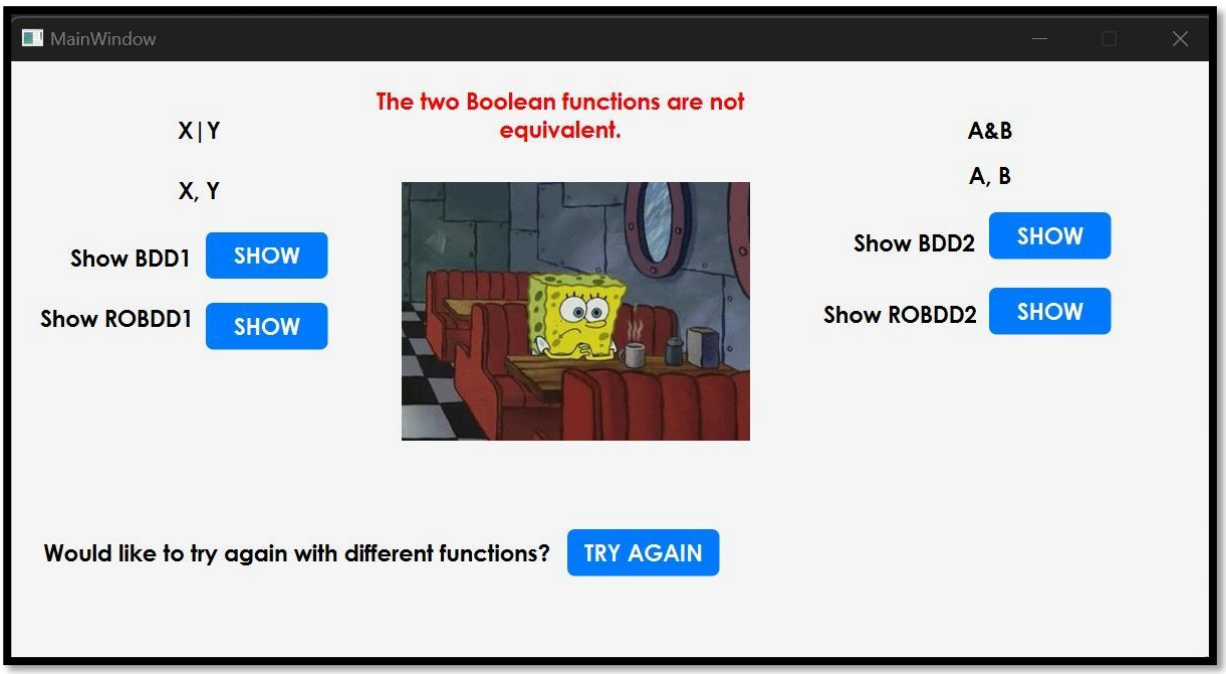
```
def save_function(self, line_edit, combo_box, error_label):  2 usages
    """Save function input, generate variable orderings, and display errors if needed."""
    # Get text from the line edit
    function_text = line_edit.text()

    # Validate the function input
    if not re.match( pattern: r'^[A-Za-z()&|+~!]+$', function_text):
        # Display the error message
        error_label.setText("Error: Invalid input! Only Alphabetical variables and these symbols '()&|+~!' are allowed.")
        error_label.setStyleSheet("color: red;")
        error_label.show()

        # Clear the line edit
        line_edit.clear()
        return
    else:
        # Clear error label if the input is valid
        error_label.clear()
```

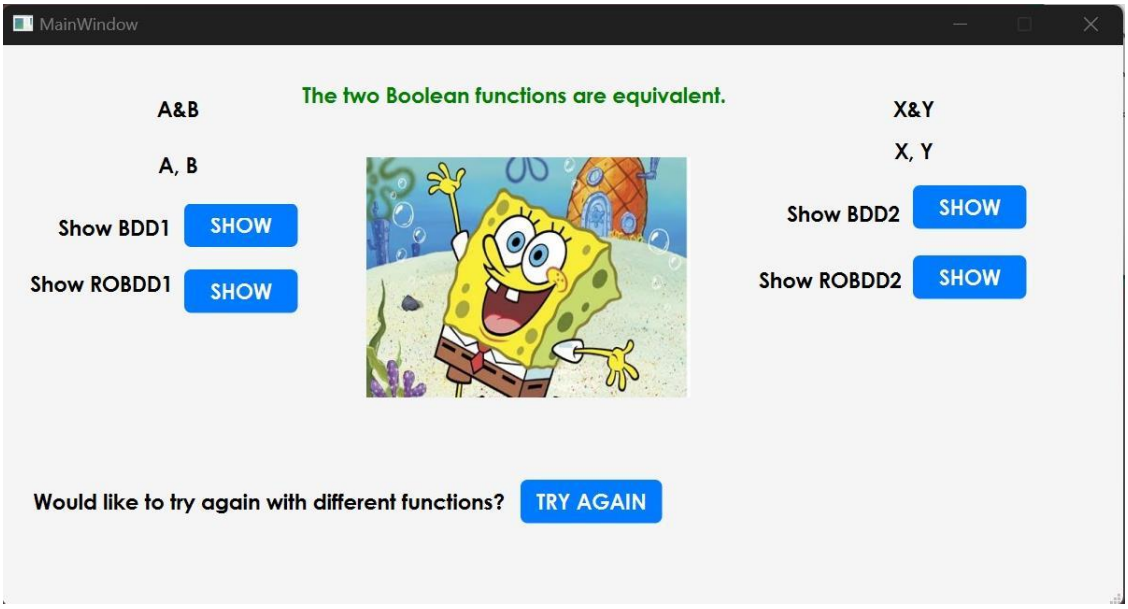
3-Output:

3.1 Showing Equivalence:

Okay, after the user enters the functions and their orderings and these inputs pass all the erros checks, the input page switches to the output page where the result of the equivalence between the functions is shown and their BDD and ROBDD graphs.

As shown here:



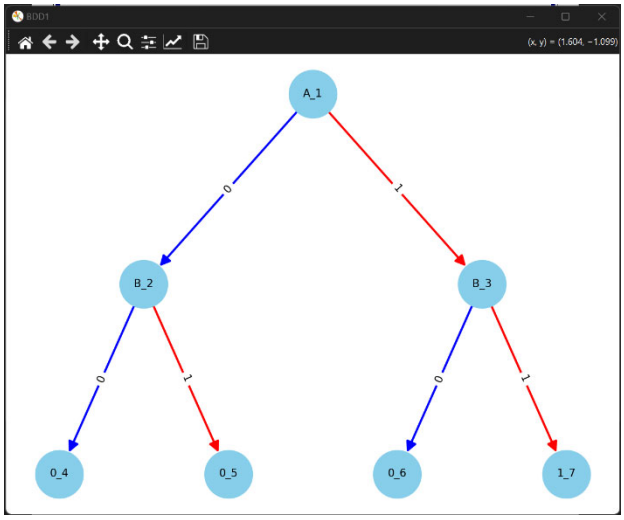


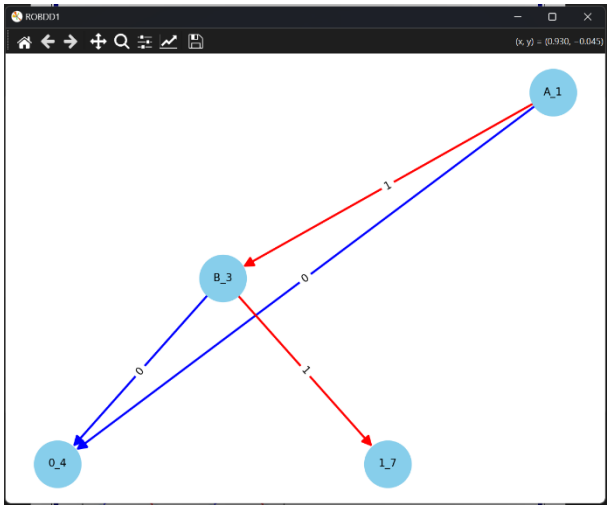
As a humorous matter, we decided it would be a nice thing to indicate the equivalence of the 2 function by using 2 images of “Spongebob Squarepants” along with the color changing sentence shown in the images :).

We also allowed the user to return to the input page if he/she wishes to enter another set of data by clicking the “Try Again” button.

3.2 Showing Graphs:

The buttons provided are connected to the visualizer function to create a plot showing the BDD and the ROBDD graphs of the function as follows:





-Code Section:

```
happy = "Images/happy_spongebob.png"
sad = "Images/sad_spongebob.png"
self.ui.Function1.setText(expr1)
self.ui.Function2.setText(expr2)
self.ui.Vorder1.setText(vorder1)
self.ui.Vorder2.setText(vorder2)
self.ui.equiv.setText("The two Boolean functions are equivalent." if result == "True" else "The two Boolean functions are not equivalent.")
if result == "True":
    self.ui.equiv.setStyleSheet("color: green;")
    self.ui.spongebob.setPixmap(QPixmap(happy))
    print("the 2 boolean functions are equivalent")
else:
    self.ui.equiv.setStyleSheet("color: red;")
    self.ui.spongebob.setPixmap(QPixmap(sad))
    print("the 2 boolean functions are NOT equivalent")
self.ui.TryAgain.clicked.connect(self.open_second_window)
self.ui.ShowBDD1.clicked.connect(lambda: self.visualise(BDD1, figname: "BDD1"))
self.ui.ShowBDD2.clicked.connect(lambda: self.visualise(BDD2, figname: "BDD2"))
self.ui.ShowROBDD1.clicked.connect(lambda: self.visualise(ROBDD1,  figname: "ROBDD1"))
self.ui.ShowROBDD2.clicked.connect(lambda: self.visualise(ROBDD2, figname: "ROBDD2"))
```

C. ***Parser Function:***

The Boolean Function Parser is a key component of the project. It evaluates a given Boolean expression and generates the truth table required for constructing the Binary Decision Diagram (BDD). This process includes evaluating the expression for all possible input combinations and organizing the results into terminal nodes for the BDD.

The parser is responsible for:

1. Translating a Boolean expression into a truth table.
2. Evaluating the expression based on all possible combinations of truth values for the variables.
3. Generating terminal nodes that form the leaves of the BDD.

This ensures that the BDD construction process starts with a solid foundation of correct results.

Key Functions

1. **evaluate_expression(expression, var_values)**

   This function evaluates the Boolean expression for a given set of truth values.

   - It replaces variables in the expression with their corresponding truth values (1 for True, 0 for False).
   - Logical operators are translated into Python syntax:
     - & → and
     - | → or
     - ! or ~ → not
   - Python's eval() function computes the result.

Example:

- **Input**: Expression: A & B | C, Variable Values: {'A': 1, 'B': 0, 'C': 1}
- **Output**: 1
-

2. **build_truth_tree(expression, var_order)**

   This function generates a truth table for the Boolean expression.

   - It calculates all possible combinations of truth values for the given variables using the itertools.product() function.
   - For each combination, the function evaluates the Boolean expression using evaluate_expression().
   - The results are stored as a list of terminal nodes, which are later used as leaves in the BDD.

   Example:

   - **Input**: Expression: A & B | C, Variables: ['A', 'B', 'C']
   - **Output**: [0, 1, 0, 1, 0, 1, 1, 1] (truth table).

Flow of the Parser

1. **Input**:
   - Boolean expression: e.g., A & B | C.
   - List of variables in the desired order: e.g., ['A', 'B', 'C'].
2. **Processing**:
   - Generate all possible truth value combinations for the variables.
   - Replace variables and operators in the expression with their respective truth values and Python logic.
   - Compute the results for each combination.
3. **Output**:
   - A truth table representing the behavior of the Boolean expression.
   - Example: For A & B | C, the truth table is [0, 1, 0, 1, 0, 1, 1, 1].

Impact on BDD Construction

The truth table produced by the parser provides the terminal nodes of the BDD. Each row in the truth table corresponds to a leaf in the BDD. Without an accurate parser, the BDD would fail to correctly represent the Boolean function.

Code Overview

1. **evaluate_expression**:

```python
def evaluate_expression(expression, var_values):
    """Evaluates a boolean expression given variable values."""
    expr_with_values = ''.join(
        [str(int(var_values.get(char, False))) if char.isalpha() else char for char in expression])
    expr_with_values = (
        expr_with_values.replace('!', 'not ').replace('&', ' and ').replace('+', ' or ').replace('~',  'not ').
        replace(  '|', ' or '))

    try:
        result = eval(expr_with_values)
        return '1' if result else '0'
    except Exception as e:
        print(f"Error evaluating expression: {e}")
        return None
```

build_truth_tree:

```python
def build_truth_tree(expression, var_order):
    """Builds a binary tree with terminal nodes as results of the expression."""
    num_vars = len(var_order)
    num_rows = 2 ** num_vars
    all_combinations = list(itertools.product([0, 1], repeat=num_vars))

    # Create the terminal leaves from left to right (2^n to 2^(n-1) values)
    terminals = []
    for values in all_combinations:
        var_values = dict(zip(var_order, [bool(val) for val in values]))
        result = evaluate_expression(expression, var_values)
        terminals.append(result)

    return terminals

def printTree(Seifos, n):
    for i in range (0, 2 ** (n+1)):
        print(f"Node number{i}: {Seifos[i]}")
```

- **Expression**: A & B | C
- **Variables**: ['A', 'B', 'C']
- **Truth Table**:
- A | B | C || Result
- 0 | 0 | 0 || 0
- 0 | 0 | 1 || 1
- 0 | 1 | 0 || 0
- 0 | 1 | 1 || 1
- 1 | 0 | 0 || 0
- 1 | 0 | 1 || 1
- 1 | 1 | 0 || 1
- 1 | 1 | 1 || 1
- **Parser Output**: [0, 1, 0, 1, 0, 1, 1, 1]

3. GetTree(truthTable, n, var_order) - Constructing the Binary Decision Diagram (BDD)

The getTree function is responsible for transforming a truth table into a binary tree structure, forming the foundation of the Binary Decision Diagram (BDD). It systematically organizes variables and outputs into nodes, creating a hierarchical representation of the Boolean function.

The purpose of the getTree function is to:

1. Construct a binary tree that represents the BDD structure.
2. Organize nodes for each variable in a specific order (variable ordering).
3. Append terminal nodes (leaves) representing the outputs of the Boolean function based on the truth table.

How It Works

1. **Initialize the Tree**:
   - The root node is added as a placeholder: ["/", "/", "/", "/"].
   - The first variable in the order is assigned to the first level of the tree, splitting into two children.
2. **Build Internal Nodes**:
   - For each level, starting from the second variable to the last:

- Nodes are connected to their parent and assigned indices for left and right children.
- Each node stores the variable corresponding to that level.

3. **Add Terminal Nodes**:
   - Terminal nodes are added at the last level of the tree.
   - Each terminal node contains:
     - The truth table result for a specific combination of variable values.
     - Its parent index.
     - No children ('/' placeholders).
4. **Output**:
   - The function prints the tree structure for debugging or validation.
   - Returns a list (Seifos) containing all nodes in the tree, including relationships between variables and truth values.

Each node is represented as:

- **Internal Node**: [Variable, Parent Index, Left Child Index, Right Child Index]
- **Terminal Node**: [Truth Value, Parent Index, '/', '/']

Input:

- Truth Table: [0, 1, 0, 1, 0, 1, 1, 1]
- Number of Variables: 3
- Variable Order: ['A', 'B', 'C']

Output (Partial Tree):

```
Node 0: ["/", "/", "/", "/"]          # Root
Node 1: ["A", "/", "2", "3"]           # Level 1 (Variable A)
Node 2: ["B", "1", "4", "5"]           # Level 2 (Variable B)
Node 3: ["B", "1", "6", "7"]
Node 4: ["C", "2", "8", "9"]           # Level 3 (Variable C)
Node 8: [0, "4", "/", "/"]            # Terminal Node
Node 9: [1, "4", "/", "/"]            # Terminal Node
```

This tree represents the initial, unoptimized form of a BDD:

- Each path from the root to a terminal node corresponds to a specific assignment of variable values.
- Terminal nodes represent the output of the Boolean function for those assignments.
- The function provides a structured foundation for further BDD optimizations, such as variable reordering and node reduction.

By organizing the Boolean function into a hierarchical structure, the getTree function enables efficient analysis and representation of logical relationships, forming a crucial step in the BDD construction process.

Code Overview

1. **GetTree(truthTable, n, var_order)**

```
47  def getTree(truthTable, n, var_order):
48      Seifos = []
49
50      # Add the root node
51      Seifos.append(["/", "/", "/", "/"])  # Root (placeholder)
52      Seifos.append([var_order[0], "/", "2", "3"])  # First level (root split)
53
54      # Build the tree level by level
55      for k in range(1, n):
56          start_idx = 2 ** k  # Starting index for level k
57          end_idx = 2 ** (k + 1)  # Ending index for level k (exclusive)
58
59          for i in range(start_idx, end_idx):
60              parent_idx = i // 2  # Calculate parent index
61              left_child_idx = 2 * i
62              right_child_idx = 2 * i + 1
63
64              # Append the node for the current level
65              Seifos.append([var_order[k], str(parent_idx), str(left_child_idx), str(right_child_idx)])
```

```
66      start_idx = 2 ** n
67      end_idx = 2 ** (n + 1)
68      for i in range(start_idx, end_idx):
69          parent_idx = i // 2
70          Seifos.append([truthTable[i - start_idx], str(parent_idx), '/', '/'])
71      print("Bdd:!!!!!!!!!!!!!!!!!!!!!!!!!!!!")
72      printTree(Seifos, n)
73      return Seifos
```

D. ROBDD Logic: -

Terminal nodes handling: -

The provided function, doSeifos, is designed to process a Binary Decision Diagram (BDD) to reduce redundancy by consolidating terminal nodes (0 and 1). In a BDD, terminal nodes represent the final output values of a Boolean function. The goal here is to ensure that there is exactly one node representing 0 and one node representing 1, eliminating any duplicates. The function works by iterating through the terminal nodes and redirecting any redundant pointers to the canonical (unique) 0 or 1 node.

```
for i in range(2 ** (n - 1), 2 ** n):
    if Seifos[int(Seifos[i][2])][0] == "0":  # Left child
        if zero_count:
            Seifos[int(Seifos[i][2])][1] = "/"
            Seifos[i][2] = str(zero_index)
        else:
            zero_count += 1
            zero_index = int(Seifos[i][2])
    elif Seifos[int(Seifos[i][2])][0] == "1":  # Left child
        if one_count:
            Seifos[int(Seifos[i][2])][1] = "/"
            Seifos[i][2] = str(one_index)
        else:
            one_count += 1
            one_index = int(Seifos[i][2])

    if Seifos[int(Seifos[i][3])][0] == "0":  # Right child
        if zero_count:
            Seifos[int(Seifos[i][3])][1] = "/"
            Seifos[i][3] = str(zero_index)
        else:
            zero_count += 1
            zero_index = int(Seifos[i][3])
    elif Seifos[int(Seifos[i][3])][0] == "1":  # Right child
        if one_count:
            Seifos[int(Seifos[i][3])][1] = "/"
            Seifos[i][3] = str(one_index)
        else:
            one_count += 1
            one_index = int(Seifos[i][3])
```

The iteration begins with the loop for i in range (2 ** (n - 1), 2 ** n). This range represents the indices of the level before the terminal nodes, which are typically located before the end of the BDD structure. The function processes each node by examining its left (Seifos[i][2]) and right (Seifos[i][3]) child pointers. If a child node points to a 0 or 1 node, the function checks whether a canonical 0 or 1 node has already been designated. If such a node exists, the current pointer is redirected to it. If not, the current 0 or 1 node is marked as canonical, and its index is stored for future reference.

To keep track of canonical nodes, the function uses the variables zero_count and one_count to determine whether a 0 or 1 node has already been encountered. It also uses zero_index and one_index to store the indices of these canonical nodes. When a redundant 0 or 1 node is identified, it is effectively ignored by redirecting its parent's pointer to the canonical node and marking it with a placeholder (e.g., /) to signify that it is no longer in use.

```
zero_count = 0
zero_index = -1
one_count = 0
one_index = -1
Seifos = copy.deepcopy(BDDSeifos)
```

By consolidating all 0 and 1 nodes into a single instance each, the function reduces the overall size of the BDD, improving space efficiency. For example, if two nodes, Seifos [5] and Seifos [7], represent 0, the first encountered node (e.g., Seifos [5]) becomes the canonical 0 node. Any subsequent nodes pointing to a 0 are then redirected to this canonical node, ensuring that only one 0 node exists in the final structure. This process is repeated for 1 nodes as well.

Removing Redundant nodes: -

This section of the code processes the Binary Decision Diagram (BDD) to further reduce its size by eliminating redundant nodes and merging equivalent nodes. The goal is to optimize the BDD by ensuring that no unnecessary nodes exist while maintaining the same Boolean function representation. The optimization is achieved through two key operations: identifying redundant nodes whose left and right children are identical and merging equivalent nodes that share the same children and behavior.

The function iterates over the BDD level by level, starting from the deepest level and moving upward toward the root. This is accomplished with the outer loop, which traverses the levels in reverse order, and an inner loop that processes each node in the current level. By starting from the terminal nodes, the function ensures that optimizations at lower levels are propagated upward.

When processing a node, the function first checks for redundancy by comparing its left and right child pointers. If both pointers are identical, the node is considered redundant because it does not provide additional information. The parent of this node is then updated to bypass the redundant node and point directly to its child. This effectively removes the redundant node from the BDD, and the node is marked as inactive. The parent pointers of the child node are also updated to reflect the change.

```
for i in range(n, 0, -1):
    for j in range(2 ** (i - 1), 2 ** i):
        if Seifos[j][1] != "/":
            if Seifos[j][2] == Seifos[j][3]:
                parent = int(Seifos[j][1])
                if 2 * parent == j:  # Left child
                    Seifos[parent][2] = Seifos[j][2]
                else:  # Right child
                    Seifos[parent][3] = Seifos[j][2]
                Seifos[int(Seifos[j][2])][1] = Seifos[j][1]
                Seifos[j][1] = "/"
```

The function checks for equivalence with other nodes in the same level. Any two nodes are considered equivalent if they have the same left and right children. In such cases, the parent of the second node is updated to point to the first node, effectively merging the two nodes into one. The second node is then marked as inactive, ensuring that no duplicate nodes exist in the BDD. This process reduces redundancy further and ensures that the BDD remains as compact as possible.

```
else:
    for k in range(j + 1, 2 ** i):
        if Seifos[k][1] != "/" and Seifos[j][2:] == Seifos[k][2:]:
            parent = int(Seifos[k][1])
            if 2 * parent == k:  # Left child
                Seifos[parent][2] = str(j)
            else:  # Right child
                Seifos[parent][3] = str(j)
            Seifos[k][1] = "/"
```

```
for i in range(n, 0, -1):
    for j in range(2 ** (i - 1), 2 ** i):
        if Seifos[j][1] != "/":
            if Seifos[j][2] == Seifos[j][3]:
                parent = int(Seifos[j][1])
                if 2 * parent == j:  # Left child
                    Seifos[parent][2] = Seifos[j][2]
                else:  # Right child
                    Seifos[parent][3] = Seifos[j][2]
                Seifos[int(Seifos[j][2])][1] = Seifos[j][1]
                Seifos[j][1] = "/"
            else:
                for k in range(j + 1, 2 ** i):
                    if Seifos[k][1] != "/" and Seifos[j][2:] == Seifos[k][2:]:
                        parent = int(Seifos[k][1])
                        if 2 * parent == k:  # Left child
                            Seifos[parent][2] = str(j)
                        else:  # Right child
                            Seifos[parent][3] = str(j)
                        Seifos[k][1] = "/"
```

E. Comparison Function:

When comparing two ROBDDs, the variable order provided by the user directly influences the structure and evaluation of the diagrams. Even if two Boolean functions are equivalent, their ROBDD representations might differ if built using different variable orders.

When comparing two ROBDDs, there are several methods to check their equivalence, each with its own pros and cons. These methods aim to verify whether two Boolean functions, represented by the ROBDDs, produce the same output for all possible input combinations. Some of these methods are (Structural comparison, canonical form comparison, etc.) but the most efficient method to compare is using the truth table. It is guaranteed that the output of the equivalence is correct for any expression, and for any number of inputs.

Step 3: Sort nodes based on their variables before returning

We will start the code from step 3 in **doSeifos.** As we explained in the previous section, the function **doSeifos** takes 3 arguments, **BDDSeifos** (the BDD), the number of variables (**n**) and the variable order (**var_order**). Step 3 in this function is responsible for sorting the nodes based on the variable order.

```
valid_nodes = []
for i in range(len(Seifos)):
    if Seifos[i][1] != "/" or i == 1:
        valid_nodes.append((i, Seifos[i]))
```

This snippet of the code identifies valid_nodes in the Seifos list. A valid node is defined whether the node has a valid parent or is not a terminal node (its parent index Seifos[i][1] not = /) or it's the root node (root index at i=1) it is always considered a valid node, regardless of its parent. For each node that passes the condition, the node is added to valid_nodes. This list will now hold all nodes that are either not terminal or are the root.

After that, we defined an internal function inside **deSeifos** responsible for sorting the nodes by variable order. This **sort_key** function defines how the nodes should be sorted. It handles couple of cases.

1. If the node doesn't have a parent or it's a terminal node (it has a value of 1 or 0) so this is why we put **isdigit()** method. We place these nodes last in the sorted list by returning a tuple **(len(var_order) + 1, 0)**

   ```
   def sort_key(item):
       index, node_data = item
       if node_data[0] == "/" or node_data[0].isdigit():
           return (len(var_order) + 1, 0)
       else:
           return (var_order.index(node_data[0]), index)
   ```

   len(var_order) + 1: Ensures that terminal nodes are pushed to the end by assigning them a priority that's greater than any of the valid nodes.
   0: Provides a secondary sort key in case multiple terminal nodes need to be sorted.

2. If the node is not a terminal node. The first element of **node_data** is assumed to be a variable. The function **var_order.index(node_data[0])** retrieves the position of the variable in the user-provided **var_order** list. This determines the priority of the node. The index is also included as a secondary sort key to ensure that nodes with the same variable order are sorted in their original order in valid_nodes.

The **sort_key** function helps to arrange the nodes such that non-terminal nodes appear in the order specified by the user in var_order, while terminal nodes are placed at the end.

```
sorted_nodes = sorted(valid_nodes, key=sort_key)
```

The list **valid_nodes** is now sorted using the **sort_key** function we just defined. This ensures that the nodes are ordered based on the variable order defined by the user, with terminal nodes appearing at the end.

Step 4: Update indices after sorting

After sorting the nodes in the desired order, based on the variable order, we need to update the indices of the nodes in their new positions. This ensures that the parent-child relation in the BDD are maintained correctly.

```
new_indices = {old_index: new_index for new_index, (old_index, _) in
                enumerate(sorted_nodes)}
```

This line is creating a mapping (**new_indices**) that will help update the node indices after sorting the nodes. **enumerate(sorted_nodes)**: This generates pairs of new_index and (old_index, _) where:

- **new_index:** the index of the node in the sorted list sorted_nodes.
- **old_index**: the original index of the node in Seifos.

The goal of this line is to map **old indices** (from the original Seifos BDD) to **new indices** (after sorting the nodes). This allows us to correctly update the indices of the parent and child nodes when we sort the BDD.

```
for i in range(len(sorted_seifos)):
    if sorted_seifos[i][1] != "/" and sorted_seifos[i][1].isdigit():  # update parent
        sorted_seifos[i][1] = str(new_indices[int(sorted_seifos[i][1])])
    if sorted_seifos[i][2] != "/" and sorted_seifos[i][2].isdigit():  # update left child
        sorted_seifos[i][2] = str(new_indices[int(sorted_seifos[i][2])])
    if sorted_seifos[i][3] != "/" and sorted_seifos[i][3].isdigit():  # update right child
        sorted_seifos[i][3] = str(new_indices[int(sorted_seifos[i][3])])
return Seifos,sorted_seifos
```

After sorting the nodes, we need to ensure that the **parent**, **left child**, and **right child** indices of each node are updated to match the new order. So we put 3 conditions for parent, left and right.

If the parent index (sorted_seifos[i][1]) is not "/" (indicating that it's a valid parent node) and is a digit, we update the parent index using the new_indices mapping.

The same approach is applied for the left child (sorted_seifos[i][2]) and the right child (sorted_seifos[i][3]). The check sorted_seifos[i][1] != "/" ensures we're only updating nodes that are non-terminal (non-leaf nodes), as terminal nodes don't have a parent.

This functions returns the original BDD (Seifos) and the sorted_seifos where the nodes have been re-ordered according to the variable order.

Evaluate_seifos:

This function is designed to evaluate ROBDD for a given variable assignment and order. It takes 3 parameters (seifos [represent the ROBDD], assignment[a list of 0 and 1 representing the truth table], var_order). We start evaluating from index 1 which is the root node.

We enter an infinite loop (while True) that will continue until a terminal node ('0' or '1') is encountered.

current_node_index is used to reference the current node in the **seifos** list.

```
while True:
    current_node = seifos[current_node_index]
    if current_node[0] == '0' or current_node[0] == '1':
        return int(current_node[0])
```

current_node = seifos[current_node_index]: We get the current node from the seifos list.

If **current_node[0]** is '0' or '1', it means we've reached a terminal node. The function directly returns the integer value of the terminal node (either 0 or 1).

If the node is not terminal, it represents a decision on a variable

variable: The variable at the current node ex: ('A', 'B').

var_index: The index of the variable in the **var_order** list, used to determine the current variable in the assignment.

```
variable = current_node[0]
var_index = var_order.index(variable)
```

If the current assignment value of the variable (assignment[var_index]) is 0, we move to the left child of the current node.

If the left child (current_node[2]) is not a terminal value, we update current_node_index to point to the left child by setting current_node_index = int(current_node[2]).

```
if assignment[var_index] == 0:

    if not current_node[2].isdigit():  # Terminal node
        return int(current_node[2])
    current_node_index = int(current_node[2])
else:
    if not current_node[3].isdigit():
        return int(current_node[3])
    current_node_index = int(current_node[3])
```

If the assignment value of the variable is 1, we move to the right child (current_node[3]) and point to the right child as well.

Compare_ROBDD

This bool function compares two ROBDDs to check if they are **equivalent** for all possible assignments of variables, respecting the variable order provided.

It takes 4 parameters. (seifos1, var_order1,seifos2,var_order2)

Where: seifos1 represent the first ROBDD. Same for seifos2

Var_order1= is the variable order of seifos1, same for seifos2.

all_variables: This combines the variable orders from both ROBDDs and removes duplicates by using set. After that, it sorts the variables.

num_variables: The total number of variables after combining both variable orders.

```
all_variables = sorted(list(set(var_order1 + var_order2)))
num_variables = len(all_variables)

for i in range(2 ** num_variables):
    assignment = [(i >> j) & 1 for j in range(num_variables)]
    val1 = evaluate_seifos(seifos1, assignment, var_order1)
    val2 = evaluate_seifos(seifos2, assignment, var_order2)

    if val1 != val2:
        return False
return True
```

We loop through all possible assignments of the variables (there are $2^{num\_variables}$ possible assignments). For each assignment (i), we generate a list assignment that represents the assignment of values (0 or 1) to all variables **in all_variables**. [(i >> j) & 1 for j in range(num_variables)] extracts the j-th bit of i and uses it as the assignment for the j-th variable.

val1: The result of evaluating seifos1 with the current assignment and var_order1.

val2: The result of evaluating seifos2 with the current assignment and var_order2.

If for any assignment i, the values **val1** and **val2** differ (not same output), the function immediately returns False, indicating the ROBDDs are **not equivalent**.

This approach guarantees the correctness of the check for any ROBDD

F. Visualizer:

Our goal is to generate a graphical representation of the Binary Decision Diagram (BDD) and Reduced Ordered Binary Decision Diagram (ROBDD). The function accepts a list of nodes where each node is defined as: [name, parent, left_child, right_child].

- **name**: The variable name or terminal value (e.g., a, b, 0, 1).
- **parent**: The parent node index or / if no parent exists.
- **left_child**: The index of the left child node or / if absent.
- **right_child**: The index of the right child node or / if absent.

We used networkx and matplotlib libraries to draw and label the graph. Edge labels (0 or 1) are displayed to indicate the type of child.

```
    # Parse input into nodes and edges
    nodes = {}
    edges = []
    edge_labels = {}  # To store labels for edges
    variables = []  # To track unique variables (e.g., 'a', 'b', 'c')
    if input_nodes[0][0] == '/':
        input_nodes.remove(input_nodes[0])
    for i, entry in enumerate(input_nodes):
        node_index = i + 1
        name, parent, left_child, right_child = entry
        nodes[node_index] = {'name': name, 'parent': parent, 'left': left_child, 'right': right_child}

        # Add variables to the list if they're not terminal nodes
        if name.isalpha() and name not in variables:
            variables.append(name)
```

This code block processes the input_nodes list to prepare a structured representation of nodes and for constructing a Reduced Ordered Binary Decision Diagram (ROBDD). First, it initializes several data structures: nodes as a dictionary to store information about each node (such as its name, parent, and children), edges as a list to later hold parent-child relationships, edge_labels to label edges (e.g., 0 for left child and 1 for right child), and variables as a list to track unique decision variables (e.g., 'a', 'b', 'c'). It then checks if the first node in the input_nodes list contains '/' as its first value, which signifies an invalid entry, and removes it if necessary to ensure only valid nodes are processed. The main part of the code iterates through the input_nodes list, assigning each node a unique index starting from 1 and extracting its attributes: name (the variable or terminal value), parent

(indicating which node it derives from), left_child, and right_child (defining the indices of its children). Each node's data is stored in the node dictionary, indexed by its unique node number. Simultaneously, the code checks if the node's name consists of alphabetic characters and adding it to the variables list if not already present, thereby excluding terminal nodes ('0' or '1') and invalid entries ('/').

The overall purpose of this block is to organize nodes into a structured format, identify and store unique variables for later use in level assignment within the ROBDD, and prepare foundational data (nodes, variables, edges) required to construct the graph.

```
    # Filter valid nodes and include the first node (index 1) even if its parent is '/'
    valid_nodes = set(idx for idx, node in nodes.items() if node['parent'] != '/' or idx == 1)

    for idx, node in nodes.items():
        parent = node['parent']
        left_child = node['left']
        right_child = node['right']

        # Skip nodes with invalid parents, except the first node (index 1)
        if idx not in valid_nodes:
            continue

        # Correct edges for valid parent-child relationships
        if left_child != '/' and int(left_child) in valid_nodes:
            edges.append((node['name'] + f"_{idx}", nodes[int(left_child)]['name'] + f"_{left_child}"))
            edge_labels[(node['name'] + f"_{idx}", nodes[int(left_child)]['name'] + f"_{left_child}")] = '0'  # Label left edge as '0'
        if right_child != '/' and int(right_child) in valid_nodes:
            edges.append((node['name'] + f"_{idx}", nodes[int(right_child)]['name'] + f"_{right_child}"))
            edge_labels[(node['name'] + f"_{idx}", nodes[int(right_child)]['name'] + f"_{right_child}")] = '1'  # Label right edge as '1'
```

This part processes the nodes dictionary to identify valid nodes and construct edges for a graph representation of the ROBDD. It begins by creating a set of valid_nodes, which includes all nodes where the parent is not '/' or the node is the root (index 1). Then, it iterates through all nodes in the nodes dictionary. For each node, it retrieves the parent, left_child, and right_child attributes. Nodes not in the valid_nodes set are skipped to exclude invalid entries. For valid nodes, it checks whether the left_child and right_child indices are valid and present in valid_nodes. If valid, edges are created between the current node and its children, with the child's name and index appended to the parent's name and index for unique identification. Edges are added to the edges list, and the edge_labels dictionary is updated to label the edges as '0' for the left child

and '1' for the right child. This step ensures that only valid parent-child relationships are captured and prepares the graph's edge data for visualization.

```
    # Assign levels to variables
    level_map = {var: i for i, var in enumerate(variables)}
    max_level = len(variables)  # Max level for terminal nodes like 0 and 1

    # Determine total width needed for the bottom level of the pyramid
    total_width = 2 ** max_level - 1

    # Generate positions for nodes in pyramid layout
    pos = {}
    level_counts = defaultdict(int)  # To track the number of nodes at each level

    for idx, node in nodes.items():
        if idx in valid_nodes:
            name = node['name']
            # Determine the level
            if name.isalpha():  # Variables like 'a', 'b', 'c'
                level = level_map[name]
            else:  # Terminal nodes (e.g., '0', '1')
                level = max_level

            # Calculate x and y coordinates for the pyramid structure
            level_width = 2 ** level
            x_spacing = total_width / level_width
            x_pos = (level_counts[level] + 0.5) * x_spacing - total_width / 2  # Center nodes at each level
            y_pos = -level  # Y decreases as level increases
            pos[name + f"_{idx}"] = (x_pos, y_pos)
            level_counts[level] += 1
```

This part calculates the positions of nodes for a pyramid-structured layout in a graphical representation of an ROBDD. It uses the variables list, which contains the unique variable names, to create a level_map that assigns each variable to a level based on its position in the list. The maximum level (max_level) is determined by the number of variables, with terminal nodes (e.g., 0 and 1) placed at the bottom-most level. The pos dictionary stores the coordinates of each node, and the level_counts dictionary keeps track of the number of nodes already placed at each level. For each valid node in nodes, the code determines its level: variables (e.g., a, b, c) use their corresponding level from level_map, while terminal nodes are assigned max_level. At each level, the width of that level is divided into equally spaced segments (x_spacing), and the x-coordinate is calculated based on the node's position within the level. The y-coordinate is simply the negative of the level, ensuring that higher levels are placed higher in the graph. The computed position (x_pos, y_pos) is stored in the pos dictionary with a unique identifier for the node, and the node count for that level is incremented in level_counts.

This part ensures that the root node of the ROBDD is centered horizontally in the graphical representation. The variable root_name identifies the root node, which is always the first node in the nodes dictionary (index 1). Its name is combined with its index to match the naming convention used in the pos dictionary. The coordinates of the root node (root_x, root_y) are then retrieved from pos. To center the root node horizontally, the offset is calculated as the difference between half of the total width of the bottom-most level (total_width / 2) and the current x-coordinate of the root node (root_x). This offset represents how much the root node needs to be shifted along the x-axis to align it at the center. The for loop iterates through all nodes in the pos dictionary, adjusts their x-coordinates by adding the calculated offset, and updates their positions. This ensures that the entire graph is shifted appropriately, keeping the relative distances between nodes unchanged while centering the root node.

```
    # Center the root node (level 0) exactly at the middle of the x-axis
    root_name = nodes[1]['name'] + "_1"  # Root node is always index 1
    root_x, root_y = pos[root_name]
    offset = total_width / 2 - root_x
    for node_name in pos:
        x, y = pos[node_name]
        pos[node_name] = (x + offset, y)
```

```
    # Create the directed graph
    G = nx.DiGraph()
    for edge in edges:
        G.add_edge(*edge)

    # Plot the graph with differentiated edges
    plt.figure(figsize=(12, 8))
    edge_colors = [
        'blue' if edge_labels[edge] == '0' else 'red'
        for edge in G.edges
    ]  # '0' edges are blue, '1' edges are red
    nx.draw(
        G, pos, with_labels=True, node_size=2000, node_color="skyblue",
        font_size=10, arrowsize=20, edge_color=edge_colors, width=2
    )

    # Add labels to edges for '0' and '1'
    nx.draw_networkx_edge_labels(
        G, pos, edge_labels=edge_labels, font_size=10, font_color="black"
    )

    plt.title("Centered Pyramid-Structured Reduced Ordered Binary Decision Diagram (ROBDD)")
    plt.show()
```

This part visualizes the Reduced Ordered Binary Decision Diagram (ROBDD) using the networkx library.

1. Graph Creation:

- A directed graph (DiGraph) object G is created.
- Edges from the edges list are added to the graph using G.add_edge.

2. Edge Coloring:

The edge_colors list is created to differentiate edges visually:

- Edges labeled '0' (representing left-child edges) are colored blue.
- Edges labeled '1' (representing right-child edges) are colored red.

3. Graph Drawing:

The plt.figure function sets up the figure with a size of 12x8 inches for better clarity.

The nx.draw function renders the graph with:

- Node labels (with_labels=True) for identification.
- Large, sky-blue-colored nodes (node_color="skyblue", node_size=2000).
- Red and blue edge colors (edge_color=edge_colors) based on their labels.
- Adjustable arrow size (arrowsize=20) and edge width (width=2) for better visibility.

4. Edge Labeling:

The nx.draw_networkx_edge_labels function adds labels ('0' or '1') to the edges. These labels are fetched from the edge_labels dictionary and rendered in black font (font_color="black").

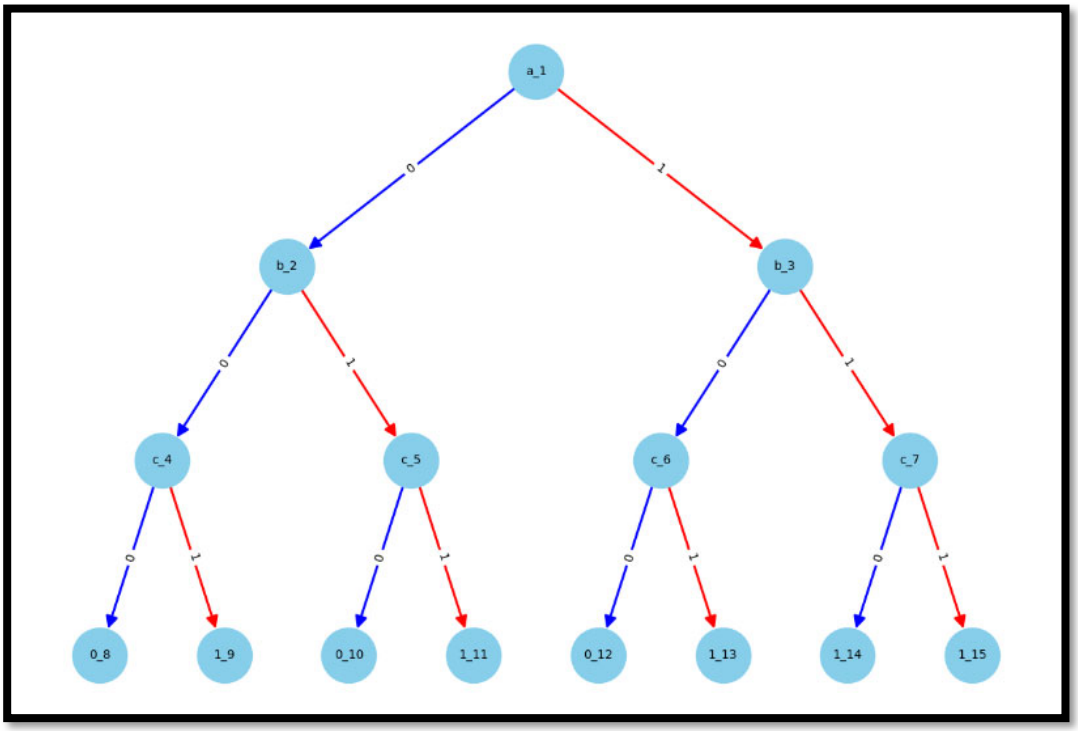
5. Final Visualization:

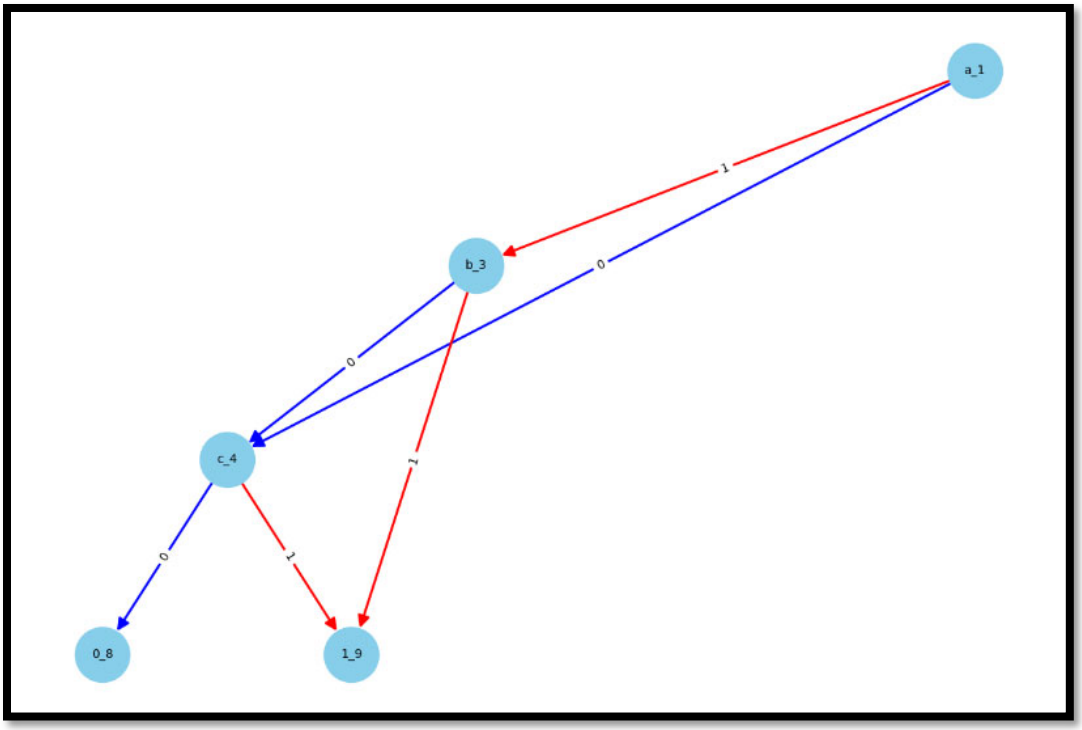
- A title is added to the plot: "Centered Pyramid-Structured Reduced Ordered Binary Decision Diagram (ROBDD)".
- The plt.show function displays the completed diagram.

```
# Example Input
input_nodes = [
    ['a', '/', '4', '3'],  # First node (index 1) has parent '/', but it should still appear
    ['b', '/', '4', '4'],  # This node should not appear
    ['b', '1', '4', '9'],  # Valid node with parent '1'
    ['c', '1', '8', '9'],  # Valid node with parent '1'
    ['c', '/', '8', '9'],  # This node should not appear
    ['c', '/', '8', '9'],  # This node should not appear
    ['c', '/', '9', '9'],  # This node should not appear
    ['0', '4', '/', '/'],  # Valid terminal node
    ['1', '3', '/', '/'],  # Valid terminal node
    ['0', '/', '/', '/'],  # Valid terminal node
    ['1', '/', '/', '/'],  # Valid terminal node
    ['0', '/', '/', '/'],  # Valid terminal node
    ['1', '/', '/', '/'],  # Valid terminal node
    ['0', '/', '/', '/'],  # Valid terminal node
    ['1', '/', '/', '/'],  # Valid terminal node
]
```

```
# Plot the ROBDD
plot_robdd(input_nodes)


input_nodes = [
     ['a', '/', '2', '3'],
     ['b', '1', '4', '5'],
     ['b', '1', '6', '7'],
     ['c', '2', '8', '9'],
     ['c', '2', '10', '11'],
     ['c', '3', '12', '13'],
     ['c', '3', '14', '15'],
     ['0', '4', '/', '/'],
     ['1', '4', '/', '/'],
     ['0', '5', '/', '/'],
     ['1', '5', '/', '/'],
     ['0', '6', '/', '/'],
     ['1', '6', '/', '/'],
     ['1', '7', '/', '/'],
     ['1', '7', '/', '/'],
 ]
plot_robdd(input_nodes)
```

The result of the previously demonstrated input is as follows:

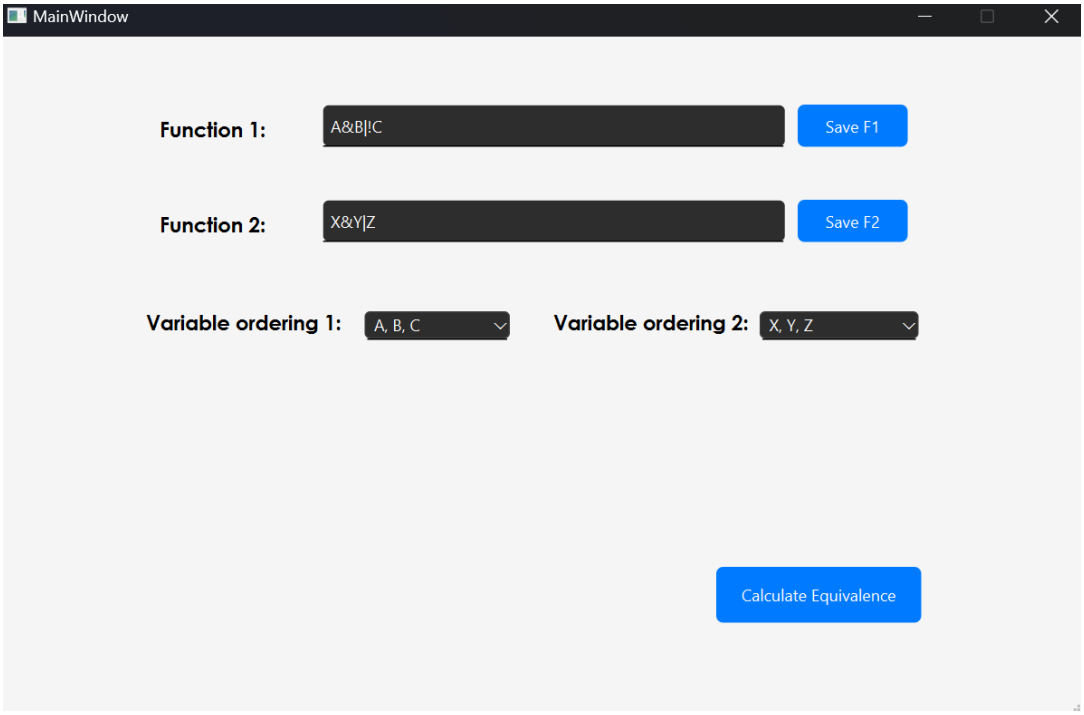




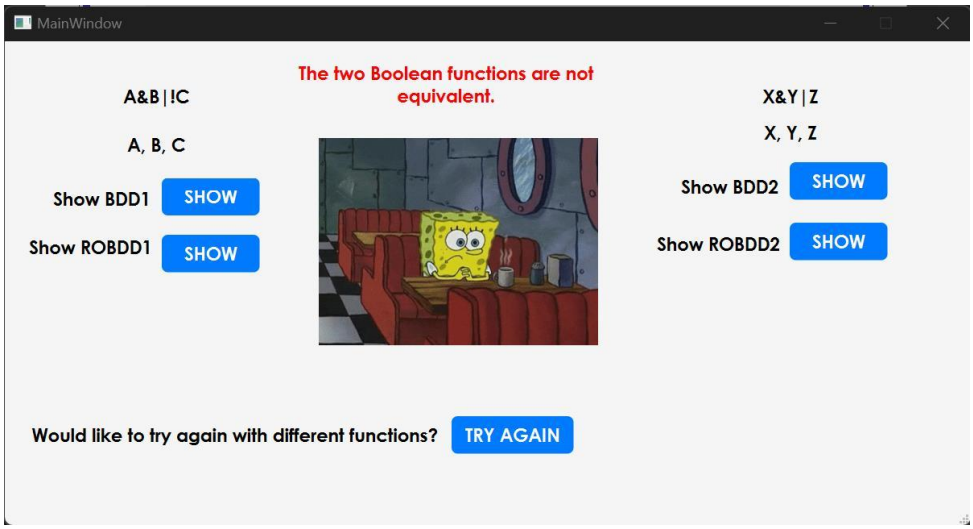
6. Test Cases:

Here are 3 different Test cases for the Application:

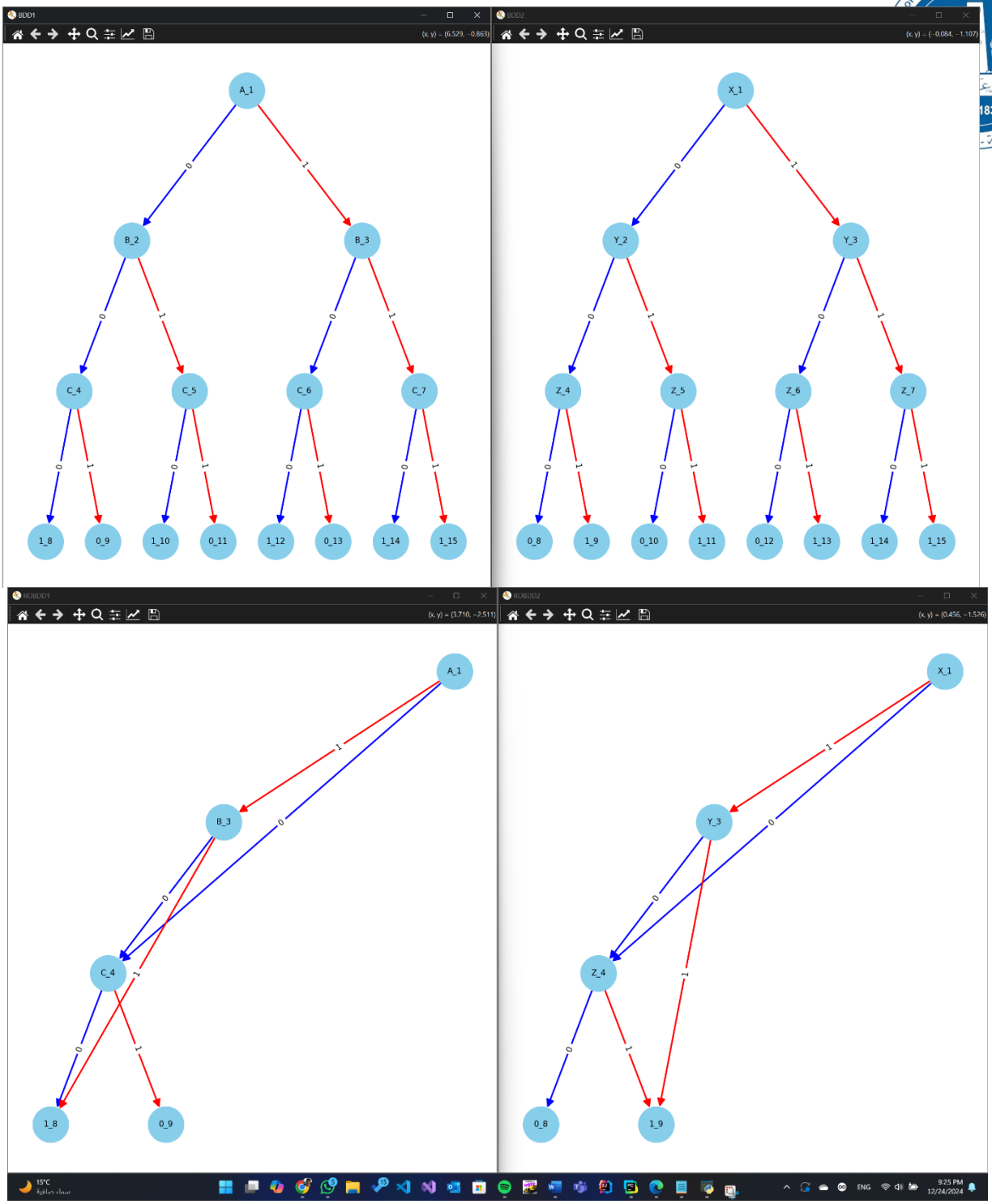
1- F 1 = A&B|!C and F 2 = X&Y|Z:



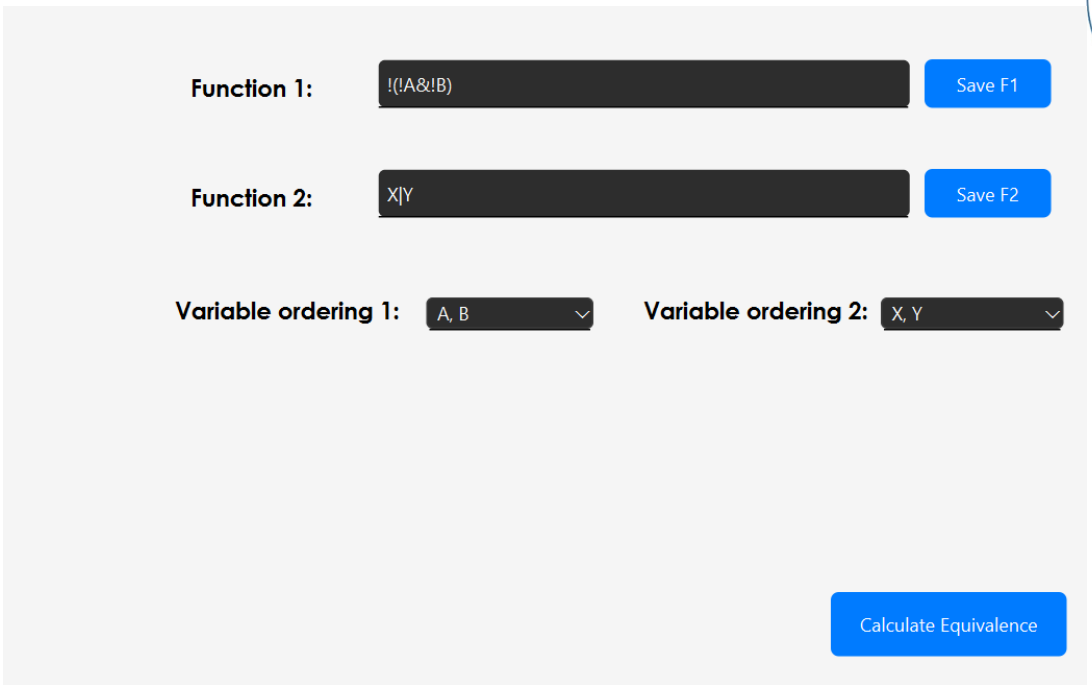
The output of this testcase is false which is the correct answer.



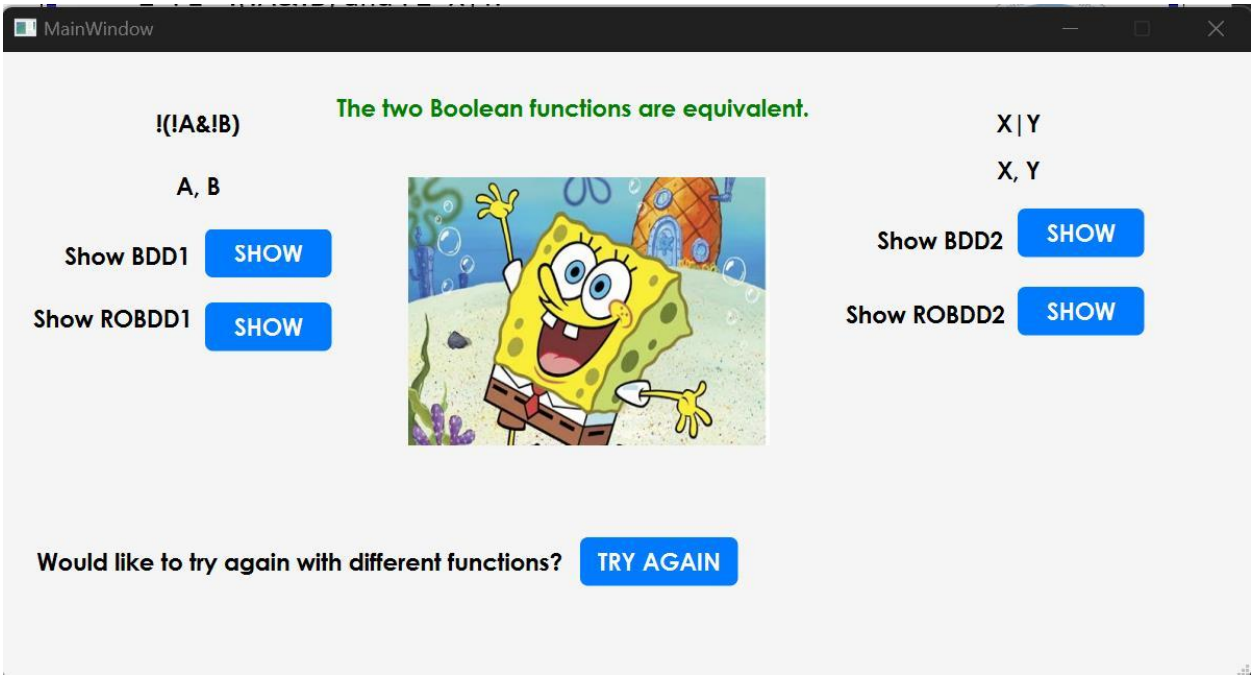
Here are the graphs:



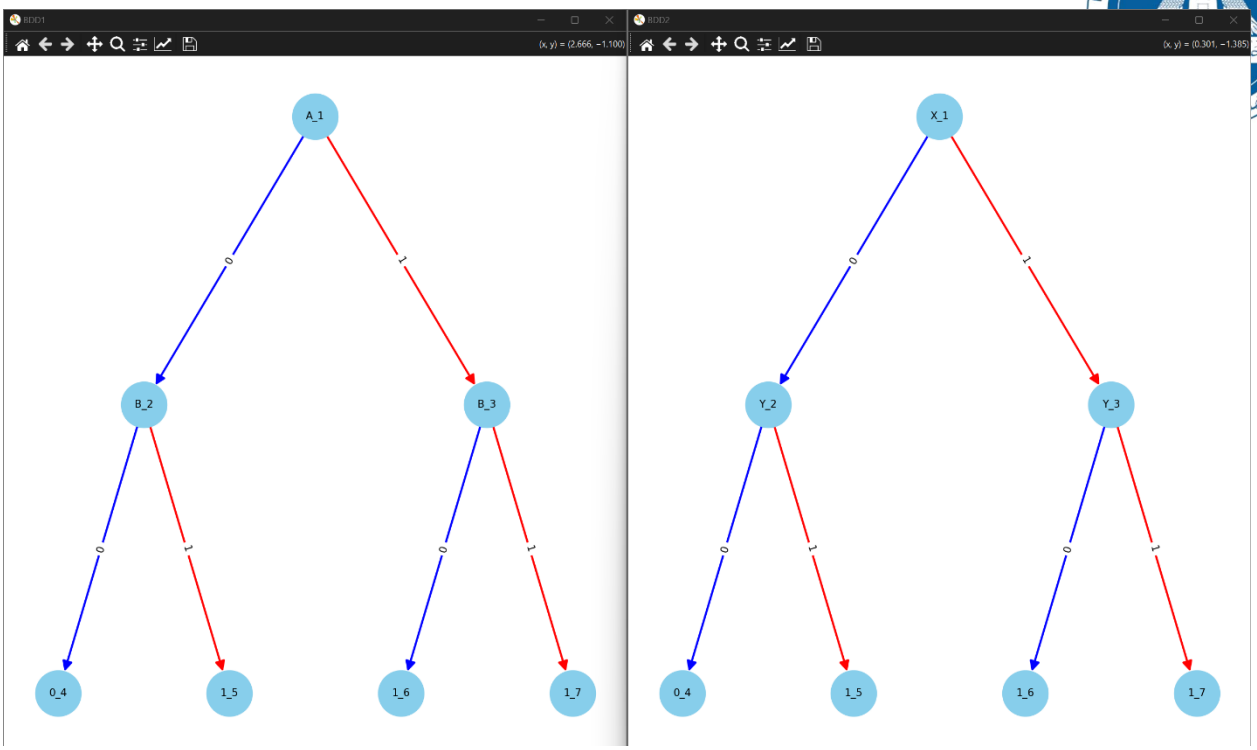
2- F1= !(!A&!B) and F2=X|Y:

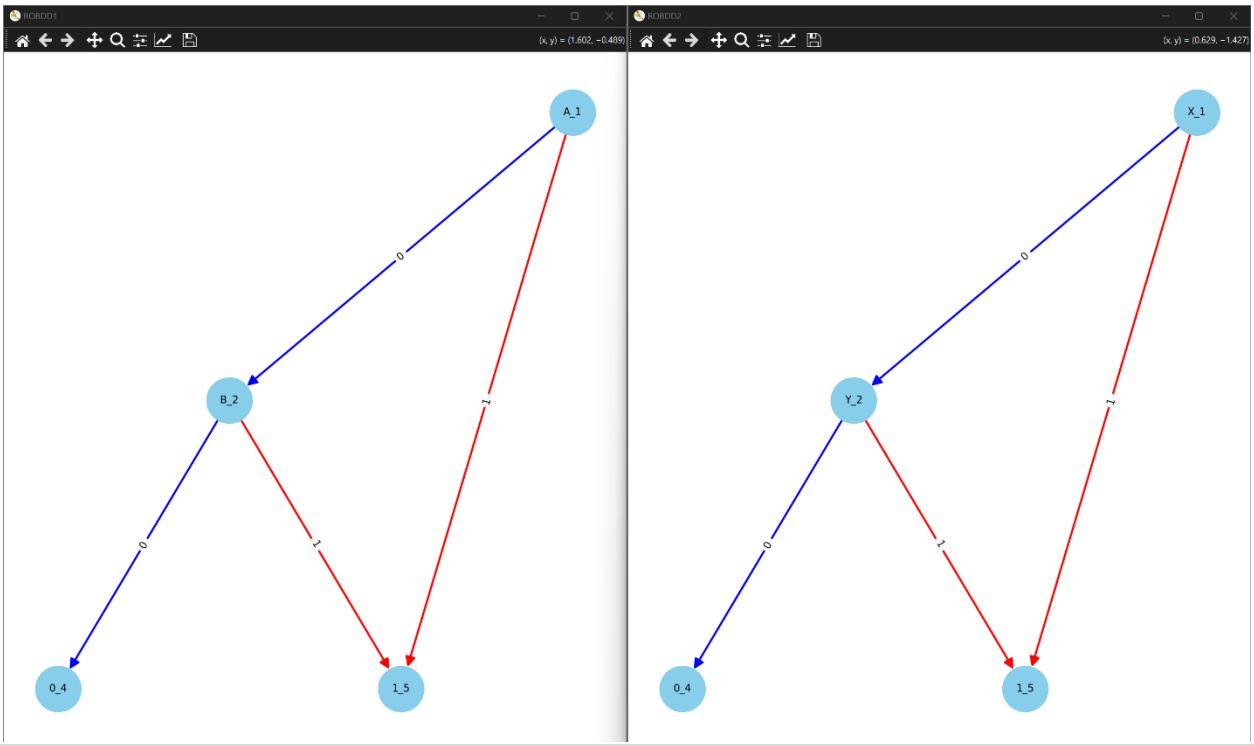


The output of this testcase is false which is the correct answer.

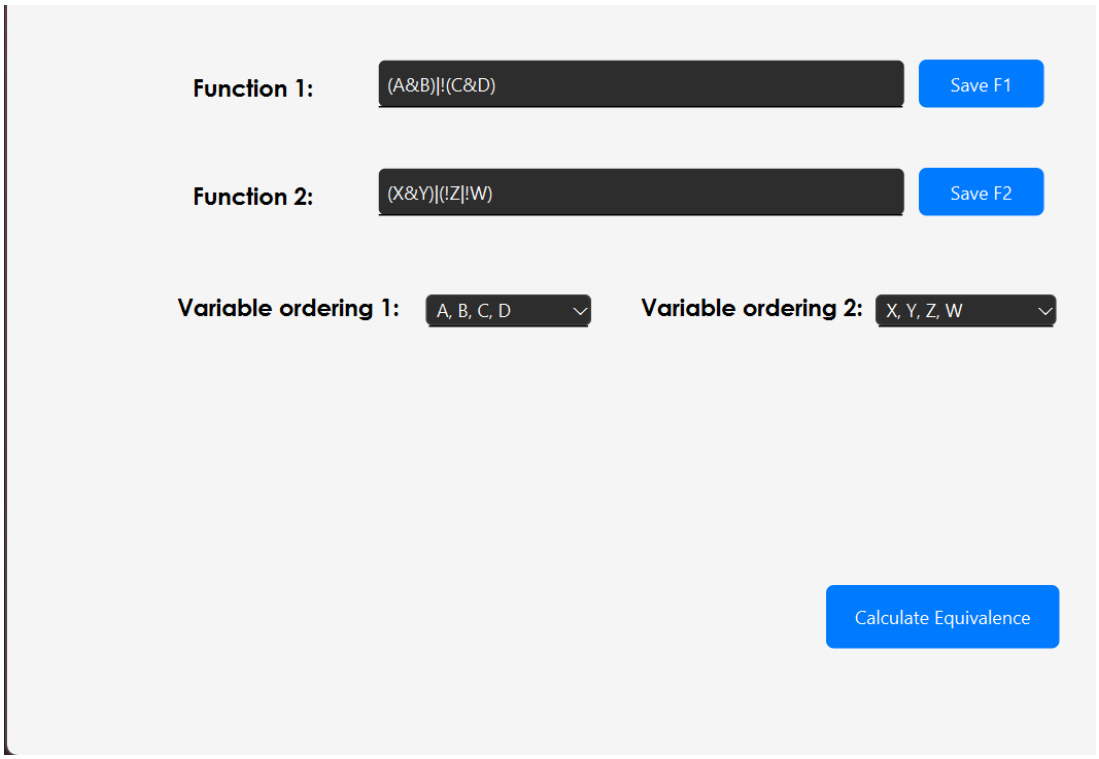


Here are the graphs:

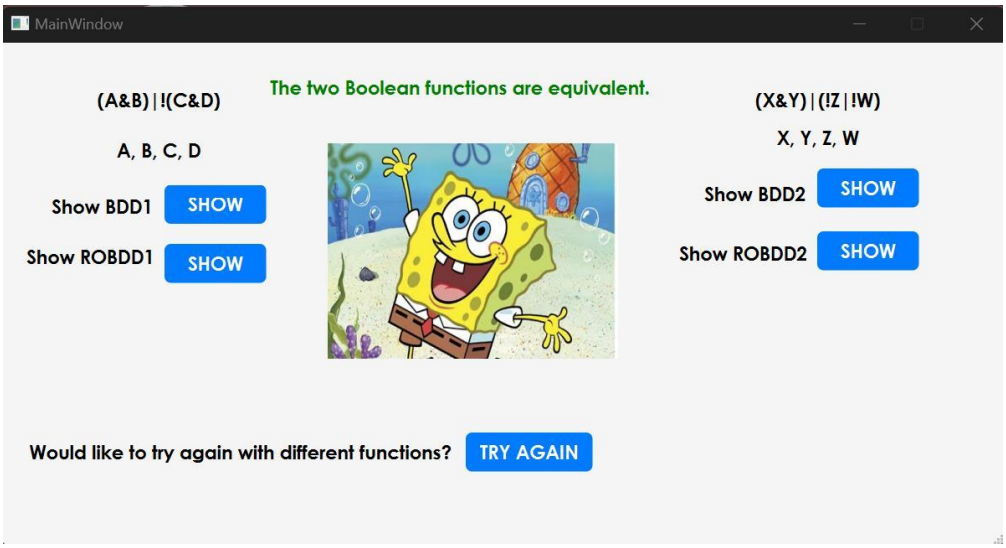




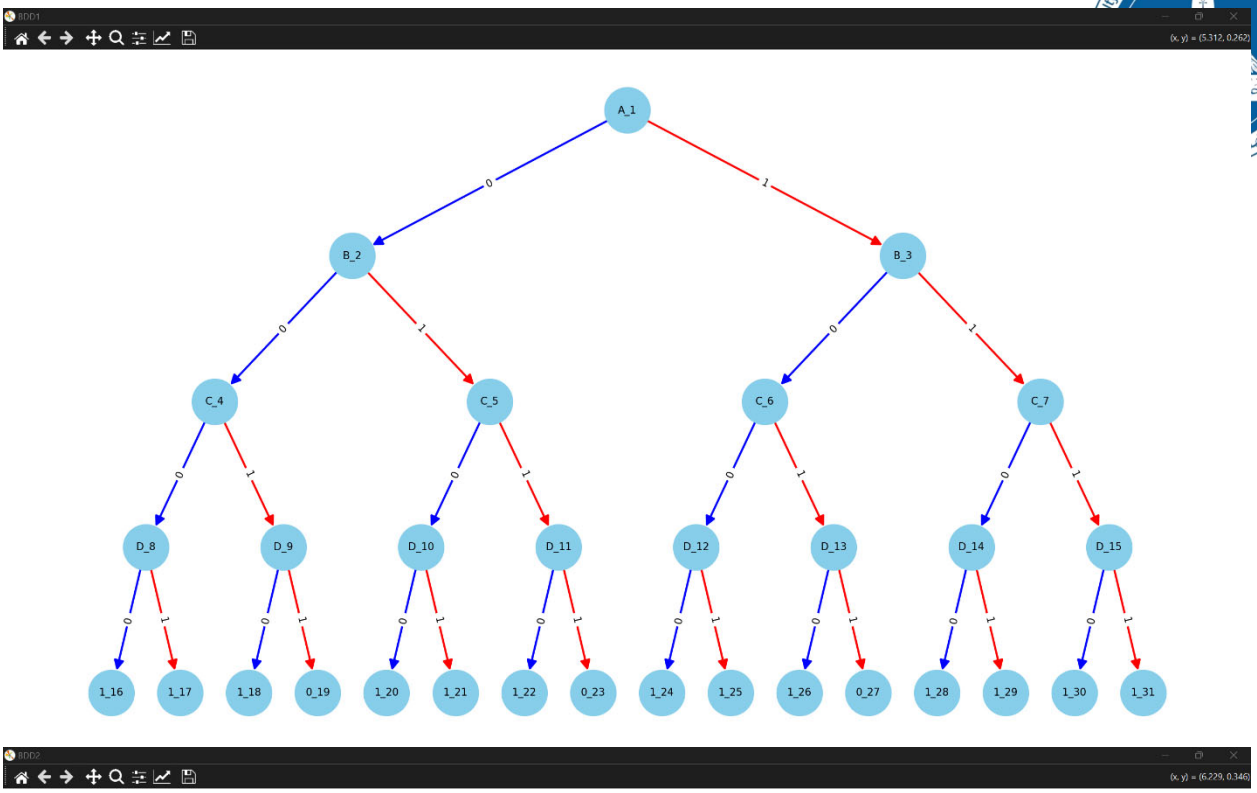
3- F1 = (A&B)| !(C & D) and F2 = (X & Y) | (!Z | !W):

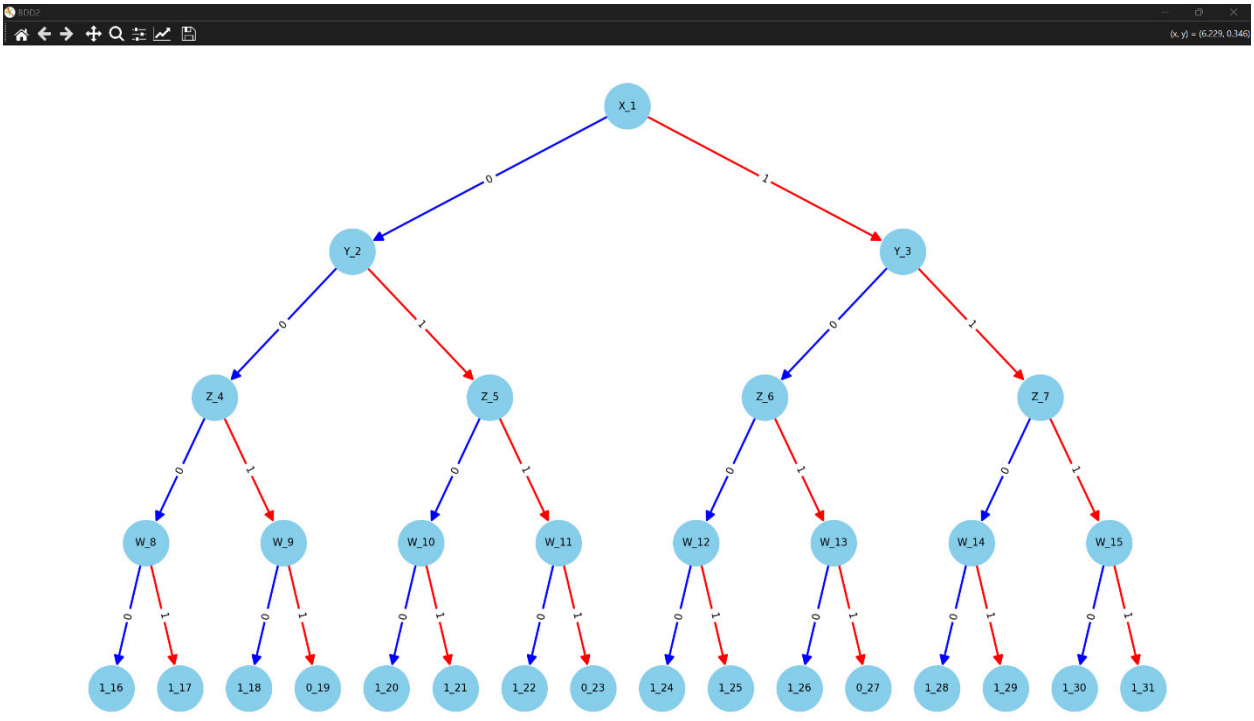


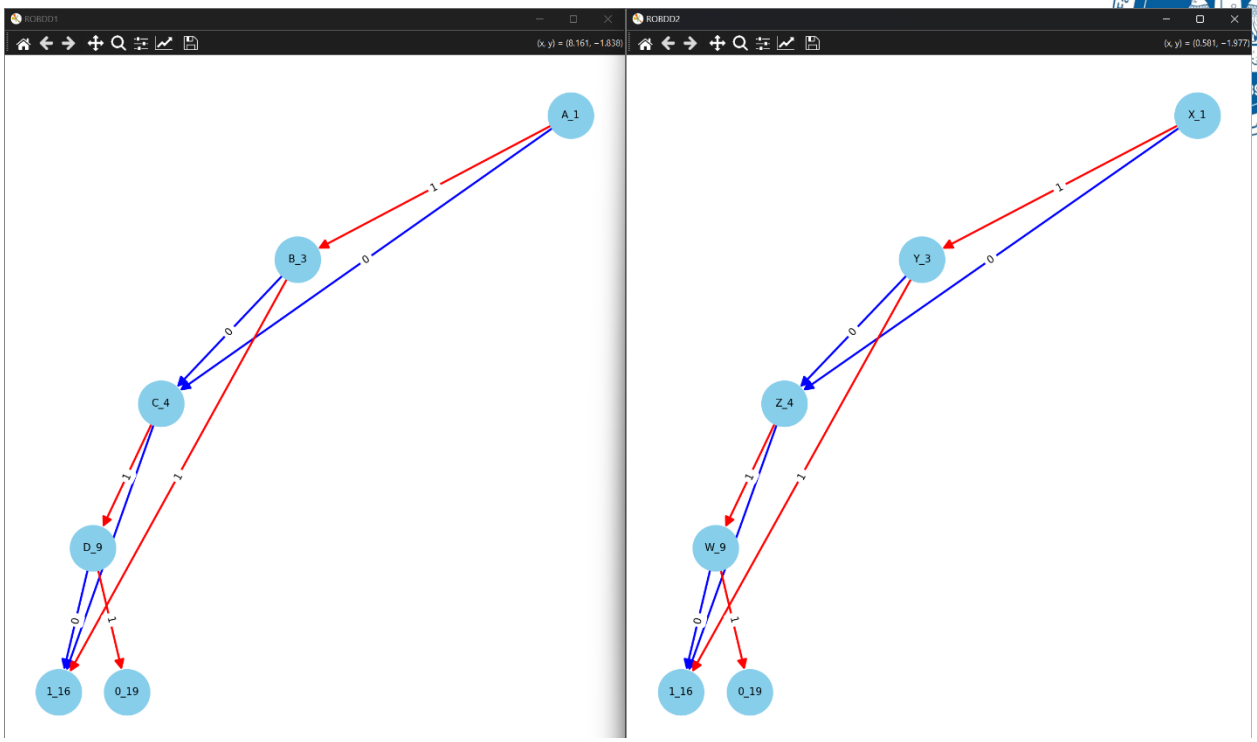
The output of this testcase is false which is the correct answer.



Here are the graphs:






ROBDD1
A_1
B_3
C_4
D_9
1_16
0_19
ROBDD2
X_1
Y_3
Z_4
W_9
1_16
0_19

G. ***Conclusion:***

In this project, we combined the theory of Boolean logic with practical tools to create a platform for analyzing and visualizing Boolean functions. Using Binary Decision Diagrams (BDD) and Reduced Ordered Binary Decision Diagrams (ROBDD), we provided efficient ways to represent and optimize these functions. The equivalence checker ensures accuracy in comparing Boolean functions, while the user-friendly GUI simplifies input and visualization processes. This project bridges complex logical concepts with practical applications, paving the way for future use in areas like digital design, verification, and automation.