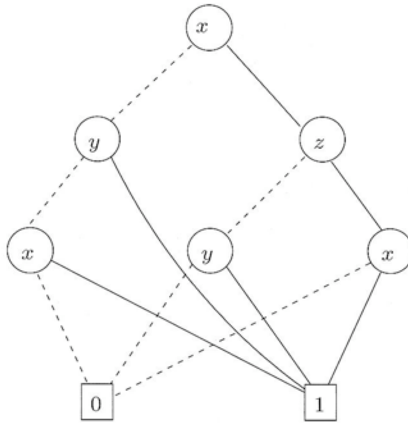

Figure 6.7. A BDD where some boolean variables occur more than once on an evaluation path.

a variable the values 0 and 1 simultaneously.) Checking validity is similar, but we check that no 0-terminal is reachable by a consistent path.

The operations $\cdot$ and $+$ can be performed by 'surgery' on the component BDDs. Given BDDs B_f and B_g representing boolean functions f and g, a BDD representing $f \cdot g$ can be obtained by taking the BDD f and replacing all its 1-terminals by B_g. To see why this is so, consider how to get to a 1-terminal in the resulting BDD. You have to satisfy the requirements for getting to a 1 imposed by both of the BDDs. Similarly, a BDD for $f + g$ can be obtained by replacing all 0 terminals of B_f by B_g. Note that these operations are likely to generate BDDs with multiple occurrences of variables along a path. Later, in Section 6.2, we will see definitions of $+$ and $\cdot$ on BDDs that don't have this undesirable effect.

The complementation operation $\bar{\ }$ is also possible: a BDD representing $\overline{f}$ can be obtained by replacing all 0-terminals in B_f by 1-terminals and vice versa. Figure 6.8 shows the complement of the BDD in Figure 6.2.

6.1.3 Ordered BDDs

We have seen that the representation of boolean functions by BDDs is often compact, thanks to the sharing of information afforded by the reductions C1–C3. However, BDDs with multiple occurrences of a boolean variable along a path seem rather inefficient. Moreover, there seems no easy way to test for equivalence of BDDs. For example, the BDDs of Figures 6.7 and 6.9 represent the same boolean function (the reader should check this). Neither of them can be optimised further by applying the rules C1–C3. However,

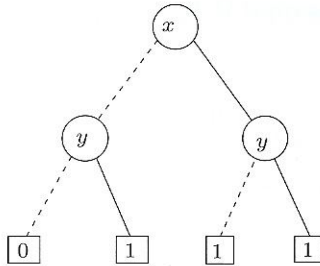

Figure 6.8. The complement of the BDD in Figure 6.2.

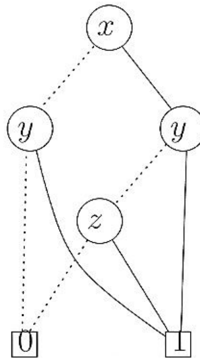

Figure 6.9. A BDD representing the same function as the BDD of Figure 6.7, but having the variable ordering $[x, y, z]$.

testing whether they denote the same boolean function seems to involve as much computational effort as computing the entire truth table for $f(x, y, z)$.

We can improve matters by imposing an ordering on the variables occurring along any path. We then adhere to that same ordering for all the BDDs we manipulate.

Definition 6.6 Let $[x_1, \ldots, x_n]$ be an ordered list of variables without duplications and let B be a BDD all of whose variables occur somewhere in the list. We say that B has the ordering $[x_1, \ldots, x_n]$ if all variable labels of B occur in that list and, for every occurrence of x_i followed by x_j along any path in B, we have $i < j$.

An ordered BDD (OBDD) is a BDD which has an ordering for some list of variables.

Note that the BDDs of Figures 6.3(a,b) and 6.9 are ordered (with ordering $[x, y]$). We don't insist that every variable in the list is used in the paths. Thus, the OBDDs of Figures 6.3 and 6.9 have the ordering $[x, y, z]$ and so

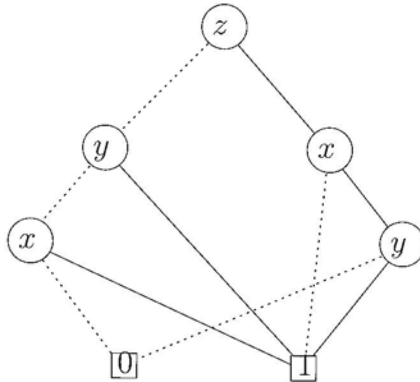

Figure 6.10. A BDD which does not have an ordering of variables.

does any list having x, y and z in it in that order, such as $[u, x, y, v, z, w]$ and $[x, u, y, z]$. Even the BDDs B_0 and B_1 in Figure 6.6 are OBDDs, a suitable ordering list being the empty list (there are no variables), or indeed *any* list. The BDD B_x of Figure 6.6(b) is also an OBDD, with any list containing x as its ordering.

The BDD of Figure 6.7 is not ordered. To see why this is so, consider the path taken if the values of x and y are 0. We begin with the root, an x-node, and reach a y-node and then an x-node again. Thus, no matter what list arrangement we choose (remembering that no double occurrences are allowed), this path violates the ordering condition. Another example of a BDD that is not ordered can be seen in Figure 6.10. In that case, we cannot find an order since the path for $(x, y, z) \Rightarrow (0, 0, 0)$ – meaning that x, y and z are assigned 0 – shows that y needs to occur before x in such a list, whereas the path for $(x, y, z) \Rightarrow (1, 1, 1)$ demands that x be before y.

It follows from the definition of OBDDs that one cannot have multiple occurrences of any variable along a path.

When operations are performed on two OBDDs, we usually require that they have *compatible variable orderings*. The orderings of B_1 and B_2 are said to be compatible if there are no variables x and y such that x comes before y in the ordering of B_1 and y comes before x in the ordering of B_2. This commitment to an ordering gives us a unique representation of boolean functions as OBDDs. For example, the BDDs in Figures 6.8 and 6.9 have compatible variable orderings.

Theorem 6.7 The reduced OBDD representing a given function f is unique. That is to say, let B and B' be two reduced OBDDs with

compatible variable orderings. If B and B represent the same boolean function, then they have identical structure.

In other words, with OBDDs we cannot get a situation like the one encountered earlier, in which we have two distinct reduced BDDs which represent the same function, provided that the orderings are compatible. It follows that checking equivalence of OBDDs is immediate. Checking whether two OBDDs (having compatible orderings) represent the same function is simply a matter of checking whether they have the same structure[1].

A useful consequence of the theorem above is that, if we apply the reductions C1–C3 to an OBDD until no further reductions are possible, then we are guaranteed that the result is always the same reduced OBDD. The order in which we applied the reductions does not matter. We therefore say that OBDDs have a *canonical form*, namely their unique reduced OBDD. Most other representations (conjunctive normal forms, etc.) do not have canonical forms.

The algorithms for $\cdot$ and $+$ for BDDs, presented in Section 6.1.2, won't work for OBDDs as they may introduce multiple occurrences of the same variable on a path. We will soon develop more sophisticated algorithms for these operations on OBDDs, which exploit the compatible ordering of variables in paths.

OBDDs allow compact representations of certain classes of boolean functions which only have exponential representations in other systems, such as truth tables and conjunctive normal forms. As an example consider the *even parity function* $f_{\text{even}}(x_1, x_2, \dots, x_n)$ which is defined to be 1 if there is an even number of variables x_i with value 1; otherwise, it is defined to be 0. Its representation as an OBDD requires only $2n + 1$ nodes. Its OBDD for $n = 4$ and the ordering $[x_1, x_2, x_3, x_4]$ can be found in Figure 6.11.

The impact of the chosen variable ordering The size of the OBDD representing the parity functions is independent of the chosen variable ordering. This is because the parity functions are themselves independent of the order of variables: swapping the values of any two variables does not change the value of the function; such functions are called symmetric.

However, in general the chosen variable ordering makes a significant difference to the size of the OBDD representing a given function. Consider the boolean function $(x_1 + x_2) \cdot (x_3 + x_4) \cdot \dots \cdot (x_{2n-1} + x_{2n})$; it corresponds to a propositional formula in conjunctive normal form. If we choose the

[1] In an implementation this will amount to checking whether two pointers are equal.

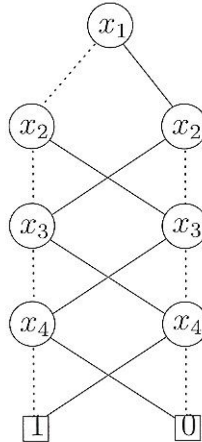

Figure 6.11. An OBDD for the even parity function for four bits.

'natural' ordering $[x_1, x_2, x_3, x_4, \ldots]$, then we can represent this function as an OBDD with $2n + 2$ nodes. Figure 6.12 shows the resulting OBDD for $n = 3$. Unfortunately, if we choose instead the ordering

$$[x_1, x_3, \ldots, x_{2n-1}, x_2, x_4, \ldots, x_{2n}]$$

the resulting OBDD requires 2^{n+1} nodes; the OBDD for $n = 3$ can be seen in Figure 6.13.

The sensitivity of the size of an OBDD to the particular variable ordering is a price we pay for all the advantages that OBDDs have over BDDs. Although finding the optimal ordering is itself a computationally expensive problem, there are good heuristics which will usually produce a fairly good ordering. Later on we return to this issue in discussions of applications.

The importance of canonical representation The importance of having a canonical form for OBDDs in conjunction with an efficient test for deciding whether two reduced OBDDs are isomorphic cannot be overestimated. It allows us to perform the following tests:

Absence of redundant variables. If the value of the boolean function $f(x_1, x_2, \ldots, x_n)$ does not depend on the value of x_i, then any reduced OBDD which represents f does not contain any x_i-node.

Test for semantic equivalence. If two functions $f(x_1, x_2, \ldots, x_n)$ and $g(x_1, x_2, \ldots, x_n)$ are represented by OBDDs B_f, respectively B_g, with a compatible ordering of variables, then we can efficiently decide whether f and g are semantically equivalent. We reduce B_f and B_g (if necessary); f

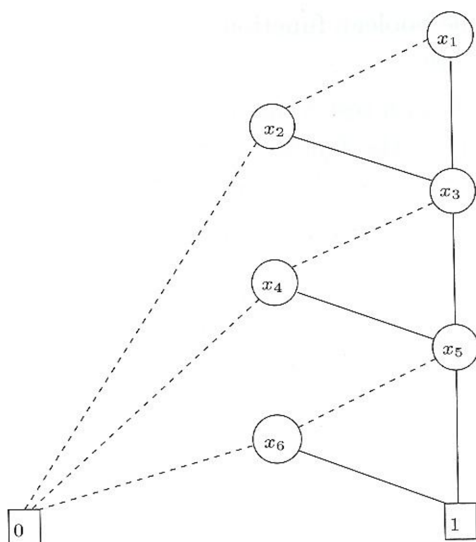

Figure 6.12. The OBDD for $(x_1 + x_2) \cdot (x_3 + x_4) \cdot (x_5 + x_6)$ with variable ordering $[x_1, x_2, x_3, x_4, x_5, x_6]$.

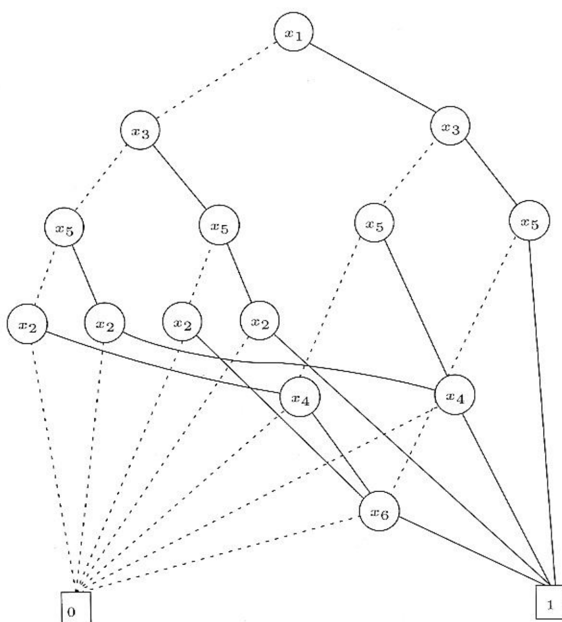

Figure 6.13. Changing the ordering may have dramatic effects on the size of an OBDD: the OBDD for $(x_1 + x_2) \cdot (x_3 + x_4) \cdot (x_5 + x_6)$ with variable ordering $[x_1, x_3, x_5, x_2, x_4, x_6]$.

and g denote the same boolean functions if, and only if, the reduced OBDDs have identical structure.

Test for validity. We can test a function $f(x_1, x_2, \ldots, x_n)$ for validity (i.e. f always computes 1) in the following way. Compute a reduced OBDD for f. Then f is valid if, and only if, its reduced OBDD is B_1.

Test for implication. We can test whether $f(x_1, x_2, \ldots, x_n)$ implies $g(x_1, x_2, \ldots, x_n)$ (i.e. whenever f computes 1, then so does g) by computing the reduced OBDD for $f \cdot \overline{g}$. This is B_0 iff the implication holds.

Test for satisfiability. We can test a function $f(x_1, x_2, \ldots, x_n)$ for satisfiability (f computes 1 for at least one assignment of 0 and 1 values to its variables). The function f is satisfiable iff its reduced OBDD is not B_0.