ROBDD(Reduced Ordered Binary Decision Diagram)

Creating a ROBDD using Make and Build

Make Function

- This is neccesary to reduce the tree while it is made.
- It takes input node in the form of the (i,l,h) tuple and returns exisiting node or else creates a new one.

```
node(u)<-make(self,i,l,h)
```

Build Function

- Recursive function which traverses through through your expression to create the Table for ROBDD
- Uses make to create or check for exisiting nodes
- Initialises with i=0 for the first node.

```
build(self,i=0)
```

In [1]:
```
from ROBDD import ROBDD
import time
```

In [2]:
```
# Quick and Easy print Table Function for ROBDDs
def printROBDD(ROBDD):
    print('============================')
    print('|  u  |  i  |   l  |   h  |')
    print('============================')
    n = ROBDD.nNodes
    for idx in range(0,n):
        node = ROBDD.T[idx]
        print('   '+str(idx)+'     '+str(node[0])+'     '+str(node[1])+'     '+str(node[2])+'
 ')
        print('----------------------------')
```

Test Case 1 - Simple Test Case

Create a ROBDD with 3 variables and the following equation:-

```
or(equiv(x0,x1),x2)
```

In [3]:
```
start = time.clock()
ROBDD1 = ROBDD(nVars = 3,switch = 0)
print('Time Spent')
time.clock() - start
```

```
Time Spent
```

Out[3]:
```
0.00038153053690020896
```

In [4]:
```
# Display the table created by the ROBDD
printROBDD(ROBDD1)
```

```
============================
|  u  |  i  |   l  |   h  |
============================
   0     4     -1     -1
----------------------------
   1     4     -2     -2
----------------------------
   2     2     0     1
----------------------------
   3     1     1     2
----------------------------
   4     1     2     1
----------------------------
   5     0     3     4
----------------------------
```

Test Case 2 - Testing all the functions (and,or,implies,equiv and not)

Create a ROBDD with 3 variables and the following equation:-

```
and(implies(not(x0),equiv(1,x1)),not(x2))
```

In [5]:
```
start = time.clock()
ROBDD2 = ROBDD(3,switch=2)
print('Time Spent')
time.clock() - start
```

```
Time Spent
```

Out[5]:
```
0.00038646716122239105
```

In [6]:
```
printROBDD(ROBDD2)
```

```
============================
|  u  |  i  |   l  |   h  |
============================
   0     4     -1     -1
----------------------------
   1     4     -2     -2
----------------------------
   2     2     1     0
----------------------------
   3     1     0     2
----------------------------
   4     0     3     2
----------------------------
```

Test Case 3 - Increasing the Number of Variables

Create a ROBDD with 8 variables and the following equation:-

```
and(and(and(x0,x1),and(x2,x3)),and(and(x4,x5),and(x6,x7)))
```

In [7]:
```
start = time.clock()
ROBDD3 = ROBDD(8,switch=3)
print('Time Spent')
time.clock() - start
```

```
Time Spent
```

Out[7]:
```
0.0009256170604094727
```

In [8]:
```
printROBDD(ROBDD3)
```

```
============================
|  u  |  i  |   l  |   h  |
============================
   0     9     -1     -1
----------------------------
   1     9     -2     -2
----------------------------
   2     7     0     1
----------------------------
   3     6     0     2
----------------------------
   4     5     0     3
----------------------------
   5     4     0     4
----------------------------
   6     3     0     5
----------------------------
   7     2     0     6
----------------------------
   8     1     0     7
----------------------------
   9     0     0     8
----------------------------
```

Testing the SatCount and AnySat

AnySat Function

- Returns one satisfying condition which makes the given expression true.
- Recursive function, which tries to reach the Node 1 and then returns the satisfying condition

```
list(x0,x1,x2....)<-anySat(self)
```

SatCount Function

- Finds total number of satisfying conditions for a given expression
- Recursive function which traverses through through the nodes and finds all possible paths to Node 1

```
count<-satCount(self)
```

In [10]:
```
from ROBDD import ROBDD
import time
```

Test Case 1 - Simple Test Case

Create a ROBDD with 3 variables and the following equation:-

```
or(equiv(x0,x1),x2)
```

In [4]:
```
start = time.clock()
ROBDD1 = ROBDD(nVars = 3,switch = 0)
count = ROBDD1.satCount()
print('Count of Satisfying Conditions is (SatCount) -> '+str(count))
anySatX = ROBDD1.anySat(None)
print('One of the satisfying conditions is (AnySat) -> '+str(anySatX))
print('Time Spent')
time.clock() - start
```

```
Count of Satisfying Conditions is (SatCount) -> 6.0
One of the satisfying conditions is (AnySat) -> [0, 0, -1]
Time Spent
```

Out[4]:
```
0.0009443057096291678
```

Test Case 2 - Increasing the Number of Variables

Create a ROBDD with 8 variables and the following equation:-

```
and(and(and(x0,x1),and(x2,x3)),and(and(x4,x5),and(x6,x7)))
```

In [5]:
```
start = time.clock()
ROBDD2 = ROBDD(nVars = 8,switch = 3)
count = ROBDD2.satCount()
print('Count of Satisfying Conditions is (SatCount) -> '+str(count))
anySatX = ROBDD2.anySat(None)
print('One of the satisfying conditions is (AnySat) -> '+str(anySatX))
print('Time Spent')
time.clock() - start
```

```
Count of Satisfying Conditions is (SatCount) -> 1.0
One of the satisfying conditions is (AnySat) -> [1, 1, 1, 1, 1, 1, 1, 1]
Time Spent
```

Out[5]:
```
0.0011865529174457379
```

Test Case 3 - Negative Test Case

This test case check when no condition satisfies the expression Create a ROBDD with 1 variables and the following equation:-

```
and(0,x0)
```

In [13]:
```
start = time.clock()
ROBDD3 = ROBDD(nVars = 1,switch = 4)
count = ROBDD3.satCount()
print('Count of Satisfying Conditions is (SatCount) -> '+str(count))
anySatX = ROBDD3.anySat(None)
print('One of the satisfying conditions is (AnySat) -> '+str(anySatX))
print('Time Spent')
time.clock() - start
```

```
Error ! :(
One of the satisfying conditions is (AnySat) -> [-1]
Time Spent
```

Out[13]:
```
0.0004442961890163133
```

ROBDD Restict

Restrict Function

- This is used to reduce the Expression by setting value of one of the variables to a fixed one.
- The function takes j as variable name (i.e x[j]) and set it to a value 'b' (x[j]=b) and returns a new table

```
Table<-restrict(self,node=None,j=0,b=0)
```

In [1]:
```
from ROBDD import ROBDD
import time
```

In [2]:
```
# Quick and Easy print Table Function for ROBDDs
def printROBDD(table,n):
    print('============================')
    print('|  u  |  i  |   l  |   h  |')
    print('============================')
    for idx in range(0,n):
        node = table[idx]
        print('   '+str(idx)+'     '+str(node[0])+'     '+str(node[1])+'     '+str(node[2])+'
 ')
        print('--------------------------')
```

Test Case 1 - Simple Test Case

Create a ROBDD with 3 variables and the following equation:-

```
or(equiv(x0,x1),x2)
```

In [3]:
```
start = time.clock()
ROBDD1 = ROBDD(nVars = 3,switch = 0)
print('Time Spent')
time.clock() - start
```

```
Time Spent
```

Out[3]: 0.00033533783788548864

In [4]:
```
printROBDD(ROBDD1.T,ROBDD1.nNodes)
```

```
============================
|  u  |  i  |   l  |   h  |
============================
   0     4     -1     -1
--------------------------
   1     4     -2     -2
--------------------------
   2     2     0     1
--------------------------
   3     1     1     2
--------------------------
   4     1     2     1
--------------------------
   5     0     3     4
--------------------------
```

We will now restrict x[1] = 0

In [5]:
```
start = time.clock()
ROBDD1.restrict(None,j=1,b=0)
print('Time Spent')
time.clock() - start
```

```
Time Spent
```

Out[5]: 0.00019852282095639273

In [6]:
```
printROBDD(ROBDD1.T_,ROBDD1.nNodes_)
```

```
============================
|  u  |  i  |   l  |   h  |
============================
   0     4     -1     -1
--------------------------
   1     4     -2     -2
--------------------------
   2     2     0     1
--------------------------
   3     0     1     2
--------------------------
```

Test Case 2 - Increasing the Number of Variables

Create a ROBDD with 8 variables and the following equation:-

```
and(and(and(x0,x1),and(x2,x3)),and(and(x4,x5),and(x6,x7)))
```

In [7]:
```
start = time.clock()
ROBDD2 = ROBDD(nVars = 8,switch = 3)
print('Time Spent')
time.clock() - start
```

```
Time Spent
```

Out[7]: 0.001516601514979482

In [8]:
```
printROBDD(ROBDD2.T,ROBDD2.nNodes)
```

```
============================
|  u  |  i  |   l  |   h  |
============================
   0     9     -1     -1
--------------------------
   1     9     -2     -2
--------------------------
   2     7     0     1
--------------------------
   3     6     0     2
--------------------------
   4     5     0     3
--------------------------
   5     4     0     4
--------------------------
   6     3     0     5
--------------------------
   7     2     0     6
--------------------------
   8     1     0     7
--------------------------
   9     0     0     8
--------------------------
```

We will now restrict x[1] = 0 and x[2] = 1

In [9]:
```
start = time.clock()
ROBDD2.restrict(None,j=1,b=0)
print('Time Spent')
time.clock() - start
#ROBDD2.restrict(None,j=2,b=1)
```

```
Time Spent
```

Out[9]: 0.0010909939752026254

In [10]:
```
printROBDD(ROBDD2.T_,ROBDD2.nNodes_)
```

```
============================
|  u  |  i  |   l  |   h  |
============================
   0     9     -1     -1
--------------------------
   1     9     -2     -2
--------------------------
   2     7     0     1
--------------------------
   3     6     0     2
--------------------------
   4     5     0     3
--------------------------
   5     4     0     4
--------------------------
   6     3     0     5
--------------------------
   7     2     0     6
--------------------------
```

ROBDD Apply

class Apply

- class Apply inherits from the ROBDD.

```
class Apply_ROBDD(ROBDD)
```

function Apply

- This recursive function builds a new table, taking an operation op as input with two different ROBDDS r1,r2

```
ROBDD<-apply(self,op,r1,r2)
```

In [1]:
```
from ROBDD import ROBDD,Apply_ROBDD
import time
```

In [2]:
```
# Quick and Easy print Table Function for ROBDDs
def printROBDD(ROBDD):
    print('============================')
    print('|  u  |  i  |   l  |   h  |')
    print('============================')
    n = ROBDD.nNodes
    for idx in range(0,n):
        node = ROBDD.T[idx]
        print('   '+str(idx)+'     '+str(node[0])+'     '+str(node[1])+'     '+str(node[2])+'
')
        print('----------------------------')
```

Test Case 1 - Simple Test Case

Create a ROBDDs with 3 variables and the following equations:-

ROBDD1 = `or(equiv(x0,x1),x2)`

ROBDD2 = `equiv(and(x0,x1),x2)`

operation = `and`

In [3]:
```
ROBDD1 = ROBDD(3,0)
ROBDD2 = ROBDD(3,1)
```

In [4]:
```
start = time.clock()
ROBDD_Applied = Apply_ROBDD(3)
ROBDD_Applied.apply('and',ROBDD1,ROBDD2)
time.clock() - start
```

Out[4]:
```
0.0001611455225170014
```

In [5]:
```
printROBDD(ROBDD_Applied)
```

```
============================
|  u  |  i  |   l  |   h  |
============================
   0     4     -1     -1
----------------------------
   1     4     -2     -2
----------------------------
   2     2     1     0
----------------------------
   3     1     2     0
----------------------------
   4     2     0     1
----------------------------
   5     1     0     4
----------------------------
   6     0     3     5
----------------------------
```

Test Case 2 - Increasing number of variables and testing with different number of variables

Create ROBDDs with different number of variables

ROBDD1 = `and(or(equiv(x0,x1),x2),x3)`

ROBDD2 = `equiv(and(x0,x1),x2)`

operation = `or`

In [15]:
```
ROBDD1 = ROBDD(4,0)
ROBDD2 = ROBDD(3,1)
```

In [16]:
```
start = time.clock()
ROBDD_Applied = Apply_ROBDD(4)
ROBDD_Applied.apply('and',ROBDD1,ROBDD2)
time.clock() - start
```

Out[16]:
```
0.0001734870833161039
```

In [17]:
```
printROBDD(ROBDD_Applied)
```

```
============================
|  u  |  i  |   l  |   h  |
============================
   0     5     -1     -1
----------------------------
   1     5     -2     -2
----------------------------
   2     2     1     0
----------------------------
   3     1     2     0
----------------------------
   4     2     0     1
----------------------------
   5     1     0     4
----------------------------
   6     0     3     5
----------------------------
```

ROBDD - Analysis

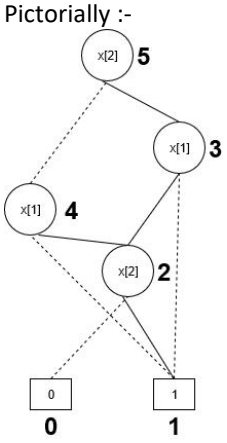
I have implemented the functions associated with the ROBDD using Python 3.6.3 on the PyCharms IDE. The implementation involved manual generation of expressions and encoding them in the ROBDDs.

Following are some test cases which check the thoroughness of the code.

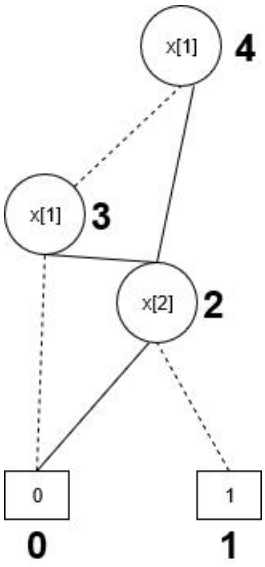
1. Build and Make

   A. Simple Test Case -
   or(equiv(x1,x2),x3)

   Pictorially :-



   B. Testing all functions-
   and(implies(not(x0),equiv(1,x1)),not(x2))
   Pictorially ->



   C. Testing by increasing the number of variables
   and(and(and(x0,x1),and(x2,x3)),and(and(x4,x5),and(x6,x7)))

   Note : The time taken for this operation increases considerably due to recursion.

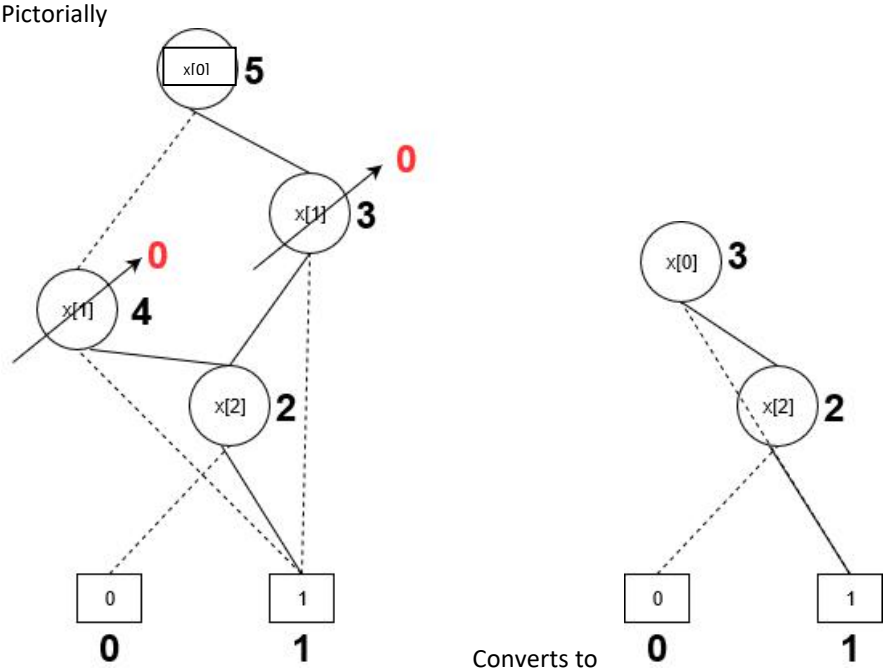
2. SatCount & AnySat

Time Analysis - > Since AnySat is a recursive function, the ROBDD with the larger table takes more time, since the recursion is deeper there.

3. Restrict

Test Case 1 : Using the Simple Test Case1 -
or(equiv(x1,x2),x3)
Restricting x2 = 0

Pictorially



Test Case 2: Increasing the number of variables and number of restricts.

Note : The time taken for this operation increases considerably due to deeper recursion.

4. Apply

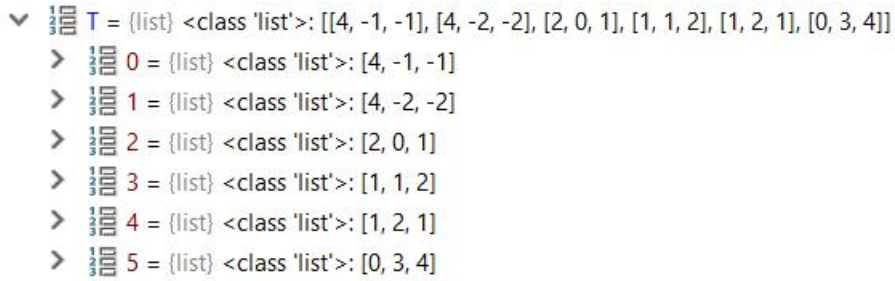
I have used the following expression to test Apply

ROBDD1 - >or(equiv(x1,x2),x3)

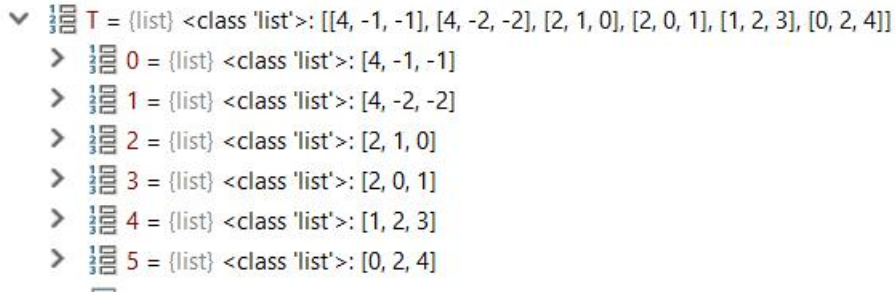
ROBDD2 - >equiv(and(x1,x2),x3)

Operator used - > and

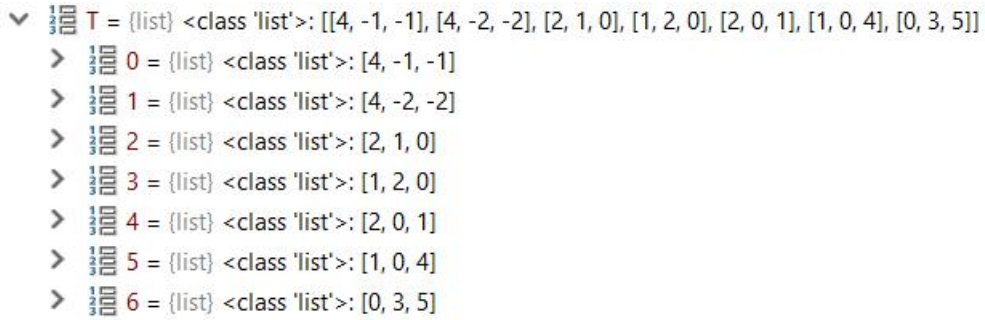
ROBDD1 ->



ROBDD2 ->



Apply(ROBDD1,ROBDD,'and') - >



Parser ::

I had also built an expression parser to get the expression from a string, but was not able to completely integrate it in the ROBDD code.

Expression - >

```
test = 'and(and(1,2),and(3,4))'
```