BDD Package Methods Documentation

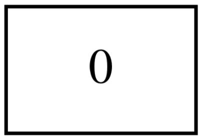
Arvind Kumar
Wagatsuma Laboratory
Kyushu Institute of Technology

2021

- (In example: g is global variable) g = graphviz.Digraph('G', filename='plot.gv'))

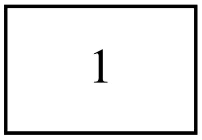
1 Example 1 (Zero Node)

```
if __name__ == '__main__':
    zero = Node(-1,None,None,'0')
    zero.plot()
    g.view()
```



2 Example 2 (One Node)

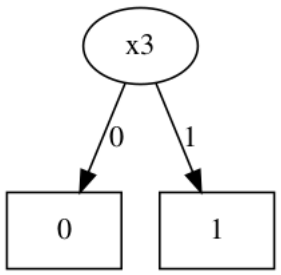
```
if __name__ == '__main__':
    one = Node(-1,None,None,'1')
    one.plot()
    g.view()
```



3 Example 3 (Variable Node)

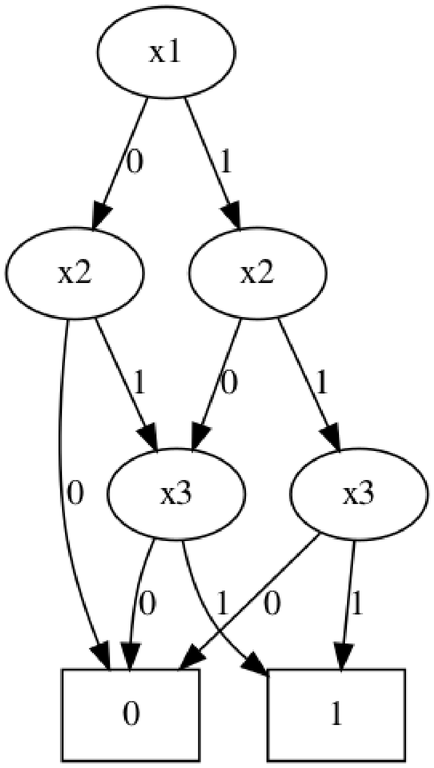
- index – level of the variable on the treee.
- low – low child of any node.
- high - high child of any node. (zero and one have low, high as None)
- X – denotes it's a variable. zero and one have value '0' and '1' respectively.

```
if __name__ == '__main__':
    zero = Node(-1,None,None,'0')
    one = Node(-1,None,None,'1')

    x3 = Node.createVariable(3,zero,one,'X')     #(index, low, high, X)
    x3.plot()
    g.view()
```



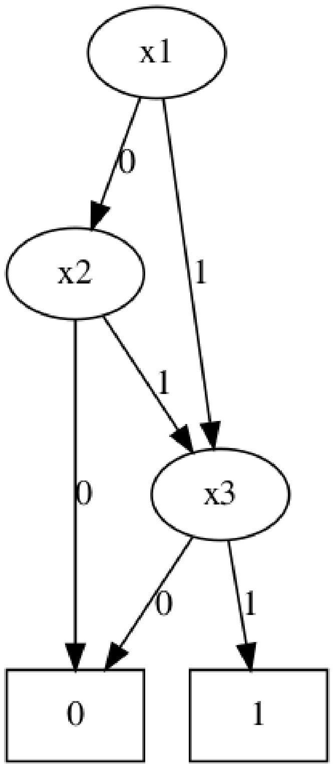
4 Example 4 (Unreduced Tree)

```
if __name__ == '__main__':
    zero = Node(-1,None,None,'0')
    one = Node(-1,None,None,'1')

    x3 = Node.createVariable(3,zero,one,'X')
    x33 = Node.createVariable(3,zero,one,'X')
    x2 = Node.createVariable(2,zero,x33,'X')
    x22 = Node.createVariable(2,x33,x3,'X')
    x1 = Node.createVariable(1,x2,x22,'X')
    x1.plot()
    g.view()
```



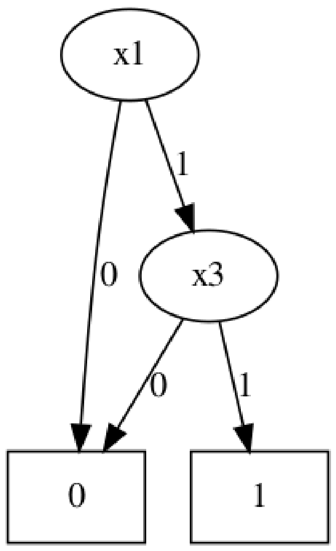
5 Example 5 (Reduced Tree)

```
if __name__ == '__main__':
    zero = Node(-1,None,None,'0')
    one = Node(-1,None,None,'1')

    x3 = Node.createVariable(3,zero,one,'X')
    x33 = Node.createVariable(3,zero,one,'X')
    x2 = Node.createVariable(2,zero,x33,'X')
    x22 = Node.createVariable(2,x33,x3,'X')
    x1 = Node.createVariable(1,x2,x22,'X')
    x1 = x1.reduce()
    x1.plot()
    g.view()
```



6 Example 6 (Apply [and])

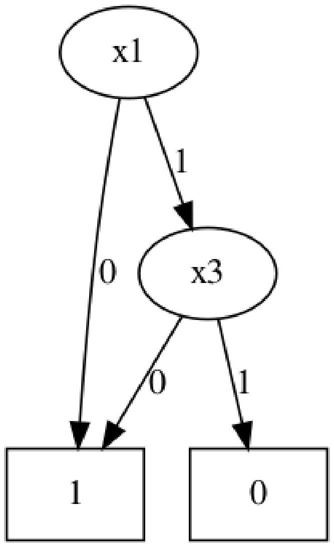
```
if __name__ == '__main__':
    zero = Node(-1,None,None,'0')
    one = Node(-1,None,None,'1')

    x1 = Node.createVariable(1,zero,one,'X')
    x3 = Node.createVariable(3,zero,one,'X')

    x1x3 = Node.apply(x1,x3,Node.__and__)

    x1x3 = x1x3.reduce()
    x1x3.plot()

    g.view()
```



7 Example 7 (Apply [and][Inverse])

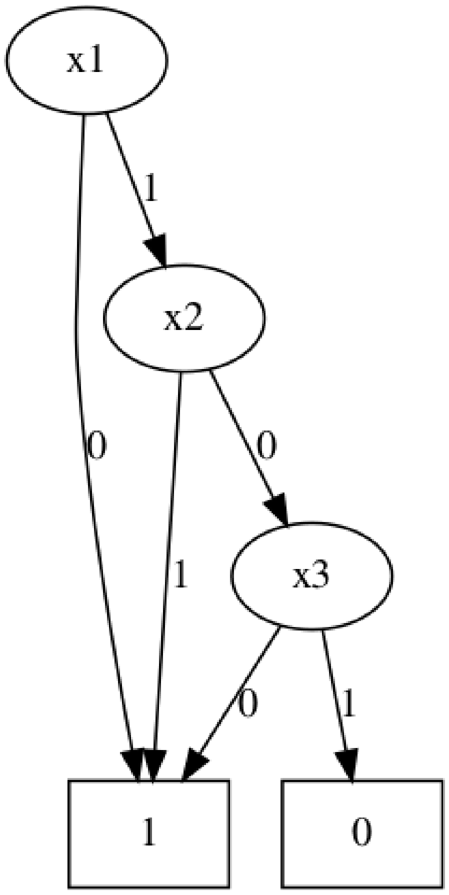
- Please always do Traverse on same variable after inverse.
- Since, invert operation changes mark value of the nodes hence to neglect above changes.

```
if __name__ == '__main__':
    zero = Node(-1,None,None,'0')
    one = Node(-1,None,None,'1')

    x1 = Node.createVariable(1,zero,one,'X')
    x3 = Node.createVariable(3,zero,one,'X')

    x1x3 = Node.apply(x1,x3,Node.__and__)

    x1x3_inv = Node.__invert__(x1x3,zero,one)
    x1x3_inv.traverse()

    x1x3_inv = x1x3_inv.reduce()
    x1x3_inv.plot()

    g.view()
```



8 Example 8 (Apply [and] [Inverse] [or])

```
if __name__ == '__main__':
    zero = Node(-1,None,None,'0')
    one = Node(-1,None,None,'1')

    x1 = Node.createVariable(1,zero,one,'X')
    x3 = Node.createVariable(3,zero,one,'X')

    x1x3 = Node.apply(x1,x3,Node.__and__)

    x1x3_inv = Node.__invert__(x1x3,zero,one)
    x1x3_inv.traverse()
    x1x3_inv = x1x3_inv.reduce()

    b = Node.apply(Node.createVariable(2,zero,one,'X'),\\
    Node.createVariable(3,zero,one,'X'),Node.__and__)

    c = Node.apply(x1x3_inv, b, Node.__or__)
    c.plot()

    g.view()
```



- Above output matches exactly with example given in Bryant paper[1].

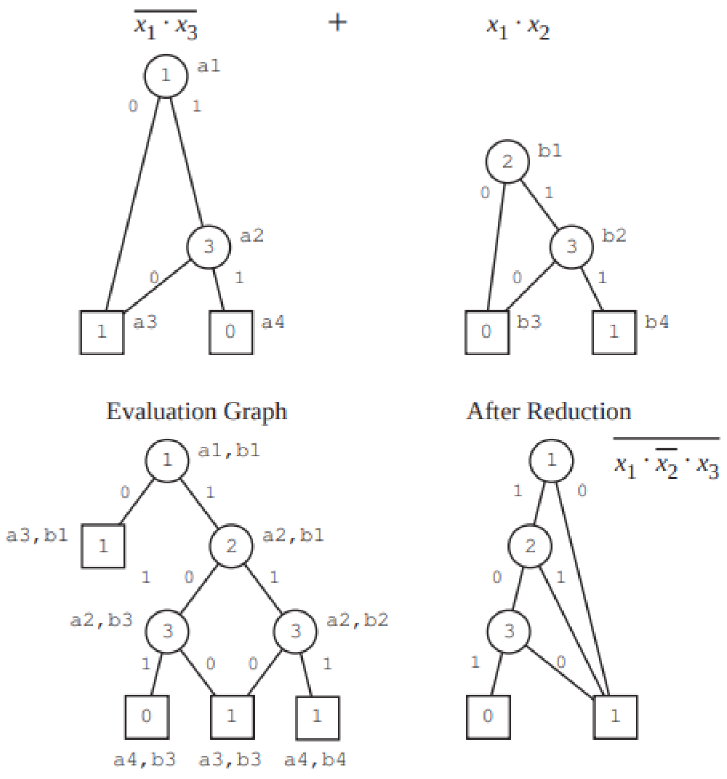
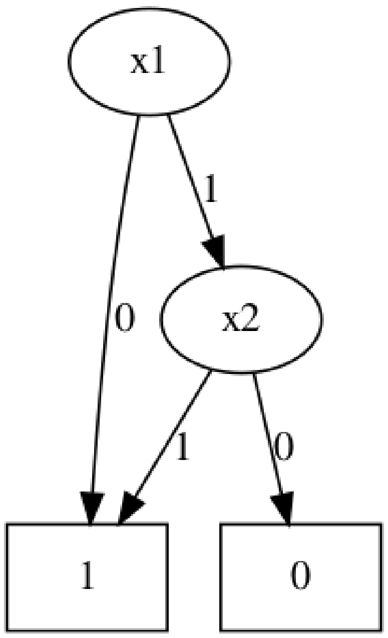


Figure 7.Example of *Apply*

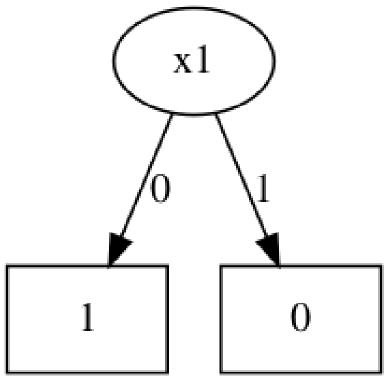
9 Example 9 (Restrict)

```
if __name__ == '__main__':
    zero = Node(-1,None,None,'0')
    one = Node(-1,None,None,'1')

    x1 = Node.createVariable(1,zero,one,'X')
    x3 = Node.createVariable(3,zero,one,'X')

    x1x3 = Node.apply(x1,x3,Node.__and__)

    x1x3_inv = Node.__invert__(x1x3,zero,one)
    x1x3_inv.traverse()
    x1x3_inv = x1x3_inv.reduce()

    b = Node.apply(Node.createVariable(2,zero,one,'X'),\\
    Node.createVariable(3,zero,one,'X'),Node.__and__)
    c = Node.apply(x1x3_inv, b, Node.__or__)
    d = Node.restrict(c,3,str(1))
    d.plot()

    g.view()
```



10 Example 10 (Restrict)

```
if __name__ == '__main__':
    zero = Node(-1,None,None,'0')
    one = Node(-1,None,None,'1')

    x1 = Node.createVariable(1,zero,one,'X')
    x3 = Node.createVariable(3,zero,one,'X')

    x1x3 = Node.apply(x1,x3,Node.__and__)

    x1x3_inv = Node.__invert__(x1x3,zero,one)
    x1x3_inv.traverse()
    x1x3_inv = x1x3_inv.reduce()

    b = Node.apply(Node.createVariable(2,zero,one,'X'),\\
    Node.createVariable(3,zero,one,'X'),Node.__and__)

    c = Node.apply(x1x3_inv, b, Node.__or__)
    d = Node.restrict(c,3,str(1))
    e = Node.restrict(d,2,str(0))
    e.plot()

    g.view()
```

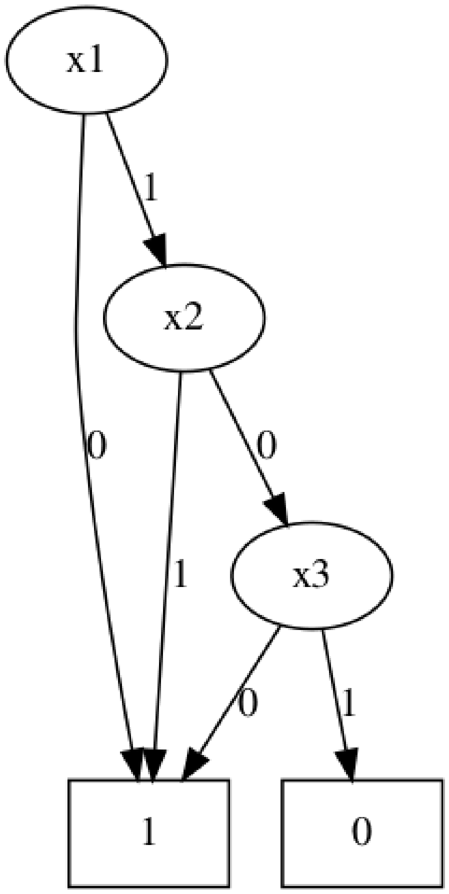


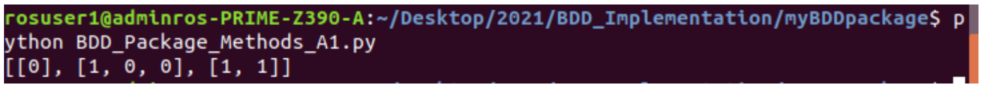
11 Example 10 (satisfyAll)

- Returns all the strings which satisfy the given binary decision diagram.
- We need to pass empty result and x_array list in the satisfyAll method.
- Syntax- Node.satisfyAll(1,c,x_array,result)
- Node — Class Name.
- 1 — index of the variable
- x_array and result. Empty list as input. x_array used as stack in the method and result stores all possible solutions.

```
if __name__ == '__main__':
    zero = Node(-1,None,None,'0')
    one = Node(-1,None,None,'1')
```

```
x1 = Node.createVariable(1,zero,one,'X')
x3 = Node.createVariable(3,zero,one,'X')

x1x3 = Node.apply(x1,x3,Node.__and__)

x1x3_inv = Node.__invert__(x1x3,zero,one)
x1x3_inv.traverse()
x1x3_inv = x1x3_inv.reduce()

b = Node.apply(Node.createVariable(2,zero,one,'X'),\\
Node.createVariable(3,zero,one,'X'),Node.__and__)

c = Node.apply(x1x3_inv, b, Node.__or__)
c.plot()
result = []
x_array = []
Node.satisfyAll(1,c,x_array,result)  #We need empty result list to be passed
print(result)

g.view()
```





```
rosuser1@adminros-PRIME-Z390-A:~/Desktop/2021/BDD_Implementation/myBDDpackage$ p
ython BDD_Package_Methods_A1.py
[[0], [1, 0, 0], [1, 1]]
```

12 References

[1]. Bryant, Randal. (1986). Bryant, R.E.: Graph-Based Algorithms for Boolean Function Manipulation. IEEE Trans. Computers 35(8), 677-691. Computers, IEEE Transactions on. C-35. 677 - 691. 10.1109/TC.1986.1676819.

[2]. Bryant, Randal. (2003). Symbolic Boolean Manipulation with Ordered Binary Decision Diagrams. ACM Computing Surveys. 24. 10.1145/136035.136043.