Program Analysis

Topic: Binary Decision Diagrams (BDD)

Nikhil Telkunte (CS16M028)
Dhiraj Kanake (CS16M030)

A **binary decision diagram** (**BDD**) is a data structure that is used to represent a Boolean function. On a more abstract level, BDDs can be considered as a compressed representation of sets or relations. Unlike other compressed representations, operations are performed directly on the compressed representation, i.e. without decompression.
BDDs can represent large sets of redundant data efficiently.

A Boolean function can be represented as a rooted, directed, acyclic graph, which consists of several decision nodes and terminal nodes. There are two types of terminal nodes called **0-terminal and 1-terminal.** Each decision node N is labelled by Boolean variable V_N and has two child nodes called low child and high child. The edge from node V_N to a low (or high) child represents an assignment of V_N to 0 (resp. 1).

There are two types of BDD
1. Ordered BDD
2. Reduced ordered BDD

Why BDD?

Using truth table representation for Boolean function we need.

Space: For *n* variables, needs to store 2^n. (n+1) bits
n+1 bits for each row of truth table including n variable and one function bit.
2^n rows for n variables.
Operations: Visit each entry of truth table.

If Boolean functions are represented using truth tables, a function with 100 variables needs more than 2^{100} bits.

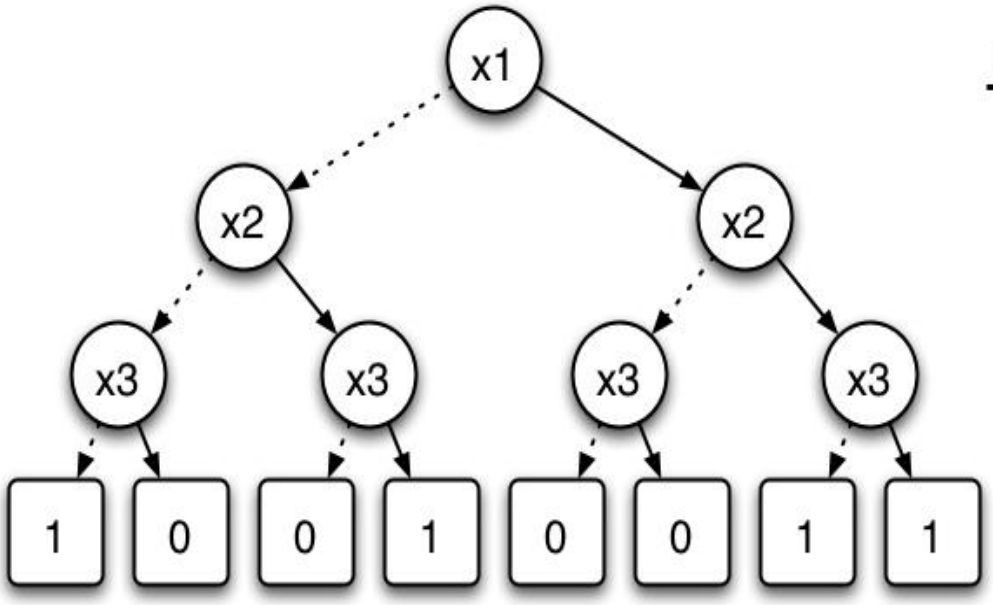
To represent a Boolean function using BDD we first need to represent it into binary decision tree.

The figure below shows a binary decision tree (the reduction rules are not applied), and a truth table, each representing the function f (x1, x2, x3). In the tree on the left, the value of the function can be determined for a given variable assignment by following a path down the graph to a terminal.

In the figures below, dotted lines represent edges to a low child, while solid lines represent edges to a high child. Therefore, to find (x1=0, x2=1, x3=1), begin at x1, traverse down the dotted line to x2 (since x1 has an assignment to 0), then down two solid lines (since x2 and x3 each have an assignment to one). This leads to the terminal 1, which is the value of f (x1=0, x2=1, x3=1).

The function is

$$f(x1, x2, x3) = x'1\ x'2\ x'3 + x1x2 + x2x3$$



x1	x2	x3	*f*
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

fig.1

Some Definitions:

Variable Ordering:
The size of the BDD is determined both by the function being represented and the chosen ordering of the variables. There exist Boolean functions $f(x_1,x_2,....x_n)$ for which depending upon the ordering of the variables we would end up getting a graph whose number of nodes would be linear (in *n*) at the best and exponential at the worst case.

Ordered Binary Decision Diagrams (OBDD) :

Ordered binary decision diagrams (OBDDs) are just like BDDs but with a defined variable ordering. That is, a tuple $p= (z_1, . . ., z_m)$. Let a node u labelled 'z_i' have two child nodes v and w. The variable v must be labelled by a variable z_j such that $z_i <_p z_j$, that is, z_j must come later in the ordered list of variables in p. The same thing must be true for the other child node (w). Size of OBDD depends on the chosen ordering. The variable ordering makes it possible to simplify (reduce) OBDDs.

Reduced Ordered Binary Decision Diagrams (ROBDD):

The advantage of an ROBDD is that it is canonical (unique) for a particular function and variable order. The binary decision tree can be transformed into a binary decision diagram by maximally reducing it according to the two reduction rules.

Elimination rule: If v is an inner node and both of its children points to the same node w, then remove v and redirect all incoming edges to w.

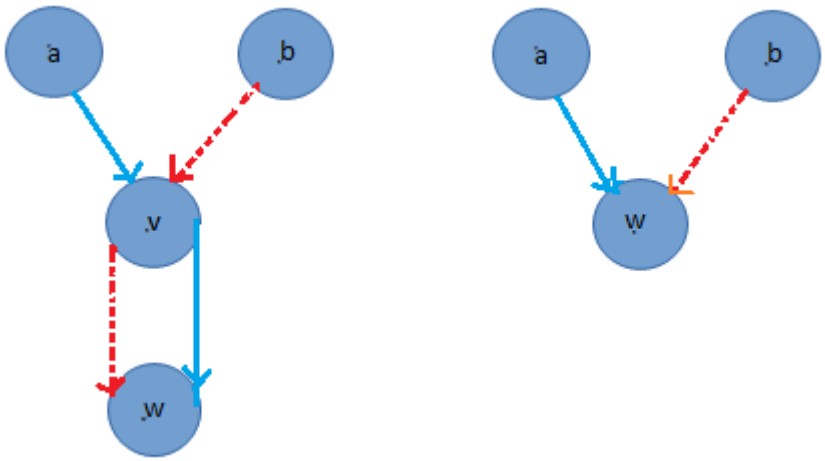


Fig.2

Isomorphism rule: If the nodes v and w are terminals and if they have the same value, then remove node v and redirect all incoming edges to node w. If v and w are inner nodes, they are labelled by the same Boolean variable and their children are the same, then remove node v and redirect all incoming edges to w.

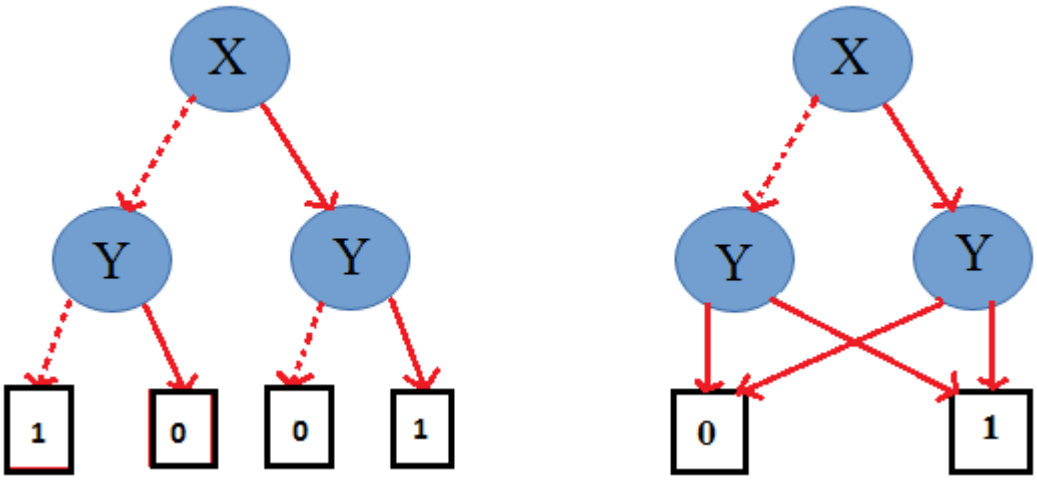


Fig.4

In general, the size of ROBDD depends of the variable ordering. The chosen variable ordering makes a significant difference to the size of the ROBDD representing a given function. To show the effect of the ordering of variable, we take f(x1, x2, x3, x4, x5, x6) = (x1+x2) .(x3+x4).(x5+x6). The ROBDD of function *f* with variable ordering [x1, x2, x3, x4, x5, x6] and with variable ordering [x1, x3, x5, x2, x4, x6] is shown in Figure below. So it may be noted that the size of ROBDD is sensible to the variable ordering. If the chosen variable ordering is a good one, then we get a very compact ROBDD representation for the given function, otherwise, the ROBDD representation is an expensive one.

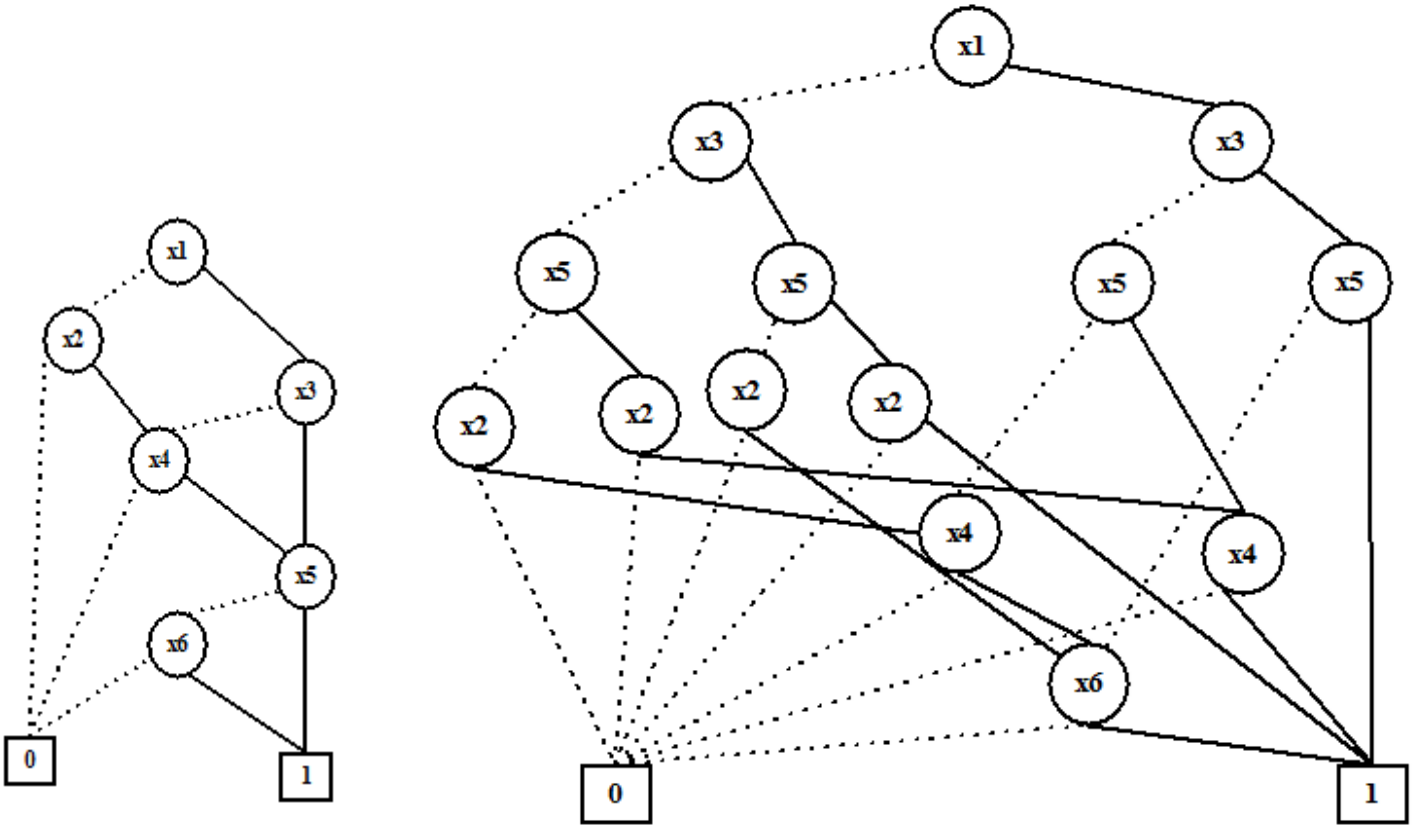


Fig.5: ROBDD with variable ordering [x1, x2, x3, x4, x5, x6] and [x1, x3, x5, x2, x4, x6]

OBDD size is strongly influenced by the variable ordering and a good variable ordering always produces a compact OBDD. Optimal variable ordering is NP-Complete problem. Many heuristics have been proposed to get a good variable ordering which leads to a compact representation of OBDD of a given function. There are two classes of heuristics – static variable ordering and dynamic variable ordering. In case of static variable ordering, order of the variable is identified before the construction of the variables and that ordering is maintained throughout the process. In case of dynamic variable ordering, we use the ordering heuristics during the construction or operation on OBDDs.

Algorithm to reduce OBDD:

Leaves: Label all 0 terminals with #0 and all 1 terminals with #1.

Intermediate node n:

- If 0-child and 1-child of n has same label, set label of that n to be that label (i.e. label of its child).
- If there is another node m such that m has the same variable x_i and the children of n and m have same label, then set label of n to be label of m.
- Otherwise set label of n to be next unused integer.

Example: 1

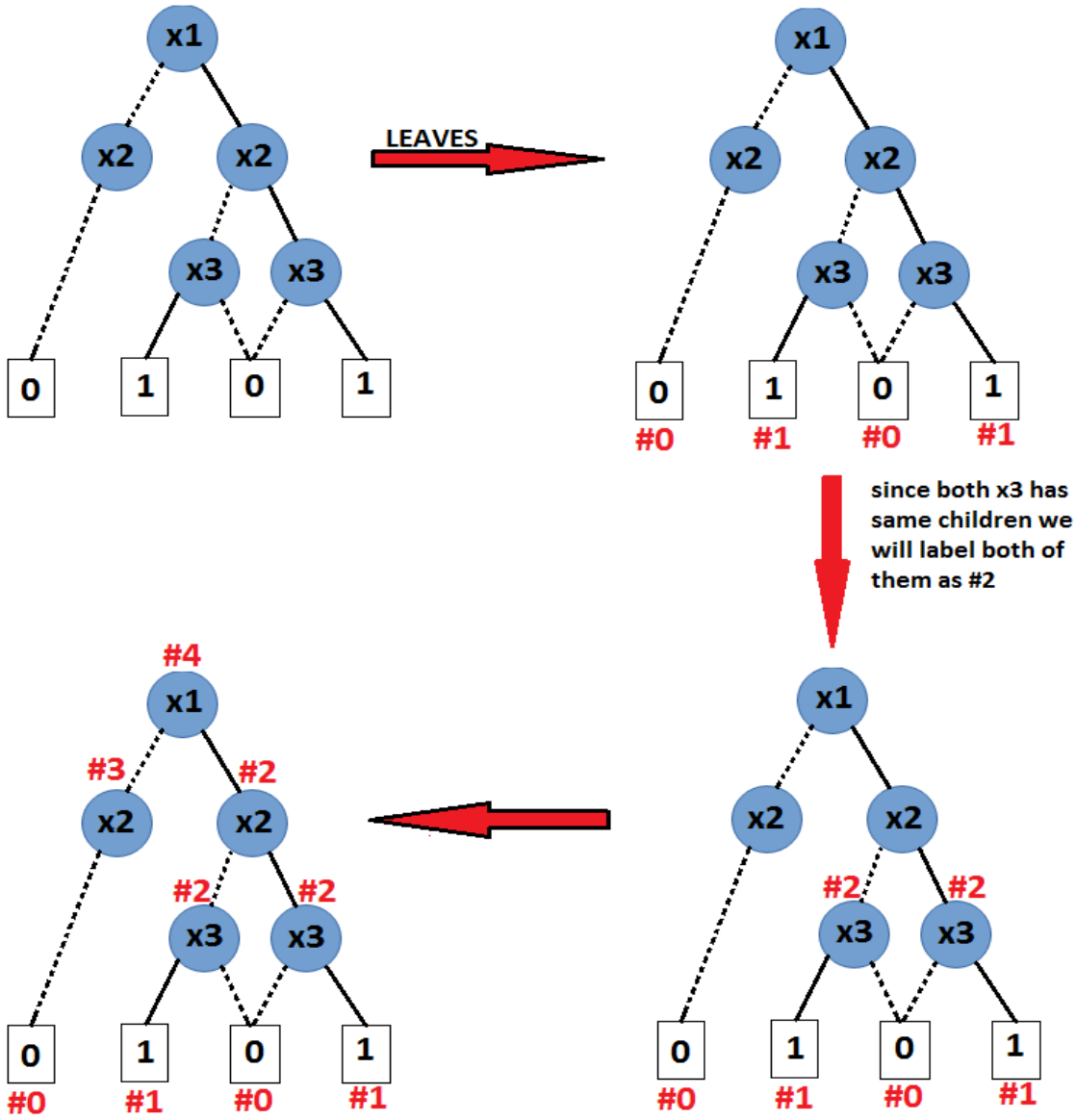


Fig.6

If the next higher node has same id for both 0 child and 1 child, then use the same id for that node. Else we have to give a new id to the node. After following the steps, we get a final tree with all labelled node. draw edges from one id to another unique id.
As we can see id #4 has a 1 edge to id #2, so draw one edge form #4 to #2.
There is 1 edge from #3 to #2, so draw that edge. After repeating the same we will get the reduced BDD as shown in Fig.7.

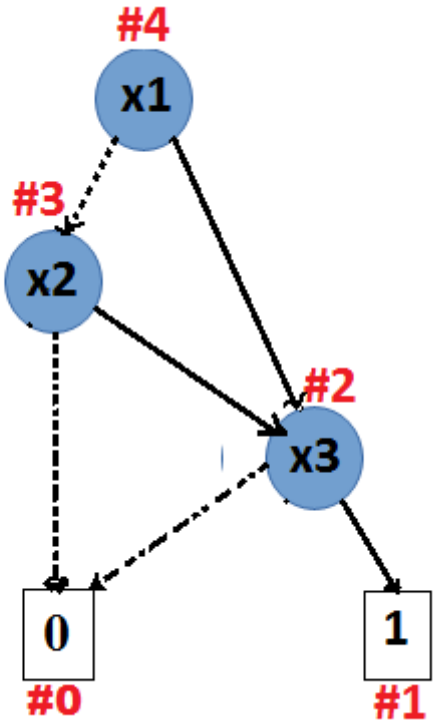


Fig.7: Reduced BDD.

Example: 2

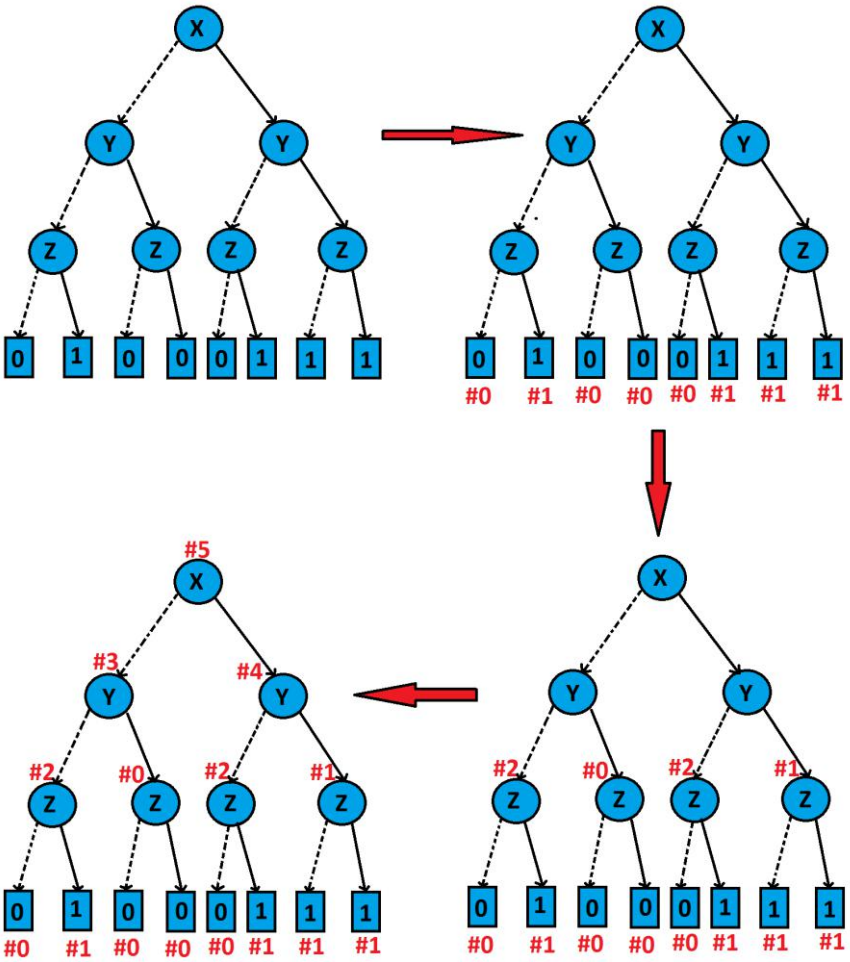


Fig.8

Id #5 edge 1 is going to id #4 and edge 0 is going to #3, so draw that edges.
Id #4 edge 1 is going to #1 and edge 0 is going to #2.
Id #3 edge 1 is going to #0 and edge 0 is going to #2.
Id #2 edge 1 is going to #1 and edge 0 is going to #0.
After drawing this edges we will get a reduced BDD as shown in Fig.9.

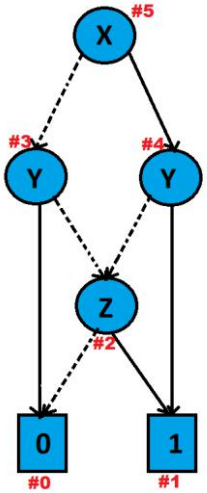


Fig.9: Reduced BDD.

Program Analysis Using Binary Decision Diagrams

Analysis of programs with pointer to memory must estimates the effects of operations performed through pointers. A points-to analysis approximates, for each pointer in the program, the set of objects to which the pointer may point. In our example points-to analysis, we represent each object by the allocation site at which it is allocated. Suppose that p and q are pointers and the assignment p = q appears in the program. since q is assigned to p, after the assignment, p may point to any object to which q was pointing. This is modelled in the analysis with the subset constraint points-to(q) ⊆points-to(p).

To use a concrete example, consider a program statement shown below.

```
a=&X;
b=&Y;
c=&z;
a=b;
b=a;
c=b;
```

the first three statement are allocation statement, which would cause the analysis to initialize the points-to sets of a, b and c to {X}, {Y} and {Z}, respectively.
After last three instructions the final points-to sets for the example would be

Points-to(a) = {X, Y}
Points-to(b) = {X, Y}
Points-to(c) = {X, Y, Z}

When analysing large programs, a key problem is that the number of points-to sets and the size of each set may become very large. Various techniques have been studied for compactly representing the points-to sets and efficiently solving the subset constraints. We review one such technique, which is to use BDDs to compactly represent the points-to sets and BDD operations to efficiently propagate them along subset constraints. To use a concrete example, we will now show how the points-to sets computed for the above statements can be encoded in a BDD. We could write the points-to sets as a set of points-to pairs, with each pair indicating that a given pointer may point to a given object, as follows:

{(a, X), (a, Y), (b, X), (b, Y), (c, X), (c, Y), (c, Z)}

Using 00 to represent a and X, 01 to represent b and Y, and 10 to represent c and Z, we can encode these points-to pairs as the set of binary vectors.

{0000, 0001, 0100, 0101, 1000, 1001, 1010}

A BDD representing this set of binary vectors is shown below. The pointers a, b, and c are encoded in first two bit positions of the BDD, and the objects X, Y, and Z are encoded third and fourth bit positions. We follow the common convention of drawing the 0- successor of each node as a dashed arrow, and the 1-successor as a solid arrow.

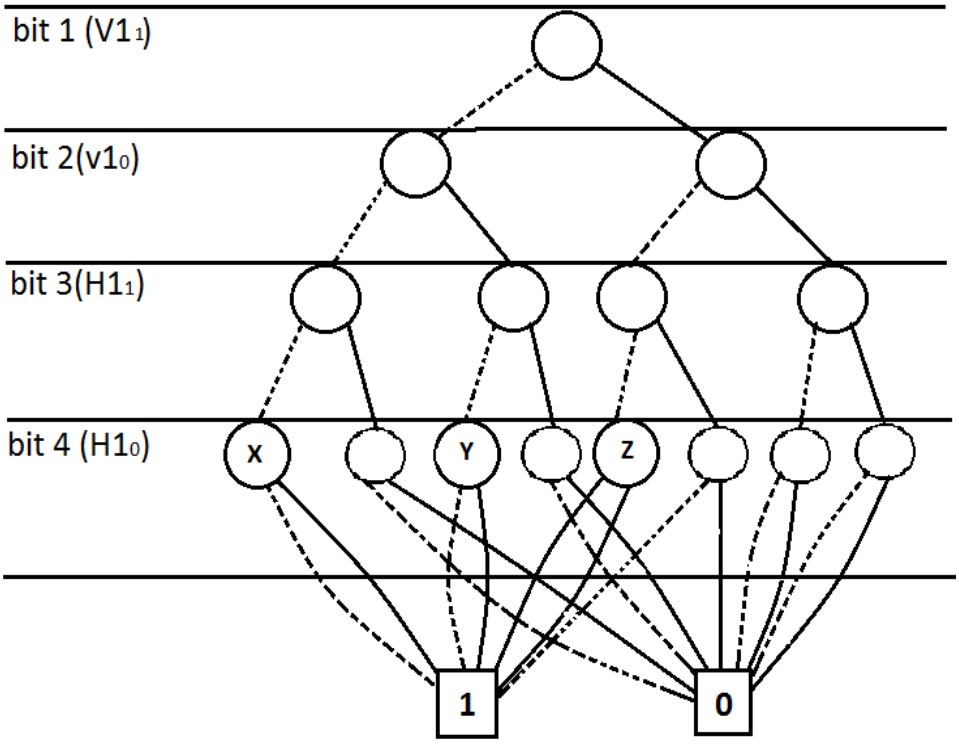


Fig.10: Unreduced BDD for points-to example.

The nodes marked x, y, and z in above figure are at the same bit position and have the same successors, because they all represent the same subset of objects {X, Y}. Since these nodes are the same, they could be merged into a single node, making the BDD smaller without changing the set that it represents.

Furthermore, since their 0- and 1-successor are the same (the 1 node), the value of the bit that they test does not affect the successor, so the bit does not need to be tested and the nodes could be removed entirely. If we repeatedly reduce the BDD in this way by finding mergeable and unnecessary nodes, we obtain the reduced BDD shown in Figure below. The BDD represents the same set as the original unreduced BDD, but it is smaller.

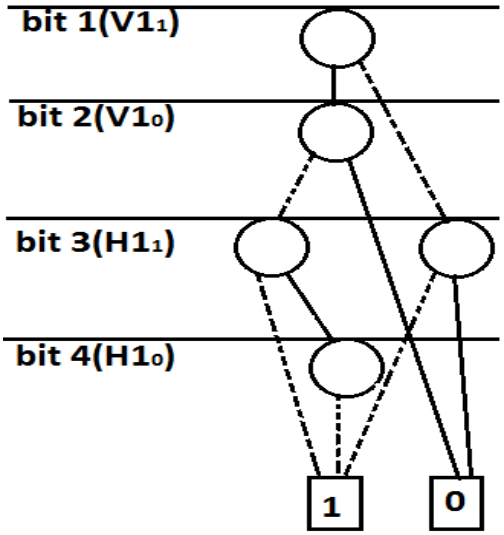


Fig.11: Reduced BDD for points-to example.

Context-sensitive call graphs in BDD

Call graphs relations

Call graphs expressed as a relation.
-five edges:

- Calls (A, B)
- Calls (A, C)
- Calls (A, D)
- Calls (B, D)
- Calls (C, B)

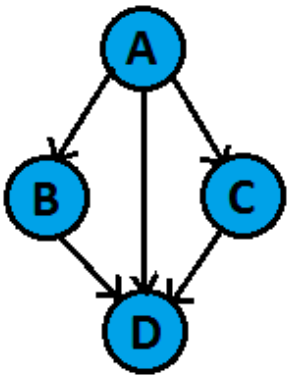


Fig.12

We can represent a call graph as binary function, for that label each node with some binary number and then translate this relation form into binary string

Relation expressed as binary function
A=00, B=01, C=10, D=11

Calls (A, B) – 00 01
Calls (A, B) – 00 10
Calls (A, B) – 00 11
Calls (A, B) – 01 11
Calls (A, B) – 10 11

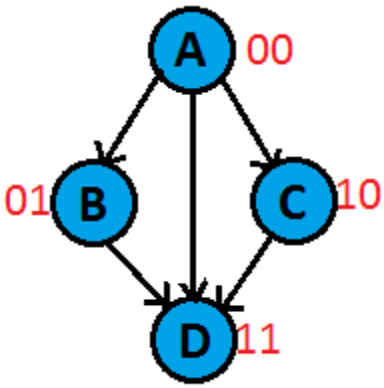


Fig.13

TRUTH TABLE

X1	X2	X3	X4	f
0	0	0	0	0
0	0	0	1	1
0	0	1	0	1
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	1
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	1
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	0

function is going to be 0
if edge is not in the graph
else 1

- **Graphical encoding of the truth table**

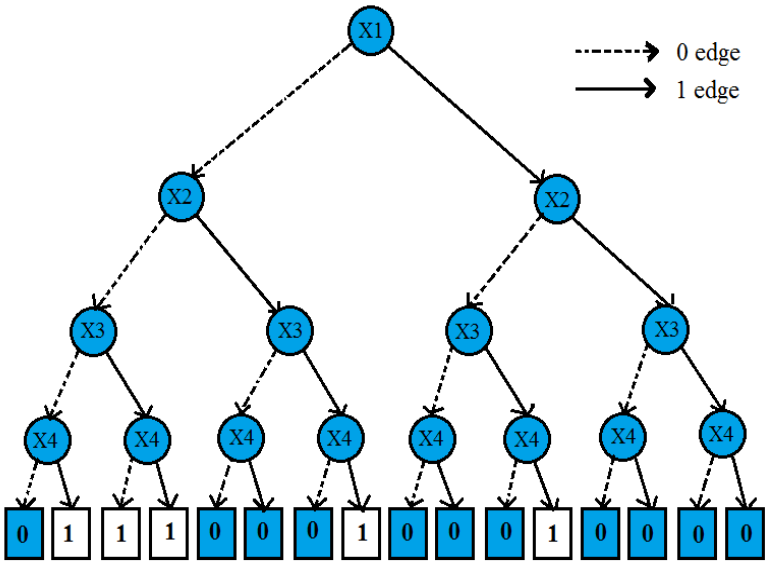


Fig.14

Label all terminal using algorithm.

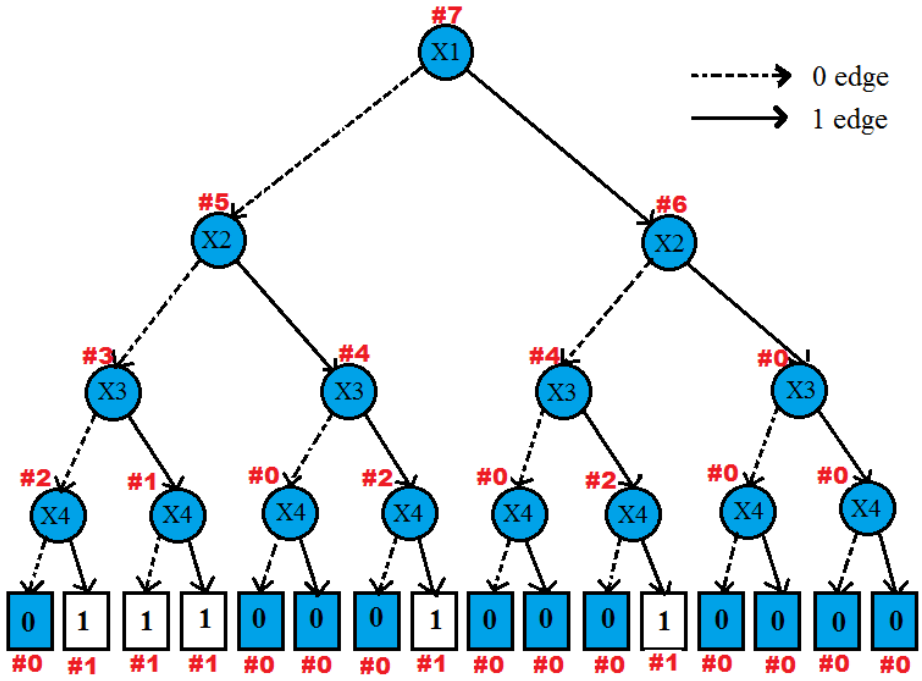


Fig.15

Id #7 edge 1 is going to id #6 and edge 0 is going to #5. Id #5 edge 1 is going to id #4 and edge 0 is going to #3, and Id #6 edge 1 is going to id #0 and edge 0 is going to #4, so draw that edges We will get tree like Fig.16

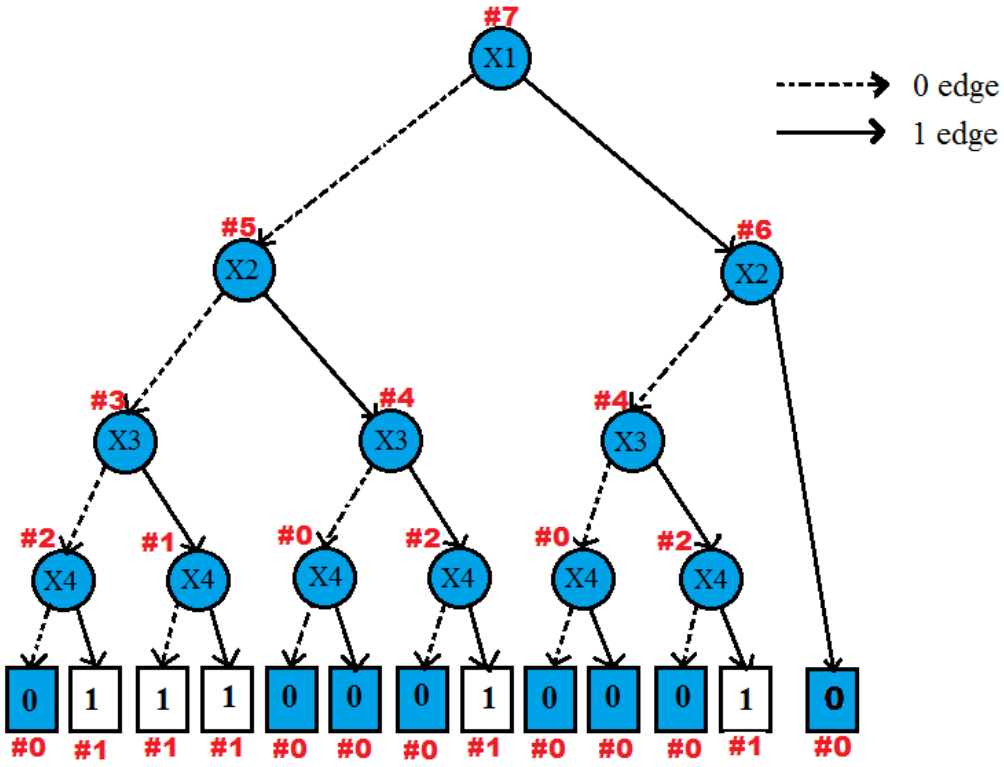


Fig.16

Both #4 id nodes have same terminals, so we can merge them together

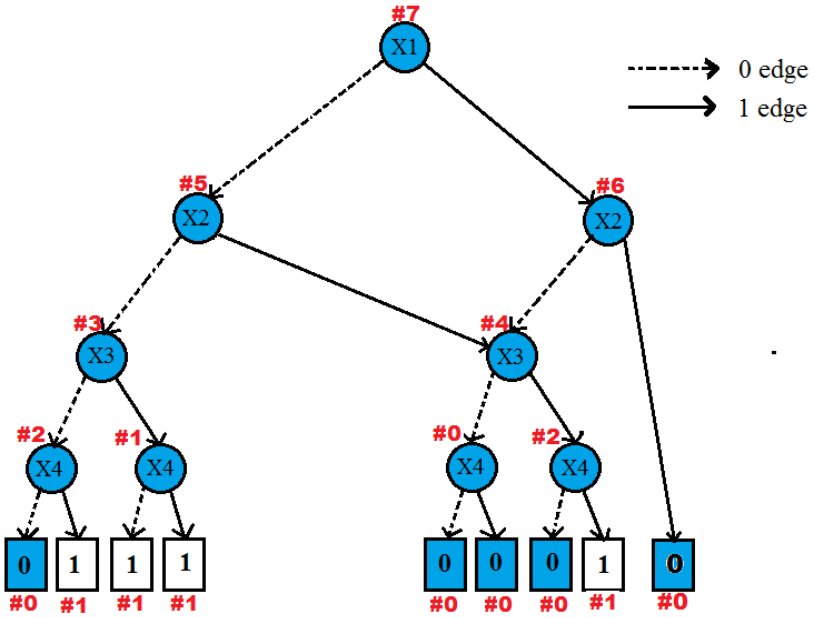


Fig.17

#3 and #4 going to same node #2 so merge them.

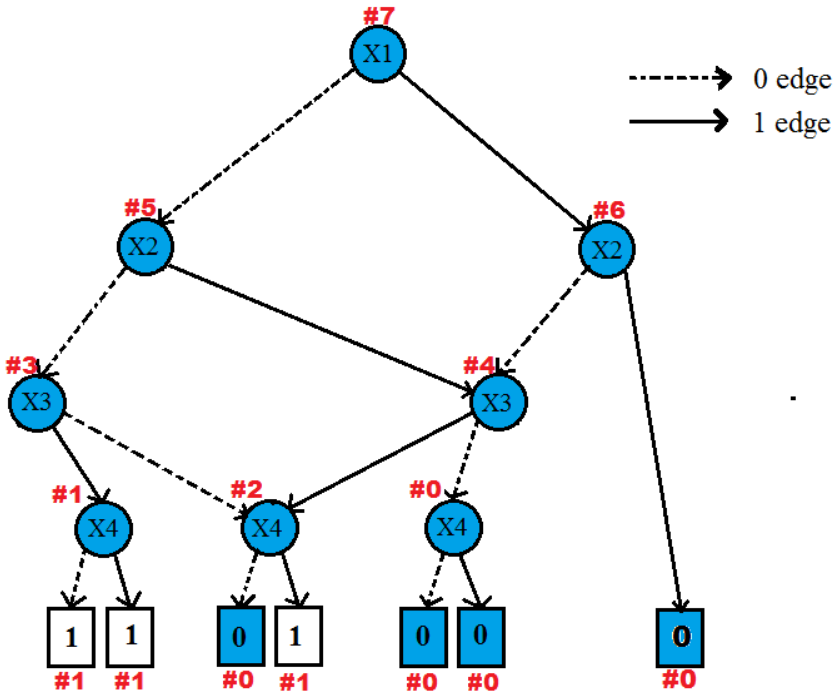


Fig.18

Id #3 edge 1 is going to id #1 and id #4 is going to #0. So we can remove both X4 nodes. after that we will get tree like Fig.19

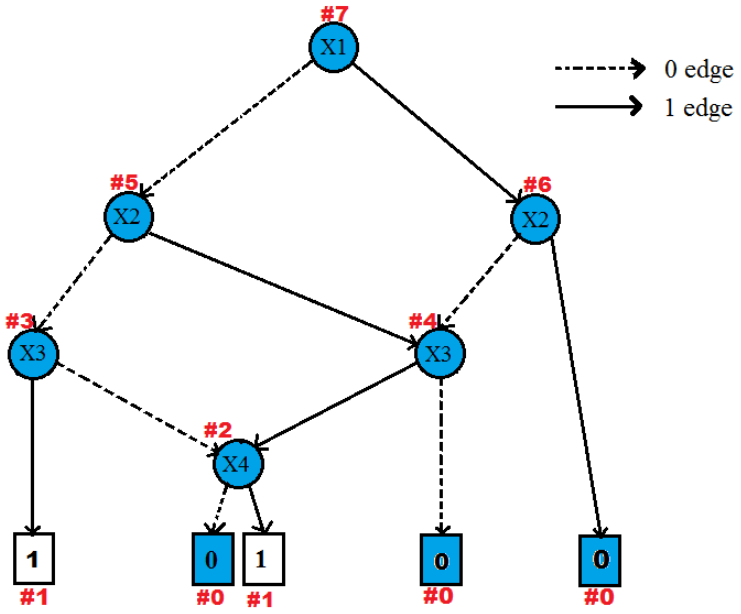


Fig.19

Here we have many nodes with same label and same successor which can be collapse. So, all the 1 node can be collapse together and all the 0 node can be collapse together. After that we will get reduced BDD.

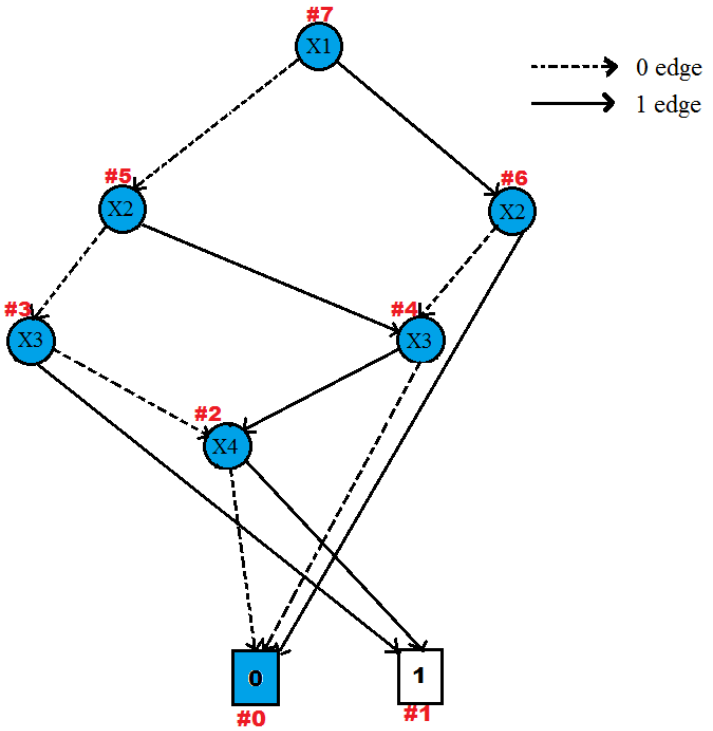


Fig.20: reduced BDD

Fig.15

Advantages of reduced order binary decision diagram (ROBDD)

- A ROBDD can represent an exponential number of paths with linear number of nodes.
- It is canonical (unique) for a particular function and variable order.

Problems with ROBDDs

- some functions have exponential-size ROBDDs
  multipliers, HWB
- size of ROBDD depends critically on variable ordering
  --adders – exponential for bad ordering, linear for good orderings.
- Finding a best variable ordering requires exponential time.

o Multipliers and hidden weighted bit(HWB) function

1) Multipliers:
   Consider multiplication of two n bit integers yielding a 2n bit result:

   $$(p_{2n-1}, \ldots, p_0) = (a_{n-1}, \ldots, a_0) \times (b_{n-1}, \ldots, b_0)$$

   The BDD representing p_n and p_{n-1} exhibit exponential complexity as n increases regardless of the variable order.

2) Hidden weighted bit(HWB):
   The function has n inputs $X= \{x_1, x_2, \ldots, x_n\}$. define weight, denoted wt(a), to be the number of 1 bits in an assignment $\{a_1, a_2, \ldots, a_n\}$ to X. the hidden weighted bit function selects the i^{th} input, i = wt(a).

   $$HWB(a)=0,\ wt(a)=0$$

   BDD representation of HWB requires $\Omega(1.14^n)$ nodes. where Ω is asymptotic lower bound.