at computing an *optimal* order, but one is satisfied with sufficiently large improvements.

The use of a reordering algorithm can now be controlled as follows:

Explicit calls: The user controls the points in time at which a reordering step is to be performed. For example, this may be desirable before beginning to perform a complex operation.

Automatic calls: The reordering algorithm is called automatically whenever certain situations arise. Typically, a reordering step is called whenever the size of the shared OBDD representation has been doubled since the last reordering call.

The reordering aspect is particularly interesting, as it can run in the background without direct interaction to the application program. Typically, for the application program only the references to the represented functions are of interest, and not their internal representation. The dynamic adjustment of the variable order allows the internally performed OBDD representation to be hidden from the outside almost completely.

9.2.1 The Variable Swap

A central observation which forms the key idea in many dynamic reordering algorithms is that *two neighboring variables in the order can be swapped efficiently.* Of course, this statement is not valid in an unrestricted manner, but depends on the chosen implementation. For this reason, we refer in the following to the basic framework described in Chapter 7, upon which nearly all existing OBDD packages are based. For this framework we show how a swap of two neighboring variables in the order can be realized efficiently.

First, we assume that the variable x_i occurs immediately before the variable x_j in the order. The effect of this swap on each node labeled by x_i can be seen by applying Shannon's expansion with respect to x_i and x_j. If the function which is represented in a node with label x_i is denoted by f, then we have:

$$f = x_i x_j f_{11} + x_i \overline{x_j} f_{10} + \overline{x_i} x_j f_{01} + \overline{x_i}\ \overline{x_j} f_{00}.$$

By using commutativity we can order the terms so that that x_j occurs before x_i:

$$f = x_j x_i f_{11} + x_j \overline{x_i} f_{01} + \overline{x_j} x_i f_{10} + \overline{x_j}\ \overline{x_i} f_{00}.$$

In other words, the actual effect of the swap is the exchange of the two subfunctions f_{10} and f_{01} in the OBDD. Here, we have to take care that all unconcerned nodes in the graph are not affected by this exchange.

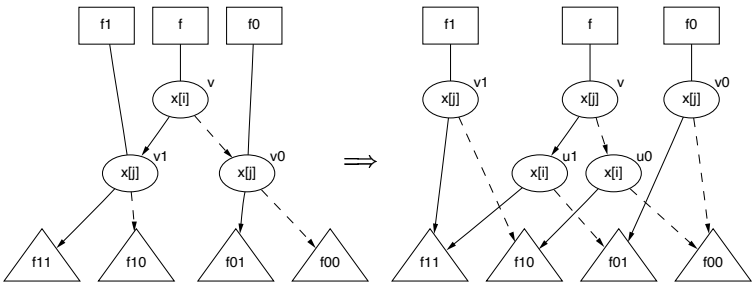

Figure 9.5. Swapping two neighboring variables

Figure 9.5 illustrates the swap of the two neighboring variables x_i and x_j within an arbitrary, possibly quite large OBDD. In the initial order the function f is represented by a node v labeled by x_i. First, we consider the case that the two sons v_1 and v_0 of v are labeled by x_j. The successor nodes of v_1 and v_0 represent the sub-OBDDs of the cofactors f_{11}, f_{10}, f_{01} and f_{00}. As f depends on the variable x_j, after the modification of the order this function has to be represented by a node with label x_j. The 1-successor of this node must possess references to the sub-OBDDs f_{11} and f_{01}, the 0-successor must have references to the sub-OBDDs f_{10} and f_{00}.

Note that the function f in Fig. 9.5 is represented by the same node v before and after the swap. Only the label and the outgoing edges of the node have been modified. This strategy guarantees that all existing references to f are not affected by the swap: neither the references which result from the upper levels in the OBDD, nor the references from outside the OBDD. As also the two cofactors f_1 and f_0 of f are represented by the original nodes v_1 and v_0 after the swap, *each* existing reference remains valid.

It is merely necessary to introduce the nodes u_1 and u_0 which represent the cofactors of f with respect to x_j. The figure seems to express that the size of an OBDD always increases during a variable swap. In case of a reduced representation, the equivalent status of x_i and x_j tell us that this cannot hold true. Indeed, there are even two reasons which reflect this general equivalence in the realization:

Nodes u_1, u_0: It is possible that the cofactors of f with respect to x_j are already represented in the original OBDD.

Nodes v_0, v_1: The preservation of the nodes v_1 and v_0 is only necessary if besides the original reference starting in the node v there is at least one other

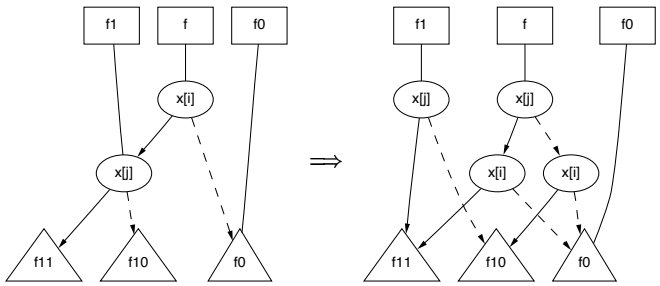

Figure 9.6. Special case of the variable swap where f_0 does not depend on x_j

reference. In a typical implementation these references cannot be efficiently determined, but the number of these references *can* be efficiently determined (see Section 7.1.6). If the reference counter of v_1 is zero after deleting the reference from the node v, then v_1 can be removed. The same holds true for v_0.

In the special cases where at least one of the two successor nodes of v is not labeled by x_j analogous constructions can be performed. For example, let the cofactor f_0 be independent of the variable x_j. By means of the construction in Fig. 9.6 the special case can be performed in such a way that all existing references remain valid.

Memory management. During performing a variable swap many dead nodes can arise. This suggests connecting the procedure directly with a garbage collection. That connection can be achieved as follows. Before starting an algorithm based on variable swaps a garbage collection is called, and the contents of the computed table is deleted such that all dead nodes are deleted. When swapping two neighboring variables x_i and x_j, only the reference counters of the nodes u_1 and u_0 may be decreased. If a reference counter reaches the value zero, then the node is removed immediately. In this way, it is guaranteed in a typical memory management framework that during dynamic reordering no dead nodes are carried along.

Complemented edges. In case of OBDDs with complemented edges the variable swap can be realized analogously. Here, it may happen at first that during the construction a 1-edge obtains the complement bit. But this can be corrected quickly and locally. We assume that not the subfunction f_{10} itself but instead its complement is represented. Hence, the edge to the sub-OBDD f_{10} carries the complement bit. Swapping the variables first leads to the graph in Fig. 9.7. The OBDD contains a complemented 1-edge which

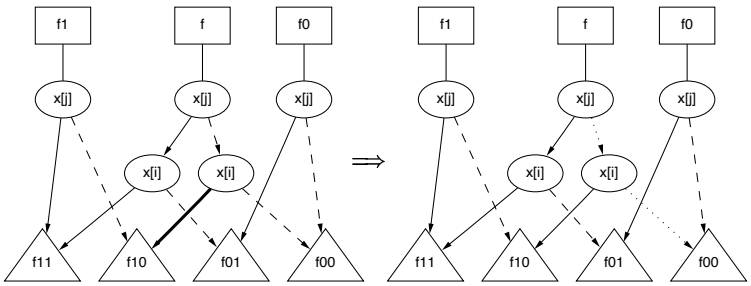

Figure 9.7. Variable swap in case of complemented edges

is drawn as a bold arrow. The figure also shows the transformation which serves to remove the complement bit from the 1-edge. Here, it is important that no unconcerned edge is affected by this transformation. At the end of the construction uniqueness of the representation has been re-established.

Time consumption. The efficiency of the variable swap substantially depends on the time needed for accessing the set of *all* nodes with label x_i. The time- and space-efficient framework in the form presented in Chapter 7 does not allow one to realize this access efficiently. However, a small modification can change this.

Let us recall that the access to the nodes is performed by means of a unique table. From each node with label x_i we have a very fast access to its sons, but not to all the other nodes with label x_i. However, by introducing a separate unique table for each variable x_i, this situation changes. Now, by using the collision lists we have fast access to the set of *all* nodes with label x_i. The required time for this access is

$$\mathcal{O}(\#lists + \#nodes),$$

where $\#lists$ is the number of collision lists in the unique table, and $\#nodes$ is the number of nodes in the table. Typically, the number of nodes is greater than the number of collision lists, so all nodes with label x_i can be visited in linear time.

As for each node with label x_i only constantly many operations are performed, the variable swap can be performed in linear time with regard to number of nodes with label x_i.