Electronic Design Automation

Project 1

Name : Sudha Ravali Yellapantula

PID : 906109009

ROBDDs :

1. Lab Overview :

For this lab project, it was required to implement Reduced Ordered Binary Decision Diagrams ie, ROBDDs using any high level programming language. The language I chose to implement this project is JAVA.
The paper starts with the introduction of Boolean operators like "AND", "OR", "NOT", "Equivalent", "Implies " etc that were useful in creation and evaluation of Boolean expressions that we are to be represented on an ROBDD.

The truth tables for **not**, **and**, **or**, **implies** and **equivalent** respectively are :

	$\neg$
0	1
1	0

$\wedge$	0	1
0	0	0
1	0	1

$\vee$	0	1
0	0	1
1	1	1

$\Rightarrow$	0	1
0	1	1
1	0	1

$\Leftrightarrow$	0	1
0	1	0
1	0	1

The different Boolean expressions are evaluated by implementing the Boolean operator functions.

Reduced Ordered Binary Decision Diagram :

It has been proved by Cook that checking the satisfiability of Boolean expressions is NP-complete. Binary Decision Diagrams (BDDs) developed as a solution to this problem in the many practical cases. A BDD is a representation of a Boolean expression in the form of a directed acyclic graph

A binary decision diagram (BDD) is defined as a data structure that is used to represent a Boolean function. When the variables occur in the same orderings on all paths from the root of the Binary Tree, it is known as an OBDD. In the scenario when all identical shared nodes and the redundant tests are eliminated, the ordered BDD gets modified to an ROBDD.

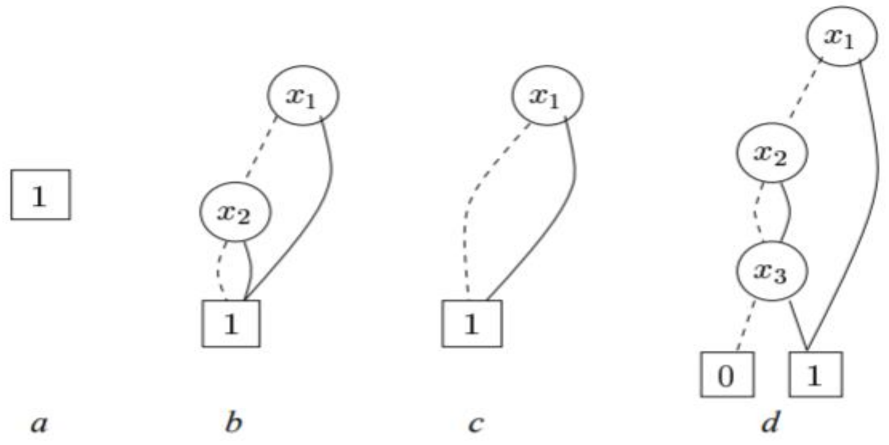
Hence, an OBDD is a Reduced OBDD if

- The nodes are Unique : The outgoing edges are given by two functions, low(u) and high (u). ie, both two distinct nodes u and v have the same variable name and low- and high successor. ie ,

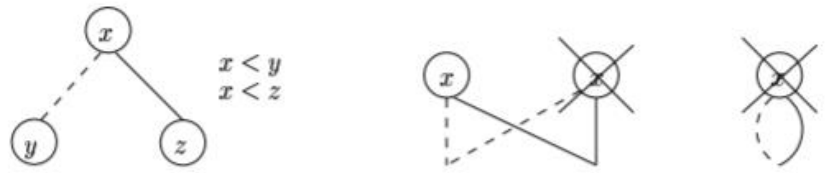
$$var(u) = var(v), low(u) = low(v), high(u) = high(v) \text{ implies } u = v,$$

- Presence of Non-Redundantt Tests : No variable node u has identical low- and high-successor i.e,

$$low(u) \neq high(u)$$



Four OBDDs: a) An OBDD for 1. b) Another OBDD for 1 with two redundant tests. c) Same as *b* with one of the redundant tests removed. d) An OBDD for $x_1 \vee x_3$ with one redundant test.



The ordering and reducedness conditions of ROBDDs. Left: Variables must be *ordered*. Middle: Nodes must be *unique*. Right: Only *non-redundant tests* should be present.

<u>Construction and Manipulation of ROBDDs :</u>

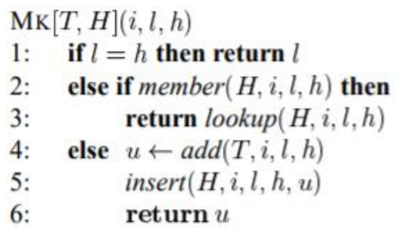
ROBDDs are canonical, meaning for any Boolean function f there is one and only one ROBDD that represents it.One way to think of a BDD is a tree composed of nodes that represent boolean variables. Such a tree would begin at some root node and grow down to one or both sinks

One of the ways to construct an ROBDD is by constructing an OBDD and then reducing it to an ROBDD. Nodes will be represented as numbers 0,1,2,3 …. with 0 and 1 reserved for the terminal nodes. The variables in the ordering x1< x2< …. < xn are represented by their indices 1, 2, 3, 4, ….n. The ROBDD is stored in a Table, T : u → (i,l,h) which maps a node ? to its three attributes, var(u) = i, low(u) = l, high(u) = h. The functions that are implemented are Mk, Build, Apply, Restrict, SatCount and AnySat.

<u>Mk:</u>

In order to make sure that the OBDD that is being constructed is getting reduced, given a a triple, (i,l,h), it is important to determine whether there exists a node in the tree with the same attributes like, , var(u) = i, low(u) = l, high(u) = h. For this purpose, we create a hash map H such that it is the inverse of the table, T ie, for variable nodes u, T(u) = (I,l,h), if and only if H(I, l,h) =u.

The algorithm implemented in Java is the following :



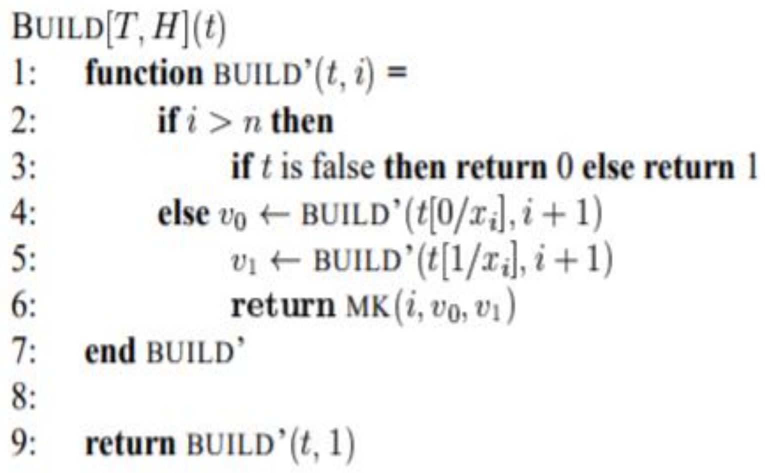
```
MK[$T, H$]($i, l, h$)
    if $l = h$ then return $l$
    else if $member(H, i, l, h)$ then
        return $lookup(H, i, l, h)$
    else $u \leftarrow add(T, i, l, h)$
        $insert(H, i, l, h, u)$
        return $u$
```

All the operations are assumed to be done under constant time t, ie, O(1). Here, the make function searches table H for a node with variable index i and low-, high-branches l, h and returns a matching node if one exists. If that is not the case, it creates a new node ⍰, inserts it into H and returns the identity of it.

Build :

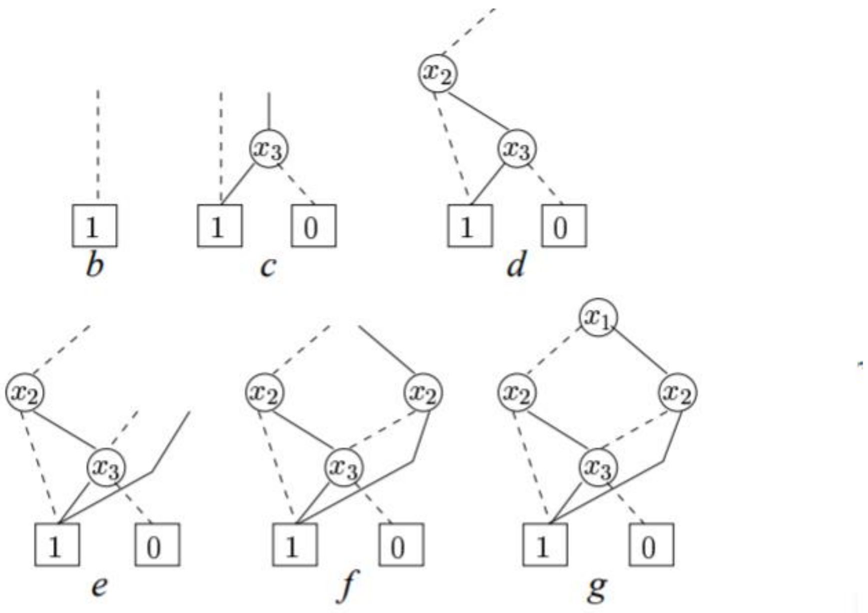
The Build function constructs and ROBDD for a Boolean expression t with variables in { x1, x2, …. Xn}. It does so by recursively constructing ROBDDs v0 and v1, and then going ahead to identify the node for t . If v0 and v1 are identical, or if there exits a same node with the same I, l, h values, no new node is created. The running time is not that great as with n variables, there will be 2^n calls generated.

The Algorithm implemented in Java is as follows :



```
BUILD[$T, H$]($t$)
    function BUILD'($t, i$) =
        if $i > n$ then
            if $t$ is false then return 0 else return 1
        else $v_0 \leftarrow$ BUILD'($t[0/x_i], i + 1$)
            $v_1 \leftarrow$ BUILD'($t[1/x_i], i + 1$)
            return MK($i, v_0, v_1$)
    end BUILD'

    return BUILD'($t, 1$)
```

<u>Diagramatic Representation :</u>

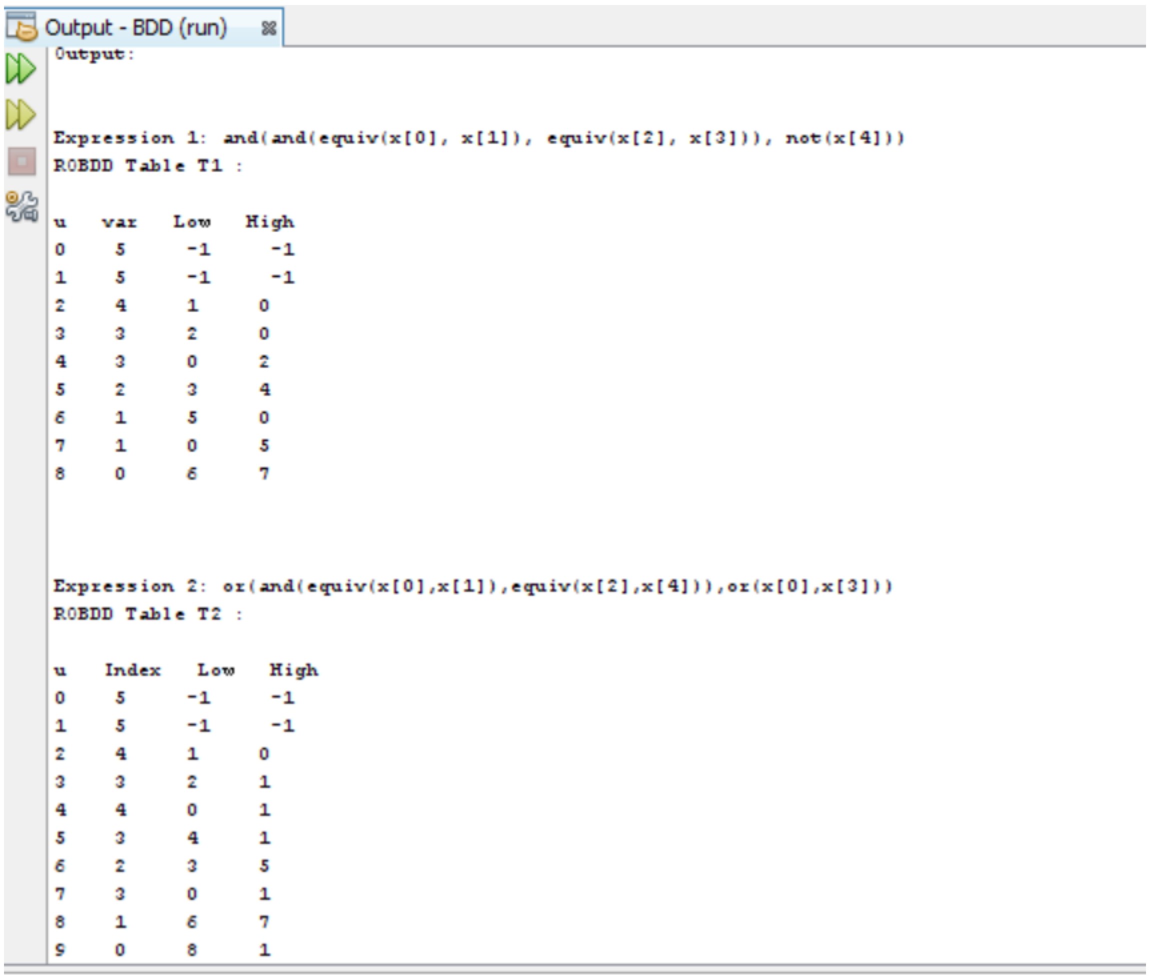


Using BUILD on the expression $(x_1 \Leftrightarrow x_2) \vee x_3$. (a) The tree of calls to BUILD. (b) The ROBDD after the call BUILD'$((0 \Leftrightarrow 0) \vee x_3, 3)$. (c) After the call BUILD'$((0 \Leftrightarrow 1) \vee x_3, 3)$. (d) After the call BUILD'$((0 \Leftrightarrow x_2) \vee x_3, 2)$. (e) After the calls BUILD'$((1 \Leftrightarrow 0) \vee x_3, 3)$ and BUILD'$((1 \Leftrightarrow 1) \vee x_3, 3)$. (f) After the call BUILD'$((1 \Leftrightarrow x_2) \vee x_3, 2)$. (g) The final result.

<u>Test Cases :</u>

Test Case 1 and 2 :

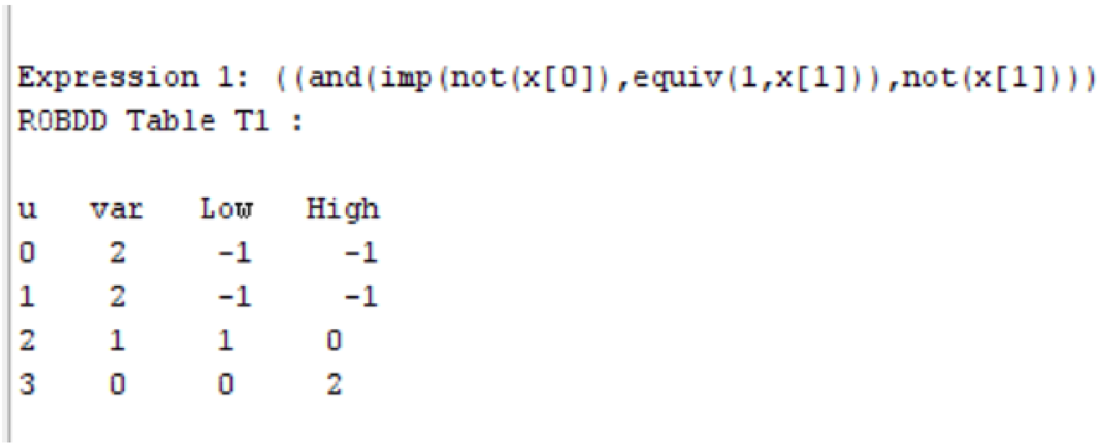
When the following two expressions are considered, the observed output for the build function is **:**



```
Output:


Expression 1: and(and(equiv(x[0], x[1]), equiv(x[2], x[3])), not(x[4]))
ROBDD Table T1 :

u    var    Low    High
0    5      -1      -1
1    5      -1      -1
2    4      1      0
3    3      2      0
4    3      0      2
5    2      3      4
6    1      5      0
7    1      0      5
8    0      6      7



Expression 2: or(and(equiv(x[0],x[1]),equiv(x[2],x[4])),or(x[0],x[3]))
ROBDD Table T2 :

u    Index    Low    High
0    5      -1      -1
1    5      -1      -1
2    4      1      0
3    3      2      1
4    4      0      1
5    3      4      1
6    2      3      5
7    3      0      1
8    1      6      7
9    0      8      1
```

Test Case 3 :

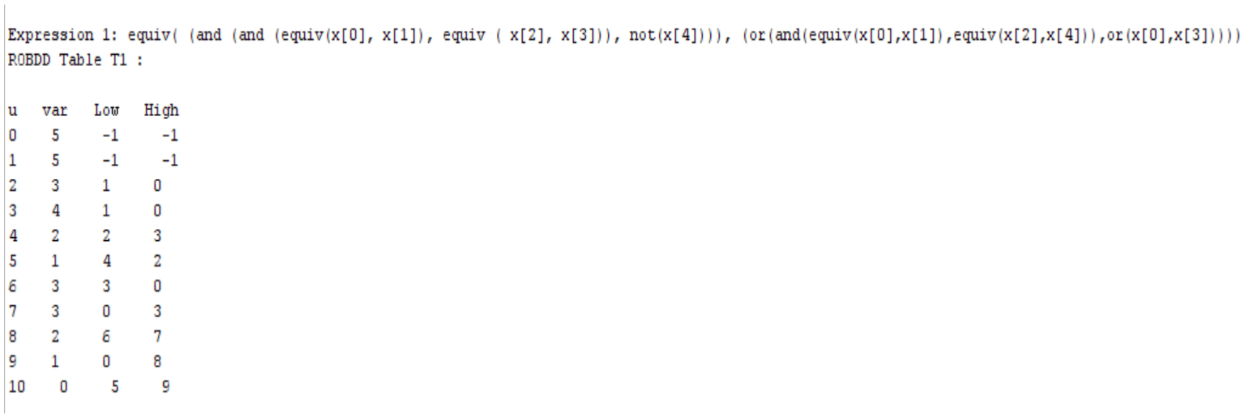
For the following test expression, the build table obtained is :



```
Expression 1: ((and(imp(not(x[0]),equiv(1,x[1])),not(x[1])))
ROBDD Table T1 :

u   var   Low   High
0    2     -1     -1
1    2     -1     -1
2    1     1     0
3    0     0     2
```

For large test case :

In case of large test expressions, such as the below one, the build tale obtained is as follows. The run time analysis of large test cases doesn't show any slowing down of time and is just as efficient.



```
Expression 1: equiv( (and (and (equiv(x[0], x[1]), equiv ( x[2], x[3])), not(x[4]))), (or(and(equiv(x[0],x[1]),equiv(x[2],x[4])),or(x[0],x[3]))))
ROBDD Table T1 :

u   var   Low   High
0    5     -1     -1
1    5     -1     -1
2    3     1     0
3    4     1     0
4    2     2     3
5    1     4     2
6    3     3     0
7    3     0     3
8    2     6     7
9    1     0     8
10    0     5     9
```

3. Apply

Every binary operator on ROBDDs is implemented by the same general algorithm Apply(op, u1,u2) where two ROBDDS compute the ROBDD for the Boolean Expression $t^{u1}\ op\ t^{u2}$. The construction of Apply is based on the Shannon Expansion :

$$t \ = \ x \rightarrow t[1/x], t[0/x]$$

If we start from the root of the two ROBDDs we can construct the ROBDD of the result by recursively constructing the low- and the high-branches and then form the new root from these. Again, to ensure that the result is reduced, we create the node through a call to MK. If both u1 and u2 are terminal, a new terminal node is computed having the value of op applied to the two truth values.

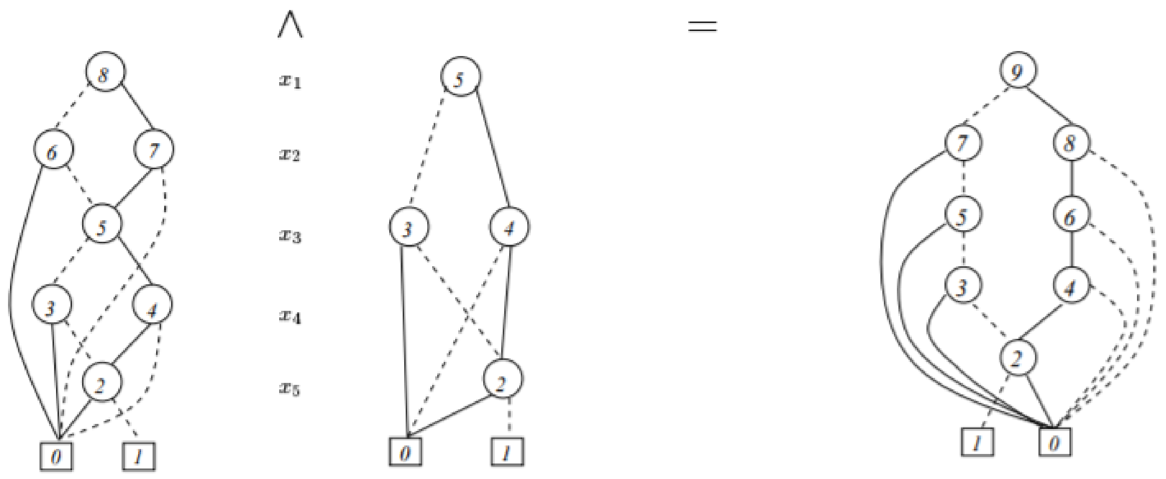
If at least one of u1 and u2 are non-terminal, we proceed according to the variable index. If the nodes have the same index, the two low-branches are paired and APP recursively computed on them. Similarly for the high-branches.

Since the indices of the terminals are to be one larger than the index of the non-terminals, in the last two cases, var(u1) < var(u2) and var(u1) > var(u2), take account of the situations where one of the nodes is a terminal.

Algorithm for Apply :

```
APPLY[$T$, $H$]($op$, $u_1$, $u_2$)
1:
2:
3:  function APP($u_1$, $u_2$) =
4:
5:    else if $u_1 \in \{0, 1\}$ and $u_2 \in \{0, 1\}$ then $u \leftarrow op(u_1, u_2)$
6:    else if $var(u_1) = var(u_2)$ then
7:          $u \leftarrow$ MK($var(u_1)$, APP($low(u_1)$, $low(u_2)$), APP($high(u_1)$, $high(u_2)$))
8     else if $var(u_1) < var(u_2)$ then
9           $u \leftarrow$ MK($var(u_1)$, APP($low(u_1)$, $u_2$), APP($high(u_1)$, $u_2$))
10:   else (* $var(u_1) > var(u_2)$ *)
11:         $u \leftarrow$ MK($var(u_2)$, APP($u_1$, $low(u_2)$), APP($u_1$, $high(u_2)$))
12:
13:   return $u$
14: end APP
15:
16: return APP($u_1$, $u_2$)
```

Diagramatic Representation for Apply :

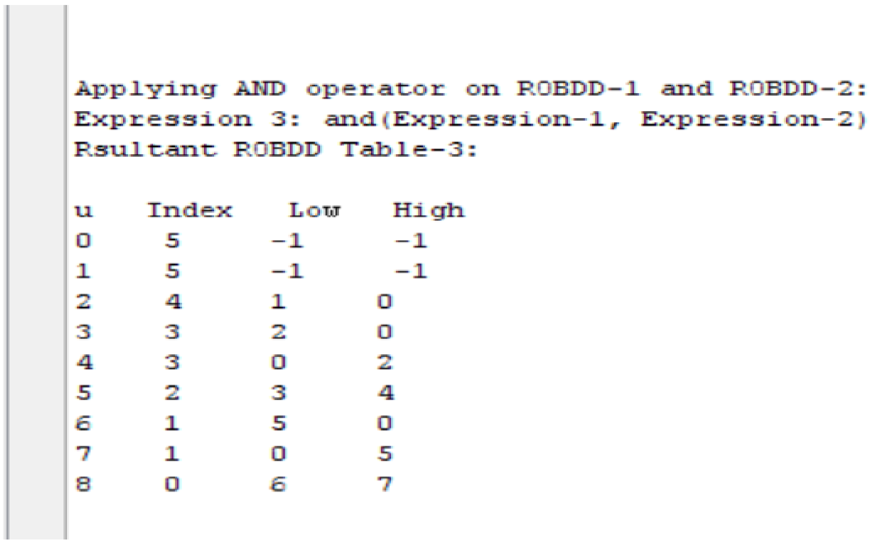


Test Cases :

Some of the **test cases** of Build function are as follows for the following Boolean Expression :

Apply Table obtained for Boolean expressions 1 and 2 as described above (Refer Test Case 1 and 2 in Build)

The solution obtained when "AND" operator is used between the two above expressions.



```
Applying AND operator on ROBDD-1 and ROBDD-2:
Expression 3: and(Expression-1, Expression-2)
Rsultant ROBDD Table-3:

u   Index   Low   High
0    5      -1     -1
1    5      -1     -1
2    4      1      0
3    3      2      0
4    3      0      2
5    2      3      4
6    1      5      0
7    1      0      5
8    0      6      7
```

Restrict :

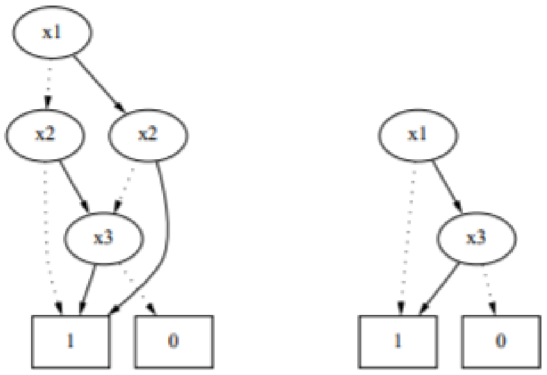
The next operation that has been implemented is Restriction of an ROBDD u. In this function, given a truth assignment [0/x3, 1/x5, 1/x6], the ROBDD of t^u is to be computed within this restriction. Here, one of the variable is forced to take up a Boolean value of either true or false, and is eliminated all together resulting a change in the ROBDD because of it.

The algorithm again uses MK to ensure that the resulting OBDD is reduced. Intuitively, in computing RESTRICT (u,j,b), we search for all nodes with var = j and replace them by their low- or high-son depending on b. Since this might force nodes above the point of replacement to become equal, it is followed by a reduction.Due to the two recursive calls, the algorithm has exponential running time.

<u>Algorithm for Restrict :</u>

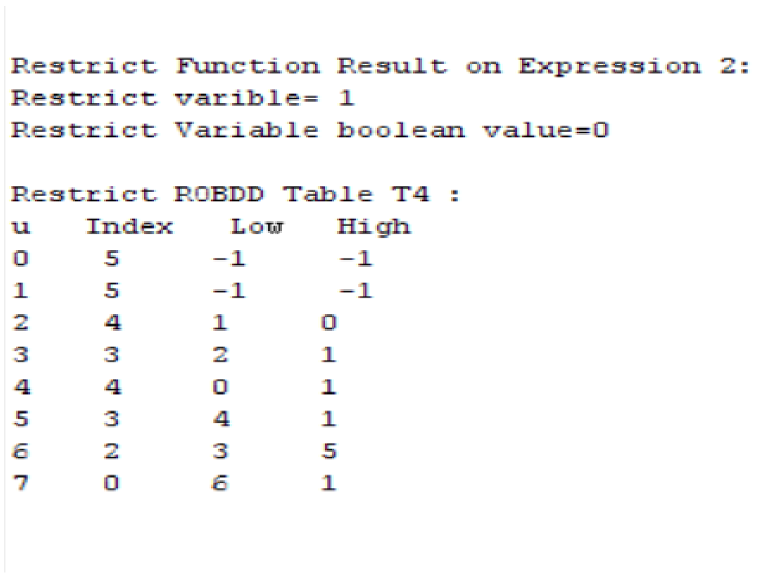
```
RESTRICT$[T, H](u, j, b)$ =
1: function $res(u)$ =
2:     if $var(u) > j$ then return $u$
3:     else if $var(u) < j$ then return MK$(var(u), res(low(u)), res(high(u)))$
4:     else (* $var(u) = j$ *) if $b = 0$ then return $res(low(u))$
5:     else (* $var(u) = j, b = 1$ *) return $res(high(u))$
6: end $res$
7: return $res(u)$
```

Diagramatic Representation for the Boolean Expression (x1 ⇔ x2) | x3 with truth assignment [0/x2].



<u>Some Test Cases for Restrict :</u>

Restrict Solution obtained for test expression 2 is observed as below. Here, the x1 variable is forced to have the value of 0, as a result x1 is restricted from being present in the ROBDD. This explanation can be verified from the below table.



```
Restrict Function Result on Expression 2:
Restrict varible= 1
Restrict Variable boolean value=0

Restrict ROBDD Table T4 :
u    Index    Low    High
0     5      -1      -1
1     5      -1      -1
2     4      1      0
3     3      2      1
4     4      0      1
5     3      4      1
6     2      3      5
7     0      6      1
```

<u>SatCount :</u>

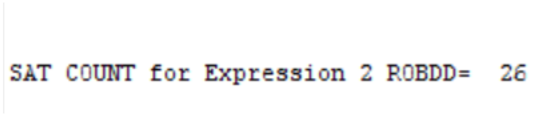
All the satisfying assignments are stored in sat(u). In this algorithms, the size of the sat(u) is determined, ie, it determines the number of valid truth assignments. The algorithm banks on the fact that for a node u, the variable index, var(u) now has two satisfying assignments to make f^u true. The first set has varu equal to 0, the other has varu equal to 1 . For the first set, the number is found by finding the number of truth assignments count count(low(u)) making low(u) true. Therefore in the case of varu being 0, a total of $2^{var(low(u)) - varu - 1}$ *count(low(u)) satisfying assignments exists.

<u>Algorithm Implemented :</u>

```
SATCOUNT[T](u)
1:  function count(u)
2:      if $u = 0$ then res ← 0
3:      else if $u = 1$ then res ← 1
4:      else res ← $2^{var(low(u))-var(u)-1} * count(low(u))$
                  $+ 2^{var(high(u))-var(u)-1} * count(high(u))$
5:      return res
6:  end count
7:
8:  return $2^{var(u)-1} * count(u)$
```

A test case for SatCount :

Here the number of satisfiability cases for the test expression 2 are being calculated. For the test expression 2, there are 5 variable, starting from x0 to x4 and the possible combinations to obtain a satisfiable condition is 26.

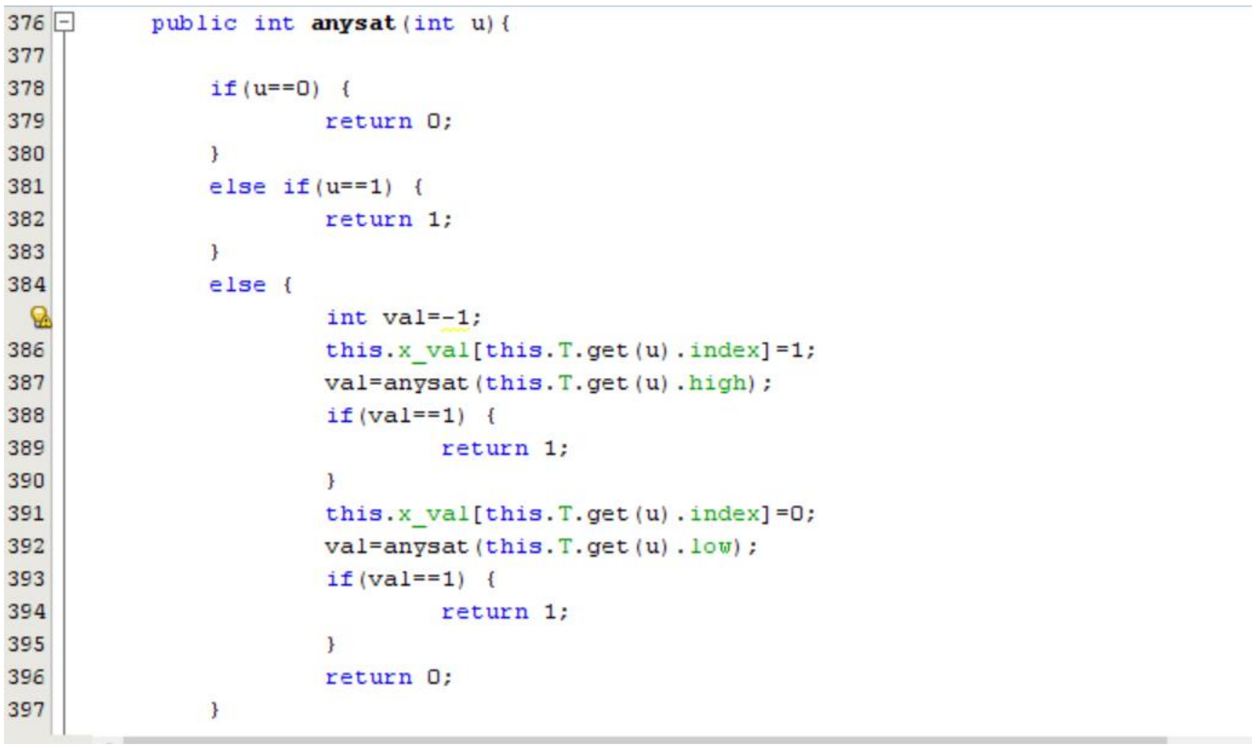


```
SAT COUNT for Expression 2 ROBDD=  26
```

AnySat :

The next algorithm ANYSAT finds a satisfying truth assignment. Some irrelevant variables present in the ordering might not appear in the result and they can be assigned any value whatsoever. ANYSAT simply finds a path leading to 1 by a depth-first traversal, prefering somewhat arbitrarily low-edges over high-edges. It is particularly simple due to the observation that if a node is not the terminal 0, it has at least one path leading to 1. The running time is clearly linear in the result.

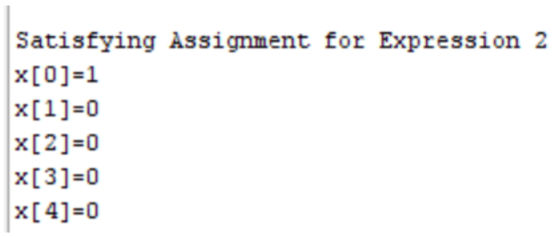
Algorithm employed – code snippet :



```
    public int anysat(int u){

        if(u==0) {
                return 0;
        }
        else if(u==1) {
                return 1;
        }
        else {
                int val=-1;
                this.x_val[this.T.get(u).index]=1;
                val=anysat(this.T.get(u).high);
                if(val==1) {
                        return 1;
                }
                this.x_val[this.T.get(u).index]=0;
                val=anysat(this.T.get(u).low);
                if(val==1) {
                        return 1;
                }
                return 0;
        }
```

Example Test Case for AnySat :

For the given test expression 2, there are 5 variables ranging from 0 to 4. The Boolean expression will be validated as true if and only if there is a presence of satisfying assignment of all five variables. One such satisfying assignment is as follows :



```
Satisfying Assignment for Expression 2
x[0]=1
x[1]=0
x[2]=0
x[3]=0
x[4]=0
```

TEST CASES USED :

In the program, initially two ROBDD tables are created using Make and Build functions for the two given test expressions. Using the first two expressions, a third ROBDD gets created which is the apply table. We parsed the two ROBDDDs generated from above as parameters to the Apply function to generate the resultant ROBDD. We are performing the AND operation on the two ROBDDs. For the function SatCount, AnySat and Restrict, ROBDD -2 , ie ROBDD generated from the second expression is used to evaluate these functions. The test cases seen below show one complete execution of the program.

1. TEST CASE 1 for all functions :

Output:

Expression 1: and(and(equiv(x[0], x[1]), equiv(x[2], x[3])), not(x[4]))
ROBDD Table T1 :

```
u  var  Low  High
0  5    -1   -1
1  5    -1   -1
2  4    1    0
3  3    2    0
4  3    0    2
5  2    3    4
6  1    5    0
7  1    0    5
8  0    6    7
```

Expression 2: or(and(equiv(x[0],x[1]),equiv(x[2],x[4])),or(x[0],x[3]))
ROBDD Table T2 :

```
u  Index  Low  High
0   5    -1    -1
1   5    -1    -1
2   4     1     0
3   3     2     1
4   4     0     1
5   3     4     1
6   2     3     5
7   3     0     1
8   1     6     7
9   0     8     1
```

Applying AND operator on ROBDD-1 and ROBDD-2:
Expression 3: and(Expression-1, Expression-2)
Resultant ROBDD Table-3:

```
u  Index  Low  High
0   5    -1    -1
1   5    -1    -1
2   4     1     0
3   3     2     0
4   3     0     2
5   2     3     4
6   1     5     0
7   1     0     5
8   0     6     7
```

Restrict Function Result on Expression 2:
Restrict variable= 1
Restrict Variable boolean value=0

Restrict ROBDD Table T4 :

```
u  Index  Low  High
0   5    -1    -1
```

```
1   5   -1   -1
2   4   1    0
3   3   2    1
4   4   0    1
5   3   4    1
6   2   3    5
7   0   6    1
```

```
SAT COUNT for Expression 2 ROBDD=  26
```

```
Satisfying Assignment for Expression 2
x[0]=1
x[1]=0
x[2]=0
x[3]=0
x[4]=0
```

2. TEST CASE 2 for all functions

Expression 1: and(imp(not(x[0]),equiv(1,x[1])),not(x[2]))

ROBDD Table T1 :

```
u  var  Low  High
0   3   -1   -1
1   3   -1   -1
2   2   1    0
3   1   0    2
4   0   3    2
```

Expression 2: (or((equiv(x[0],x[1])), x[2]))
ROBDD Table T2 :

```
u  Index  Low  High
0   3     -1   -1
1   3     -1   -1
```

```
2   2    0    1
3   1    1    2
4   1    2    1
5   0    3    4



Applying AND operator on ROBDD-1 and ROBDD-2:
Expression 3: and(Expression-1, Expression-2)
Resultant ROBDD Table-3:

u   Index   Low   High
0   3    -1    -1
1   3    -1    -1
2   2    1    0
3   2    0    2
4   0    0    3



Restrict Function Result on Expression 2:
Restrict variable= 1
Restrict Variable boolean value=0

Restrict ROBDD Table T4 :
u   Index   Low   High
0   3    -1    -1
1   3    -1    -1
2   2    0    1
3   0    1    2



SAT COUNT for Expression 2 ROBDD=  6



Satisfying Assignment for Expression 2
x[0]=1
x[1]=1
x[2]=0
```

3. TEST CASE 3 for all functions :

For large test cases :

Output:

Expression 1: equiv((and (and (equiv(x[0], x[1]), equiv (x[2], x[3])), not(x[4]))), (or(and(equiv(x[0],x[1]),equiv(x[2],x[4])),or(x[0],x[3])))

ROBDD Table T1 :

```
u   var   Low   High
0    5    -1    -1
1    5    -1    -1
2    3     1     0
3    4     1     0
4    2     2     3
5    1     4     2
6    3     3     0
7    3     0     3
8    2     6     7
9    1     0     8
10   0     5     9
```

Expression 2: and(equiv((or (and (equiv(x[0], x[1]), equiv (x[2], x[3])), not(x[4]))), (or(and(equiv(x[0],x[1]),equiv(x[2],x[4])),or(x[0],x[3])))),or((equiv(x[0],x[1])), x[2]))

ROBDD Table T2 :

```
u   Index   Low   High
0    5      -1    -1
1    5      -1    -1
2    4       1     0
3    3       0     1
4    2       2     3
5    4       0     1
6    3       5     2
7    2       0     6
8    1       4     7
9    2       0     2
10   3       1     2
```

11	3	2	1
12	2	10	11
13	1	9	12
14	0	8	13

Applying AND operator on ROBDD-1 and ROBDD-2:
Expression 3: and(Expression-1, Expression-2)
Resultant ROBDD Table-3:

u	Index	Low	High
0	5	-1	-1
1	5	-1	-1
2	5	1	0
3	3	2	0
4	4	1	0
5	4	0	4
6	2	3	5
7	5	0	1
8	3	7	0
9	3	0	8
10	1	6	9
11	3	4	0
12	3	0	4
13	2	11	12
14	1	0	13
15	0	10	14

Restrict Function Result on Expression 2:
Restrict variable= 4
Restrict Variable boolean value=0

Restrict ROBDD Table T4 :

u	Index	Low	High
0	5	-1	-1
1	5	-1	-1
2	3	0	1
3	2	1	2
4	2	0	2
5	1	3	4
6	2	0	1

```
7   1   6   1
8   0   5   7
```

```
SAT COUNT for Expression 2 ROBDD=  14
```

```
Satisfying Assignment for Expression 2
x[0]=1
x[1]=1
x[2]=1
x[3]=1
x[4]=0
```

Conclusion :

From the above description and examples, it can be seen that the code correctness can be verified easily as the program generates the output functions accurately. All the possible test cases were verified and the run time analysis of the large test cases also proved to be efficient and gave positive results.