Chapter 5. Heuristic Minimization of Two-Level Circuits

If we do not request exact minimization, we get:

\[ p \ 0 \ 3 \ 0 \ 1 \ 0 \ 1 \ 0 \ 1 \ 0 \]

which has the same number of product terms, but one fewer literal. This can be explained by noting that the cost considered by espresso in exact minimization is just the number of terms or cubes. Therefore, espresso returns the first solution with three cubes that it finds.

On the other hand, when espresso is run in heuristic mode, it tries to maximally expand every cube. When \[0000001\] is maximally expanded, \[1010001\] is found to be a better solution than \[110001\], because it has one fewer literal.

Note that the solution found by the heuristic minimizer is better in terms of literals. However, the minimized function depends on three variables, \(x_2\), \(x_1\), and \(x_0\), instead of two. This may be sub-optimal, if long connections have to be drawn from where the inputs are available to where the function is placed in the layout of the circuit. Though in our simple circuit this is not likely to be the case, this problem gives an example of how abstracting the cost into a simple formulation—the number of cubes and the number of literals—may sometimes hide important details. It should be mentioned that it is possible to use as cost criterion the number of variables a function depends on. In that case we say that we minimize the support of the function.

\[ \]

Chapter 6

Binary Decision Diagrams (BDDs)

In this chapter we develop theory, algorithms, and data structures for the treatment of BDDs (Binary Decision Diagrams) and their applications. Along the way we discuss the relative advantages of Canonical and Non-Canonical Representations, and introduce the reader to BDDs by way of examples.

Many synthesis, verification, and testing algorithms manipulate large switching formulae. It is therefore important to have efficient ways of representing and manipulating such formulae. In recent times Binary Decision Diagrams (BDDs) have emerged as the representation of choice for many applications. Though BDDs are relatively old [169, 5], it was only with the work of Bryant [47], which brought out their advantages as canonical representations, that they began attracting the attention of many researchers.

The impact of BDDs has been enormous. Since 1986, when BDDs were unknown in synthesis circles, BDDs have penetrated virtually every subfield in the areas of synthesis and verification. As we shall see, BDDs have two remarkable properties. First, they are canonical, so if you correctly build the BDDs for two circuits, the two circuits are equivalent if and only if the BDDs are identical. This has led to significant breakthroughs in circuit optimization, testing, and equivalence checking. Second BDDs are amazingly effective at representing combinatorially large sets. This has led to stunning breakthroughs in FSM equivalence checking (and many other forms of formal verification [71]) and in two level logic minimization [79, 77, 86, 76]. This recent research is summarized below in Section 6.4.

Clearly a whole book can and should be written about BDDs. However, in this book we limit ourselves to a brief, introductory treatment in support of our discussion on "Symbolic" FSM equivalence checking. In our description, we shall follow [47] and [29], though with slightly different notation.

\[ ^1 \text{Public domain BDD packages are freely available, for example our own CUDD package — see Section 6.4.} \]
6.1 Representing Logic Functions with BDDs

We have already discussed several ways of representing a Boolean function, for example by Boolean formulae, or by the minterm or maxterm canonical form. The latter two forms are canonical. A form is canonical if the representation of a function in that form is unique.\(^2\)

A canonical form is desirable because it makes equivalence tests easy. From the definition, it follows that \(f_1\) and \(f_2\) have the same representation in a canonical form if and only if \(f_1 = f_2\). The minterm and maxterm canonical forms, however, have a serious drawback: The representations tend to be quite large (they are exponential in the number of variables). So they are not used except for the simplest cases.

Among the non-canonical forms, the formula type known as the sum of products (SOP) and the product of sums (POS) have been widely used. These two-level representations are discussed in detail in Section 4.3. They are two-level representations, and quite useful in synthesis and verification. However, there are also some fundamental difficulties with these forms.

- The two-level representations of some functions are too large to be practical (e.g., EXCLUSIVE-OR);
- Passing from SOP to POS and vice versa is difficult. As a consequence:
  - Taking the complement is difficult;
  - Taking the AND of two sums of products (or the OR of two products of sums) is difficult.
- Since SOP and POS are not canonical forms, answering the equivalence question for two functions is difficult.
- Furthermore, deciding whether a product of sums is satisfiable (a sum of product is a tautology) is NP-complete (coNP-complete).\(^3\)

In the next sections we discuss, and define formally, a canonical form—the BDD—that has the advantage of being compact for many functions and definitely superior to the other known canonical forms in that respect.

6.1.1 Binary Decision Diagrams by Way of Examples

We first introduce BDDs with the help of two examples and then give a formal definition. First, a BDD is a DAG (Directed Acyclic Graph — see Section 7.9.1, Page 305), such as the DAG shown at the right in Figure 6.1. Note the BDD nodes are in one-to-one correspondence with the gates of the MUX circuit at the left of the figure. This shows how BDDs can be viewed as a shorthand representation for MUX circuits, just as an SOP form can be viewed as a shorthand representation for an OR of ANDS.

\(^2\)We normally disregard reordering of terms.

\(^3\)The best known algorithms for these problems have exponential worst-case run times.

Second, let us consider the following function, given in SOP form:

\[
f = abc + b'd + c'd.
\]

A BDD for this function is given in Figure 6.2. If we want to know the value of \(f\) for a particular assignment to the variables \(a, b, c,\) and \(d,\) we just follow the corresponding path from the square box labeled \(f\) (this node is the root of the BDD\(^4\)). Suppose we want to know

\[
f(1, 0, 1, 0).
\]

The first variable encountered from the root is \(a,\) whose value is 1. We then follow the edge labeled \(T\) (which stands for then). We then come across a node labeled \(b.\)

\(^4\)In the following we will sometimes omit the root from the figures, when that does not generate confusion.
Since the value of \( b \) is 0, we follow the edge labeled \( E \) (else). The next node is labeled \( d \), which implies that for \( a = 1 \) and \( b = 0 \), the value of \( f \) does not depend on \( c \). Following the \( E \) edge we finally reach the leaf labeled 0. This tells us the value of the function is 0, as can be easily verified from the SOP formula.

The BDD of Figure 6.2 is an ordered binary decision diagram, because the variables appear in the same order along all paths from the root to the leaves. The ordering in this case is

\[ a \leq b \leq c \leq d. \]

The appearance and the size of the BDD depend on the variable ordering. This is illustrated in Figure 6.3, where a different BDD for \( f \) is given according to the following variable ordering:

\[ a \leq d \leq b \leq c. \]

Finally, for the ordering

\[ b \leq c \leq a \leq d, \]

we obtain the BDD of Figure 6.4. This is an optimal ordering, since there is exactly one node for each variable. Whenever not otherwise specified, we shall assume that our BDDs are ordered. Figure 6.5 gives the BDDs for some elementary functions. Notice the similarity of the BDDs for \( f = a \) and \( f = a' \). One is obtained from the other by swapping the two terminal nodes.

### 6.1.2 Formal Definition of BDDs

We now give a formal definition of a binary decision diagram. We shall then outline the algorithms for BDD manipulation and finally, based on the requirements of those algorithms, we shall devise the data structures and the details of the algorithms.

**Definition 6.1.1** A BDD is a directed acyclic graph \( (V \cup \Phi \cup \{1, E\}, E) \) representing a multiple-output switching function \( F \). The nodes are partitioned into three subsets. 
\( V \) is the set of the internal nodes. The output of \( v \in V \) is 2. Every node \( v \) has a label \( l(v) \in \Phi \). Here \( \Phi = \{z_1, \ldots, z_n\} \) denotes the support of \( F \), i.e., the set of variables on which \( F \) actually depends. Thus, \( l(v) \) is one of the variables variable \( \{x_i\} \). The terminal node's outdegree is 0. \( \Phi \) is the set of the function nodes: The outdegree of \( \Phi \) is 1 and its in degree is 0. The function nodes are in one-to-one correspondence with the components of \( F \). The outgoing edges of function nodes may have the complement attribute. The two outgoing edges for a node \( v \in V \) are labeled \( T \) and \( E \), respectively. The \( E \) edge may have the complement attribute. We use \( l(v), T, E \) to indicate an internal node and its two outgoing edges. The variables in \( \Phi \) are ordered and if \( u_j \) is a descendant of \( u_i \) (\( u_i, u_j \in V \)), then \( l(u) < l(v) \). The function \( F \) represented by a BDD is defined as follows:

1. The function of the terminal node is the constant function 1.
2. The function of an edge is the function of the head node, unless the edge has the complement attribute, in which case the function of the edge is the complement of the function of the node.
3. The function of a node \( v \in V \) is given by \( l(v) T_r + l(v) E_r \), where \( T_r \) (\( E_r \)) is the function of the \( T \) (\( E \)) edge.
4. The function of \( \Phi \) is the function of its outgoing edge.

An edge with (without) the attribute is called a complement (regular) edge.

BDDs are canonical (the representation of \( F \) is unique for a given variable ordering) if:

\[ f(a, b, c, d, \ldots ) = \begin{cases} 1 & \text{if } a = 1, b = 0, c = 0, d = 1. \\ 0 & \text{otherwise}. \end{cases} \]
6.1.3 How to Build the BDD for $f$

BDDS can be built from recursive use of Boole's expansion theorem (See Section 3.3.3). We shall see that the expansion theorem plays a central role in the definition and manipulation of BDDS.

As an example, consider how the BDD is built for

$$f = abc + \overline{b}d + c'd$$

under the variable ordering: $b \leq c \leq d \leq a$.

We start by computing the cofactors of $f$ with respect to $b$, the first variable in the ordering. We get:

$$f_b = ac + c'd$$

and

$$f'_b = d + c'd.$$  

We can summarize this initial result by a partial diagram as the one of Figure 6.6. It is true in general that the two children of a node represent the two cofactors of the function represented by the node with respect to the variable labeling the node. We then compute the cofactors of $f_b$ and $f'_b$ with respect to $c$. This yields:

$$(f_b)_c = f_{bc} = a \quad f'_{bc} = d$$

$$f_{bc} = d \quad f'_{bc} = d.$$  

We observe that three of these four cofactors are identical. Hence we create a single node for them in the new partial BDD, shown in Figure 6.7. Recognizing that some
cofactors are identical guarantees that the BDD will be reduced (intuitively, it does not contain duplicated and superfluous nodes). This is an important property, as we shall see.

Finally, noting that

\[(x_i)_x = 1 \quad \text{and} \quad (x_i)_y = 0\]

and

\[(x_i)_y = (x_j)_y = z_i \quad \text{for} \quad i \neq j\]

we get the BDD of Figure 6.8. Notice the similarity to the BDD of Figure 6.4. This example illustrates that the optimal variable order is not unique in general.

6.1.4 Reduced BDDs

Notice that if we had not identified the identical cofactors, we would have obtained the tree of Figure 6.9. This BDD, unlike the previous ones, is not reduced. There are isomorphic subgraphs. A non-reduced BDD can be systematically transformed into a reduced one. Consider the two subgraphs highlighted in Figure 6.10. They represent the same function and therefore they can be merged, as shown in Figure 6.11. We now notice that the node pointed by the arrow is redundant (it corresponds to no decision), hence it can be removed. This is shown in Figure 6.12. By iteratively applying:

- Identification of isomorphic subgraphs;
- Removal of redundant nodes;

we obtain the initial reduced graph.

Given an ordering, the reduced graph for a function is unique. Hence, the Reduced Ordered BDD (ROBDD) is a canonical form. This is the first important property of binary decision diagrams, that is extremely useful for verification. Two functions are equivalent if and only if they have the same BDD.

Other interesting properties of BDDs are:
Chapter 6. Binary Decision Diagrams (BDDs)

6.1 Representing Logic Functions with BDDs

Figure 6.10: Two isomorphic subgraphs.

Figure 6.11: Merging two isomorphic subgraphs.

Figure 6.12: Elimination of a redundant node.

- The size of the BDD (the number of nodes) is exponential in the number of variables in the worst case (e.g., multipliers); however, BDDs are well-behaved for many functions that are not amenable to two-level representations (e.g., EXCLUSIVE-OR).

- The logical AND and OR of BDDs have the same complexity (polynomial in the size of the operands). Complementation is inexpensive.

- Both satisfiability and tautology can be solved in constant time. Indeed, a formula is a tautology if and only if its BDD consists of the terminal node 1.

- Covering problems can be solved in time linear in the size of the BDD representing the constraints.

On the other side:

- BDD sizes depend on the ordering. Finding a good ordering is not always simple.

- There are functions for which the SOP or POS representations are more compact than the BDDs. Unfortunately, many constraint functions of covering problems fall into this category.

- In some cases SOP/POS forms are closer to the final implementation of a circuit. For instance, if we want to implement a PLA, we need to generate at some point a SOP or POS form.
6.1.5 Why Ordering is Important

Before looking in more detail at the manipulation of BDDs, let us try to better understand why ordering is important. We consider

\[ f = ab + cd + ef \]

with ordering

\[ a \leq b \leq c \leq d \leq e \leq f. \]

The BDD is shown in Figure 6.13. Let us now considering the ordering

\[ a \leq c \leq e \leq b \leq d \leq f. \]

The resulting BDD, shown in Figure 6.14, is considerably more complex. (In that BDD, the \( E \) edges go to the left.) The reason for the big difference is the following. In the decision making process that eventually gives us the value of the function for a given assignment, we follow two opposite strategies depending on the ordering.

With the first ordering we consider one product term at the time. After the first two variables have been examined, we know whether the first product term \((ab)\) is 0 or 1. If it is 1, we are done. If it is not, we just have to remember that it evaluated to 0. We don’t need to know which variable \((a\) or \(b\)) caused it to be 0. After the first four variables have been considered, we just need to remember whether either product term evaluated to 1, or whether both were 0. The specifics are not important to determine the value of the function. Since we have very little to remember, we get by with very few nodes.

With the second ordering, we process all product terms “in parallel.” If the first three variables are all 1’s, we cannot tell the value of any product term. In addition, the value of \( a \) along any given path must be remembered until \( b \) is met—three levels below. Similarly for \( c \) and \( e \). This fact prevents the recombination of different paths.

We can imagine a bit-serial processor that examines the values of the variables one at the time. The size of the BDD is related to the amount of information that must be stored to compute the result. We shall return to these considerations when we consider algorithms for finding optimal—or simply good—orderings.

6.2 Design Considerations for a BDD Package

We have seen how switching functions can be represented as ordered reduced BDDs. We now consider the efficient implementation of BDDs, in terms of both memory and CPU. Before we proceed in detailing data structures and algorithms, and before we give a formal definition, we need to add a few design considerations. We shall closely follow [29].

Shared BDDs. We have seen that each node of a BDD has a function associated with it. If we have several functions, chances are that they will have subexpressions in common. For instance, if we have \( f_1 = b + c \) and \( f_2 = a + b + c \), we would like to represent them like in Figure 6.15. As a special case, two equivalent functions could be represented by the same BDD (not just two identical BDDs). This amounts to dealing with a single multi-rooted directed acyclic graph (DAG) with a root for each function we are explicitly interested in. All functions share the same DAG.

Unique Table. We are ultimately interested in reduced BDDs. Rather than generating non-reduced BDDs and then reducing them, we are interested in guaranteeing that at any time there are no isomorphic subgraphs and no redundant nodes in the multi-rooted DAG. This can be achieved by checking for the existence of a node representing the function we want to add, prior to the creation of a new node. A straightforward approach would consist of searching the whole DAG every time we want to insert a new node. However, that would be far too inefficient. Instead, we
shall keep a dictionary of the functions represented in the DAG. This dictionary is called unique table and is best implemented as a hash table.

**Strong Canonicity.** Because of the unique table, two equivalent functions end up sharing exactly the same subgraph. Hence checking for equivalence just requires checking that the pointers in the DAG associated with the two functions are identical. This property is called strong canonicity and makes constant-time equivalence check possible—a very desirable consequence.

**Attributed Edges.** We have seen that the BDDs for $f$ and $f'$ are very similar. The only difference being the values of the leaves that are interchanged. This suggests the possibility of actually using the same subgraph to represent both $f$ and $f'$. Suppose the BDD actually represents $f$. If we are interested in $g = f'$ it is then sufficient to remember that the function we have in the multi-rooted graph is the complement of $g$. This can be accomplished by attaching an attribute to the edge pointing to the top node of $f$. An edge with the complement attribute is called a complement edge. The edges without the attributes are called regular edges. The use of complement edges slightly complicates the manipulation of the BDDs, but has two advantages. Obviously, it decreases the memory requirements. However, the most important consequence of using complement edges is that the fact that complementation can be done in constant time—the BDD is already in place—and checking two functions for one being the complement of the other also takes constant time. Note that with complement edges we need only one constant function (we choose 1) and hence only one leaf in the multi-rooted DAG. The attribute mechanism is quite general: Other attributes have been used for other purposes [201, 192].

---

### 6.3 Algorithms

**Computed Table.** As a speed improvement device, we shall keep a table of recently computed functions. The purpose of this table is different from that of the unique table. With the unique table we answer questions like: "Does there exist a node labeled $v$ with children $g$ and $h$?" On the other hand, the computed table answers questions like: "Did we recently compute the AND of $f_1$ and $f_2$?" We can ask this question before we actually know that the AND of $f_1$ and $f_2$ is a function whose top node is labeled $v$ and whose children are $g$ and $h$. Hence we can avoid recomputing the result.

**Memory Management and Dynamic Re-Ordering.** In a typical application we build and then dispose of many BDDs. An efficient memory management is important. We shall adopt a strategy based on garbage collection, i.e., we shall not immediately free nodes that are no longer used. Instead, from time to time we shall visit our data structure to recover all the unused memory.

Also, an *a priori* ordering of the shared BDDs may continue to grow as the BDDs are manipulated in a particular application. If this growth is allowed to go unchecked, it may (and often does, for large problems) occur that we run out of memory. In many cases this can be alleviated by dynamically re-ordering the BDD variables. This can be quite dramatic for some circuits.

Both of these mechanisms are essential to the operation of a robust BDD package.

---

### 6.3 Algorithms

We now outline the algorithms for BDD manipulation. Then, based on the requirements of those algorithms, we shall devise appropriate data structures and give the details of the algorithms.

The usual way of generating new BDDs is to combine existing BDDs with connectives like AND, OR, EX-OR. As a starting point one generates the simple BDDs for the functions $f_1 = x_1$, for all the variables in the functions of interest. We are therefore interested in an algorithm that, given BDDs for $f$ and $g$, will build the BDD for $f \lor g$, where $(\lor)$ is a binary connective (a switching function of two arguments).

The basic idea comes—not surprisingly—from the expansion theorem, since:

$$f \lor g = u(f_1(\lor)g_1) + u(f_2(\lor)g_2).$$

So, if $v$ is the top variable of $f$ and $g$, we can first cofactor the two functions with respect to $v$ and solve two simpler problems recursively, and then create a node labeled $v$ that points to the results of the two subproblems (if such a node does not exist yet; otherwise we just return the existing node).

Finding the cofactors of $f$ and $g$ with respect to $v$ is easy: If $f$ does not depend on $v$, $f_v = f$, that is, the cofactors are the function itself. If, on the other hand, $v$ is the top variable of $f$, the two cofactors are the two children of the top node of $f$. Similarly for $g$.
### Table 6.16: Two argument operators expressed in terms of ITE.

<table>
<thead>
<tr>
<th>Table</th>
<th>Name</th>
<th>Expression</th>
<th>Equivalent Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0001</td>
<td>AND(F,G)</td>
<td>F · G</td>
<td>ITE(F,G,0)</td>
</tr>
<tr>
<td>0010</td>
<td>F &gt; G</td>
<td>F · G'</td>
<td>ITE(F,G',0)</td>
</tr>
<tr>
<td>0011</td>
<td>F</td>
<td>F</td>
<td>ITE(F,0,1)</td>
</tr>
<tr>
<td>0100</td>
<td>F &lt; G</td>
<td>F' · G</td>
<td>ITE(F,0,0)</td>
</tr>
<tr>
<td>0101</td>
<td>G</td>
<td>G</td>
<td>ITE(G,0,1)</td>
</tr>
<tr>
<td>0110</td>
<td>XOR(F,G)</td>
<td>F ⊕ G</td>
<td>ITE(F,G',G')</td>
</tr>
<tr>
<td>0111</td>
<td>OR(F,G)</td>
<td>F + G</td>
<td>ITE(F,G,G')</td>
</tr>
<tr>
<td>1000</td>
<td>NOR(F,G)</td>
<td>(F + G)'</td>
<td>ITE(F,0,G')</td>
</tr>
<tr>
<td>1001</td>
<td>XNOR(F,G)</td>
<td>(F ⊕ G)'</td>
<td>ITE(F,G,G')</td>
</tr>
<tr>
<td>1010</td>
<td>NOT(G)</td>
<td>G'</td>
<td>ITE(G,0,1)</td>
</tr>
<tr>
<td>1011</td>
<td>F ≥ G</td>
<td>F + G'</td>
<td>ITE(F,1,G')</td>
</tr>
<tr>
<td>1100</td>
<td>NOT(F)</td>
<td>F'</td>
<td>ITE(F,0,1)</td>
</tr>
<tr>
<td>1101</td>
<td>F ≤ G</td>
<td>F' + G'</td>
<td>ITE(F,G,1)</td>
</tr>
<tr>
<td>1110</td>
<td>NAND(F,G)</td>
<td>(F · G)'</td>
<td>ITE(F,G',1)</td>
</tr>
<tr>
<td>1111</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

The approach based on the expansion theorem\(^7\) can be further improved by considering the if-then-else operator `ITE`, which is a tertiary operator defined as follows:

\[
ITE(F,G,H) = F \cdot G + F' \cdot H,
\]

where \(F, G, H\) are three arbitrary switching functions. One interesting property of the `ITE` operator is that it is a two-argument operator that can be expressed in terms of it, as shown in Figure 6.16. Therefore, most of the standard manipulations of BDDs can be done with `ITE`.

### 6.3. Algorithms

Whenever we encounter one of these cases, we just return the pointer to \(F\). Otherwise we find the top variable of the three functions and apply the recursive formula. This is the basic idea of the algorithm. However, before we examine the pseudo-code, we need to consider some of the design issues of Section 6.2. In particular, we need to consider the unique table and the computed table.

As we said, we use a hash table for the unique table, i.e., a data structure that stores an item in a location of a table identified by its key. The key in our case is the tuple \((v, G, H)\) that identifies a node in the DAG. In the key, \(v\) is an integer—the index of the variable—while \(G\) and \(H\) are pointers. The hashing function maps the key into the location in the table. The mapping is not one-to-one, so that several keys may correspond to the same location in the table, thereby producing collisions. All colliding entries are kept in a linked list called the collision chain. A typical hashing function may shift \(v, G,\) and \(H\) by different numbers of bits, add the results, and finally compute the remainder of the sum on division by the size of the table.\(^8\)

If we want to see if a given key is present in the hash table, we compute the hashing function with the key as argument. This identifies one entry in the table, i.e., a collision chain. We then examine all the elements in the collision chain until we find one that matches our entry or until all elements have been examined.\(^9\) The advantage of a hash table is that the collision chains are short if the table is large enough and if the hashing function is good; therefore search is performed in expected constant time.

In the unique table, the pointer to the node with label \(v\) and children \(F\) and \(G\) is found in the collision list corresponding to the key \((v, G, H)\). In the recursive algorithm, we first find \(G\) and \(H\) recursively, and then ask whether we should introduce a new node labeled \(v\) and pointing to \(G\) and \(H\). The unique table tells us if such a node already exists. If not, a new entry is created in the table.

For the computed table we initially make the assumption that we also use a hash table. This corresponds to assuming infinite memory, which is a convenient assumption for our purpose of analyzing the qualitative operation and complexity of the algorithm.\(^10\) The key for the computed table is the tuple \((F, G, H)\). At every level of the recursion, we check the computed table for the result. If it is already there, we just return. If not we recur and, before returning, we insert the newly computed result in the table.

We are now ready to look at the pseudo-code of the `ITE` algorithm, reported in Figure 6.17. For the time being, we ignore the complement edges.

Notice how the algorithm maintains the BDD reduced by checking if \(T\) equals \(E\) and by consulting the unique table. An example of application of the algorithm is in Figure 6.18. The result consists of two newly created nodes and two already existing nodes. (Those labeled \(C\) and \(D\).) This is a fairly typical situation.

In terms of complexity, let us first examine what would happen without the computed table. Assuming the access to the unique table take constant time, then all operations performed by the algorithm require constant time, except for the recursive

\(^{7}\) The size of the table should be a prime number for best results. Another approach uses \(v\) to select a sub-table and then \(G\) and \(H\) to select a position in the sub-table.

\(^{8}\) This particular scheme is called open hashing. Hash tables are treated in [1, 69].

\(^{9}\) This is not realistic in practice. However, management of the computed table is intricate, and beyond the scope of this text.
ITE\((F, G, H)\) {
  \(\langle \text{result, terminal.case} \rangle = \text{TERMINAL\_CASE}(F, G, H)\)
  if (terminal.case) {
    return (result)
  }
  \(\langle \text{result, \text{computed.table}} \rangle = \text{COMPUTED\_TABLE\_HAS\_ENTRY}(F, G, H)\)
  if (\text{computed.table}) {
    return (result)
  }

  \(v = \text{TOP\_VARIABLE}(F, G, H)\)
  \(T = \text{ITE}(F_v, G_v, H_v)\)
  \(E = \text{ITE}(F_{\overline{v}}, G_{\overline{v}}, H_{\overline{v}})\)
  if \((T = E)\) return \((T)\)

  \(R = \text{FIND\_OR\_ADD\_UNIQUE\_TABLE}(v, T, E)\)
  \(\text{INSERT\_COMPUTED\_TABLE}(F, G, H, R)\)
  return \((R)\)
}

Figure 6.17: Pseudo-code of the ITE algorithm.

\[ I = \text{ITE}(F, G, H) \]
\[ = \langle a, \text{ITE}(F_a, G_a, H_a), \text{ITE}(F_{\overline{a}}, G_{\overline{a}}, H_{\overline{a}}) \rangle \]
\[ = \langle a, \text{ITE}(1, C, H), \text{ITE}(B, 0, H) \rangle \]
\[ = \langle a, C, \langle 0, \text{ITE}(B_0, 0, H_0), \text{ITE}(B_{\overline{0}}, 0, H_{\overline{0}}) \rangle \rangle \]
\[ = \langle a, C, \langle 0, \text{ITE}(1, 0, 1), \text{ITE}(0, 0, D) \rangle \rangle \]
\[ = \langle a, C, \langle 0, 0, D \rangle \rangle \]

Figure 6.18: Example of application of ITE.

6.3. Algorithms

Figure 6.19: Equivalent pairs of functions.

calls. However, every call to the procedure generates two other calls unless we are in a terminal case, so that the total number of calls—and hence the execution time—is exponential in the number of variables. However, if we consider the computed table, things change dramatically. Let \(|F|\) be the number of nodes in the BDD for \(F\). The effect of the computed table is to cause ITE to be called at most once for each distinct combination of nodes in \(F, G,\) and \(H\). Hence, ITE can be called \([|F| \cdot |G| \cdot |H|]\) times at most and the complexity of the procedure is \(O([|F| \cdot |G| \cdot |H|])\). Note that when using ITE to compute the AND of two functions, one of the arguments to ITE is 0; in such a case the complexity is quadratic in the size of the operands. In practice performance is normally better than quadratic and typically closer to the size of the resulting BDD. In the worst case, \([|F| \cdot |G|]\) is comparable to \([|F| \cdot |G|]\). However, in many cases, \([|F| \cdot |G|] \ll [|F| \cdot |G|]\). In these cases the time taken by ITE is typically proportional to \([|F| \cdot |G|]\).

If we remove the infinite memory assumption, as we do in practice, the worst-case complexity returns to be exponential. However, the typical performance is not affected significantly by the loss of infinite memory, if the computed table is implemented carefully.

6.3.2 Complement Edges

We mentioned that edges may carry attributes that specify how to derive the function of the edge from the function of the node it points to. The most common attribute is the complement. It allows the BDDs for \(f\) and \(f'\) to be shared, thus saving space and making complementation a constant-time operation. We use a dot to indicate a complement edge.

To maintain canonicity, we must constrain where complement edges can be used. The four pairs of functions of Figure 6.19 are equivalent. Each pair consists of two functions that can be obtained from each other by application of the identity

\[ f' = v \cdot f'_0 + \overline{v} \cdot f'_1, \]

whence

\[ f = (v \cdot f'_0 + \overline{v} \cdot f'_1)'. \]

Also, the functions represented in Figure 6.19 represent all the eight possible ways of placing dots on the incoming and outgoing edges of a node. To guarantee canonicity, we impose that the then edge of every node be regular. Notice that this is true of exactly one function for each of the four pairs of Figure 6.19. It will be the task of the algorithms, ITE in particular, to ensure that every non-canonical case be transformed into the corresponding canonical case. The overhead is indeed minimal. Since the
dote may only appear on dolc edges, in the sequel we shall adopt the convenion of marking an elc edge with an empty circle if it is regular and with a dot if it is complemented. The then edges will be unmarked. The outgoing edges of function nodes may have the complement attribute. In that case they will have a dot. We shall also convene that dangling edges point to the constant node, unless otherwise stated.

The use of complement edges makes testing for another special case of ITE possible. Indeed,
\[ \text{ITE}(F, 0, 1) = F'. \]

Though this is true in general, without complement edges we could not terminate the recursion because the BDD for \( F' \) would not necessarily exist and could not be easily found if it existed. Another advantage of complement edges, to be discussed in Section 6.3.3, is the ability to use a single entry of the computed table for both \( F \) and \( F' \).

We end this section with one observation that is useful when drawing BDDs by hand. Since the then edges are never complemented, the value of a function for 1, 1, \ldots, 1 will be 0 if and only if the outgoing edge of the function node is complemented. Computing the value of the function for which a BDD must be drawn when all variables equal 1 allows one to determine how to start (with a regular or complement edge) and normally prevents one from pushing too many complement dots around.

6.3.3 The Computed Table

The computed table stores results of recent computations. Under the assumption of infinite memory—all previously computed results are stored—we have seen that the computed table makes the complexity of ITE polynomial in the size of the operand BDDs. In general memory is at a premium; it is important to implement the computed table efficiently. However, discussion of this crucial issue is beyond the scope of this text.

6.3.4 Conditioning of the ITE Calls

For every triple \( (F, G, H) \) there are other triples \( (F_1, G_1, H_1) \) such that \( \text{ITE}(F, G, H) = \text{ITE}(F_1, G_1, H_1) \) though at least one of the three arguments is different. For instance, if
\[
F = z_1 + z_2 z_4, \quad G = (z_1 + z_2) z_4, \quad H = 0, \]
and
\[
F_1 = z_2 z_4 (z_1 + z_2), \quad G_1 = 1, \quad H_1 = z_2 z_2 z_4, \]
then
\[ \text{ITE}(F, G, H) = \text{ITE}(F_1, G_1, H_1) = z_2 z_4. \]

If we could identify all triples that give the same result, we could map all of them into the same entry of the computed table. In general, finding all such triples is too expensive, but there are special cases that can be identified with very little effort. For instance,
\[ \text{ITE}(F, G, F) = \text{ITE}(F, G, 0) = \text{ITE}(G, F, G) = \text{ITE}(G, F, 0) = F \cdot G. \]

For these special cases, we want to transform the triple into a standard form before looking up the computed table. In this way, all the triples that map into the same standard form share the same entry. This saves both memory and time.

A mapping of a triple into another can be based on the identification of the following occurrences:

- Two arguments are the same function (e.g., \( \text{ITE}(F, F, G) \));
- Two arguments are one the complement of the other (e.g., \( \text{ITE}(F, G, F') \));
- One or more arguments are constants (e.g., \( \text{ITE}(F, G, 0) \)).

Note that these checks can be performed in constant time. A first set of transformations replaces as many arguments as possible with constants:
\[
\begin{align*}
\text{ITE}(F, F, G) & \Rightarrow \text{ITE}(F, 1, G) \\
\text{ITE}(F, G, F) & \Rightarrow \text{ITE}(F, G, 0) \\
\text{ITE}(F, G, F') & \Rightarrow \text{ITE}(F, G, 1) \\
\text{ITE}(F', F', G) & \Rightarrow \text{ITE}(F, 0, G).
\end{align*}
\]

A second set of transformations permutes the arguments based on the following equalities.
\[
\begin{align*}
\text{ITE}(F, 1, G) & = \text{ITE}(G, 1, F) \\
\text{ITE}(F, G, 0) & = \text{ITE}(G, F, 0) \\
\text{ITE}(F, G, 1) & = \text{ITE}(G', F', 1) \\
\text{ITE}(F, 0, G) & = \text{ITE}(G', 0, F') \\
\text{ITE}(F, G, G') & = \text{ITE}(G, F, F').
\end{align*}
\]

If one of these cases occurs, we choose the triple with the first argument having the smallest index for its top variable. For instance, if the top variable of \( F \) is \( z_3 \) and the top variable of \( G \) is \( z_4 \), we prefer \( (F, G, 1) \) to \( (G', F', 1) \). In case of a tie, we compare the addresses of the top nodes of the first arguments—they are guaranteed to be different—and choose the triple with the lower address.

A third and final set of transformations is based on the following equalities.
\[
\begin{align*}
\text{ITE}(F, G, H) & = \text{ITE}(F', H, G) = \overline{\text{ITE}}(F', G', H') = \overline{\text{ITE}}(F', H', G').
\end{align*}
\]

As in the other cases, we want to choose only one of the equivalent forms and map the others onto it. We notice that there is only one form such that the first two arguments should be pointed via regular edges. For instance, if \( F \) is regular and \( G \) is complement, \( \text{ITE} \) will replace \( (F, G, H) \) by \( (G', F', H') \) and take the complement of the result before returning. If, on the other hand, \( F \) is complement and \( H \) is regular, \( \text{ITE} \) will replace \( (F, G, H) \) by \( (F', H, G) \), and will not take the complement of the result before returning.

Among the effects of these transformations is the ability to detect that \( F \cdot G = (F' + G') \), i.e., the ability to apply De Morgan's laws. This demonstrates the power and flexibility of the approach based on the ITE operator.
6.3. Algorithms

6.3.5 The Ite.constant Algorithm

It is often the case that one wants to check whether, for two functions $F$ and $G$, $F \leq G$, or equivalently $F \Rightarrow G$, holds. This amounts to checking if $F + G$ is identically one. We could use Ite to compute $F + G$ and then test the result for being the tautology. However, there is a more efficient way of proceeding, which avoids building the intermediate result; it is based on a specialized version of Ite called Ite.constant, and is outlined in Figure 6.20. The check for implication $F \Rightarrow G$ can then be performed by computing Ite.constant($F, G, 1$). The procedure returns one of three values: 1, 0, and non_constant. It is based on the observation that for the resulting function to be constant, the two cofactors must be identical and constant. As soon as a violation of this condition is detected, Ite.constant returns non_constant. This early termination in unsuccessful cases, combined with not building intermediate results, makes this procedure much faster than the general approach. Notice that the speed-up occurs only for cases where $F \neq G$. Hence, the speed-up observed in practice depends on the problem at hand. A typical value could be 20.

As an example of Ite.constant, consider the following functions.

$$f = ab + bcd$$
$$g = ab + ac'd$$

We are interested in checking whether $f \Rightarrow g$. This translates into checking whether Ite.constant($f, g, 1$) = 1. The BDD for these functions and the computation are given in Figure 6.21. Note that unlike Ite, the fact that the two cofactors of the
Chapter 6. Binary Decision Diagrams (BDDs)

6.4 Notes

Next, if we wanted to compute $\text{ITE} \cdot \text{CONSTANT}(f, g, h)$, we would proceed as follows.

\[
\text{ITE} \cdot \text{CONSTANT}(a, \text{ITE} \cdot \text{CONSTANT}(p, g, h), \text{ITE} \cdot \text{CONSTANT}(c, g, h)) = \\
(a, (b, \text{ITE} \cdot \text{CONSTANT}(1, q, h), \text{ITE} \cdot \text{CONSTANT}(c, d, h)), \text{ITE} \cdot \text{CONSTANT}(c, g, h)) = \\
(a, (b, \text{non-constant}, \text{ITE} \cdot \text{CONSTANT}(c, d, h)), \text{ITE} \cdot \text{CONSTANT}(c, g, h)).
\]

and then all pending calls return non-constant.

6.4 Notes

There has been a surprisingly steady influx of "new" types of DDDs (Decision Diagrams). The work of Bryant in [47] focused on Boolean functions and decision diagrams that were based on the Boole-Shannon expansion of Section 3.3.3 and specific types of reduction to canonical form. However, some new forms of DDDs focus on Pseudo-Boolean Functions have been introduced: [67] (called MTBBDDs), [11], [12, 131, 132] (called ADDSs), [165] (called EVBBDDs), [46] (called BDDs).

These works have successfully extended the applicability of DDDs into the realm of functions with Boolean domains but integer or real valued ranges. This has opened up new vistas for research in Markov Analysis and Probabilistic Verification. Shortest Path Analysis on graphs of vast size, "Exact" (that is free of false paths) timing analysis of combinational circuits, low power synthesis, technology mapping, etc.

Other work on extending the applicability of DDDs has focused on the reduction to canonicity. Different reduction or ordering rules have been studied: [200] (Zero-Suppressed BDDs), [23] (Free BDDs), [91] (Functional DDDs), [218] (Extended Decision Diagrams). Some or all of these variants can be mixed into a single "Hybrib Decision Diagram". Zhao and Clarke of CMU have developed this DDD in a package for word level verification of sequential arithmetic circuits which has won internal awards at INTEL.

A serious issue with DDDs is ordering. In every type of DDD, the variables in the support of the function being represented must be ordered to obtain canonicity. Unfortunately, in many cases, the size of the BDD depends critically on the specific ordering chosen. This dependence is so drastic that in many cases it is impractical to build the BDD at all. For example it is known that BDD size grows exponentially with variable count, independent of the order. This limits the use of BDDs for multipliers to 16 bits or less.

An exact algorithm for BDD variable ordering is the improved version by Ishihara et al. [150] of Friedman and Supowit's [104]. Various iterative techniques are discussed in [106, 150, 237, 207, 206]. Heuristic techniques to find a good order for a circuit are discussed in [105, 163, 201, 262]. A comparison of the performance of various methods can be found in [154]. Partial BDDs have been studied in [230, 53]. Other references on variable ordering are: [269, 102, 48, 96, 90].

At the time of writing, several BDD packages of high quality were in use. Rudell's package was proprietary to Synopsys. David Long's package, developed at CMU, is employed in SMV (CMU verification package), VIS (UC Berkeley Verification Package) and SIS (UC Berkeley Synthesis Package) as well as AT&T and other industrial sites. A more recent entry into the BDD package field is the CUDD package from the University of Colorado at Boulder. This package is used in VIS and in

---

**Figure 6.22: BDDs f, g, h and $\text{ITE}(f, g, h)$.**

The result are computed serially does matter. This is taken into account in Figure 6.21 by not expanding $\text{ITE} \cdot \text{CONSTANT}(q', 0, 1)$ until the result of the other branch is known. The computation of $\text{ITE} \cdot \text{CONSTANT}(q', 0, 1)$ itself is a special case, since $q'$ is not constant and therefore $q' \cdot 0 + q' \cdot 1 = q$ is not constant either.

The correctness of the result can be verified by observing that for $a \cdot b \cdot c \cdot d = f$, $g = 1$ and $h = 0$. The procedure, however, does not even need to find such an example.

We conclude with a more meaningful example, which shows how to deal with complement edges in ITE and ITE\$CONSTANT. Consider the functions

\[
f = a + c,
\]

\[
g = b + d,
\]

\[
h = c + d,
\]

with the variable ordering

\[
a \leq b \leq c \leq d.
\]

Now suppose that we are given the shared BDD (with complement edges) for $f$, $g$, $h$, and need to compute the BDD for $\text{ITE}(f, g, h)$. The results, shown in Figure 6.22, are obtained as follows.

\[
\text{ITE}(f, g, h) = (a, \text{ITE}(p, g, h), \text{ITE}(c, g, h)) = \\
(a, (b, \text{ITE}(1, q, h), \text{ITE}(c, d, h)), (b, \text{ITE}(c, q, h), \text{ITE}(c, d, h))) = \\
(a, (b, \text{ITE}(c, d, h), (b, \text{ITE}(c, d, h), (c, d, h)))) = \\
(a, (b, \text{ITE}(c, d, h), (c, d, h))).
\]

Specifically, it matters for the number of recursive calls. It does not affect the final result.
Motorola's VERDICT verification package as well as at several university sites. Of all of these, the CUDD package has the most efficient dynamic reordering package, and goes the farthest in supporting diverse BDD types (it currently supports BDDs, ADDs/MTBDDs, and ZDDs (zero-suppressed BDDs)).

The CUDD package is available on the web via anonymous FTP at

\texttt{vlsi.colorado.edu}

(login as anonymous, give your email address as password, and then type "cd pub").

6.5 Summary

In this chapter, we have introduced BDDs and summarized briefly the enormous impact they have had on the merging fields of synthesis and verification. We have shown how to build and manipulate them (see the solved problems for applying the usual Boolean operators to two functions represented by BDDs). We have shown how an ordered BDD is made canonical by imposing certain reduction rules.

We have studied the qualitative operation of the ITE and ITE\_CONSTANT algorithms, and show how many of the BDD manipulations that might be needed in practice can be done efficiently, given efficient implementation of these algorithms.

We have avoided the intricacies of memory management, but they are all-important to the robustness of a BDD package\footnote{Detailed notes on this subject are available from the authors.}.

We have given enough background so that the discussion of FSM traversal using binary decision diagrams of Section 7.9.1 can be appreciated. This subject is of pervasive importance in synthesis, testing, and verification [61, 63].

We have demonstrated by example why (and how) variable ordering is important. In Section 6.4 we discussed the state of the art in variable ordering techniques. We also discussed design considerations for BDD packages, and how to obtain efficient and recently developed BDD packages.

6.6 Problems

1. Find the reduced BDD without complement edges for the BDD of Figure 6.23. Repeat the problem, drawing this time a reduced BDD with complement edges. Finally, write a sum-of-product expression for \( f \).

Solution. The two BDDs are in Figure 6.24. A sum-of-product expression for \( f \) can be derived from the BDDs:

\[
f = abd' + ab'd + ac + a'c'd.
\]

2. Write a sum-of-product expression for the function \( f \) represented by the BDD with complement edges of Figure 6.25.
result are computed serially does matter. This is taken into account in Figure 6.21 by not expanding \texttt{ite.constant}(q', 0, 1) until the result of the other branch is known. The computation of \texttt{ite.constant}(q', 0, 1) itself is a special case, since q' is not constant and therefore q' \cdot 0 + q \cdot 1 = g is not constant either.

The correctness of the result can be verified by observing that for a'\cdot b'd', f = 1 and g = 0. The procedure, however, does not even need to find such an example.

We conclude with a more meaningful example, which shows how to deal with complement edges in \texttt{ite} and \texttt{ite.constant}. Consider the functions

\[
\begin{align*}
f &= ab + c, \\
g &= bd' + d, \\
h &= e + d',
\end{align*}
\]

with the variable ordering

\[a \leq b \leq c \leq d.\]

Now suppose that we are given the shared BDD (with complement edges) for \(f, g, h\), and need to compute the BDD for \texttt{ite}(f, g, h). The results, shown in Figure 6.22, are obtained as follows.

\[
\begin{align*}
\texttt{ite}(f, g, h) &= (a, \texttt{ite}(p, g, h), \texttt{ite}(c, g, h)) \\
&= (a, (b, \texttt{ite}(1, q, h), \texttt{ite}(c, d, h)), (b, \texttt{ite}(c, q, h), \texttt{ite}(c, d, h))) \\
&= (a, (b, q, \texttt{ite}(c, d, h)), (b, \texttt{ite}(c, q, h), \texttt{ite}(c, d, h))) \\
&= (a, (b, q, \texttt{ite}(c, d', h)), (c, d', d')) \\
&= (a, (b, q, \texttt{ite}(c, d', h)), (c, d', d')).
\end{align*}
\]

\textsuperscript{11}Specifically, it matters for the number of recursive calls. It does not affect the final result.

6.4 Notes

There has been a surprising steady influx of "new" types of DDDs (Decision Diagrams). The work of Bryant in [47] focused on Boolean functions and decision diagrams that were based on the Boolean expansion of Section 3.3.3 and specific types of reduction to canonical form. However, some new forms of DDDs focus on Pseudo-Boolean Functions have been introduced: [67] (called MTBDDs), [11, 12, 13, 132] (called ADDs), [165] (called EVBDDs), [46] (called BDMs).

These works have successfully extended the applicability of DDDs into the realm of functions with Boolean domains but integer or real valued ranges. This has opened up new vistas for research on Markov Analysis and Probabilistic Verification, Shortest Path Analysis on graphs of vast size, "Exact" (that is free of false paths) timing analysis of combinational circuits, low power synthesis, technology mapping, etc.

Other work on extending the applicability of DDDs has focused on the reduction to canonicity. Different reduction or ordering rules have been studied: [209] (Zero-Suppressed BDDs), [23] (Free BDDs), [91] (Functional DDDs), [218] (Extended Decision Diagrams). Some or all of these variants can be mixed into a single "Hybrid Decision Diagram". Zhao and Clarke of CMU have developed this DD in a package for word level verification of sequential arithmetic circuits which has won internal awards at INTEL.

A serious issue with DDDs is ordering. In every type of DDD, the variables in the support of the function being represented must be ordered to obtain canonicity. Unfortunately, in many cases, the size of the BDD depends critically on the specific ordering chosen. This dependence is so drastic that in many cases it is impractical to build the BDD at all. For example it is known that BDD size grows exponentially with variable count, independent of the order. This limits the use of BDDs for multipliers to 16 bits or less.

An exact algorithm for BDD variable ordering is the improved version by Lahura et al. [150] of Friedman and Supowit's [104]. Various iterative techniques are discussed in [106, 150, 237, 207, 206]. Heuristic techniques to find a good order for a circuit are discussed in [105, 183, 201, 262]. A comparison of the performance of various methods can be found in [154]. Partial BDDs have been studied in [230, 53]. Other references on variable ordering are: [209, 102, 48, 26, 90].

At the time of writing, several BDD packages of high quality were in use. Radek's package was proprietary to Synopsys. David Long's package, developed at CMU, is employed in SMV (CMU verification package), VIS (UC Berkeley Verification Package) and SIS (UC Berkeley Synthesis Package) as well as AT&T and other industrial sites. A more recent entry into the BDD package field is the CUDD package from the University of Colorado at Boulder. This package is used in VIS and in
Motorola's VERDICT verification package as well as at several university sites. Of all of these, the CUDP package has the most efficient dynamic reordering package, and goes the farthest in supporting diverse BDD types (it currently supports BDDs, ADDs/MTBDDs, and ZDDs (zero-suppressed BDDs)).

The CUDP package is available on the web via anonymous FTP at

\[ \text{vlsi.colorado.edu} \]

(login as anonymous, give your email address as password, and then type "cd pub").

6.5 Summary

In this chapter, we have introduced BDDs and summarized briefly the enormous impact they have had on the merging fields of synthesis and verification. We have shown how to build and manipulate them (see the solved problems for applying the usual Boolean operators to two functions represented by BDDs). We have shown how an ordered BDD is made canonical by imposing certain reduction rules.

We have studied the qualitative operation of the \texttt{ite} and \texttt{ite:const} algorithms, and show how many of the BDD manipulations that might be needed in practice can be done efficiently, given efficient implementation of these algorithms.

We have avoided the intricacies of memory management, but they are all-important to the robustness of a BDD package\textsuperscript{12}

We have given enough background so that the discussion of FSM traversal using binary decision diagrams of Section 7.9.1 can be appreciated. This subject is of pervasive importance in synthesis, testing, and verification [61, 63].

We have demonstrated by example why (and how) variable ordering is important. In Section 6.4 we discussed the state of the art in variable ordering techniques. We also discussed design considerations for BDD packages, and how to obtain efficient and recently developed BDD packages.

6.6 Problems

1. Find the reduced BDD without complement edges for the BDD of Figure 6.23. Repeat the problem, drawing this time a reduced BDD with complement edges. Finally, write a sum-of-product expression for $f$.

Solution. The two BDDs are in Figure 6.24. A sum-of-product expression for $f$ can be derived from the BDDs:

$$ f = ab'd' + ab'd + a'c + a'c'd. $$

2. Write a sum-of-product expression for the function $f$ represented by the BDD with complement edges of Figure 6.25.

\textsuperscript{12}Detailed notes on this subject are available from the authors.
3. Find a good variable ordering for

\[ f = (a + be)(d + c). \]

Discuss what was your reasoning. Draw the BDD for your ordering using complement edges.

**Solution.** Since \( f \) is the product of two functions with disjoint support, we keep the variables in the two supports separate in the order. We then notice that \( b \) and \( c \) on the one hand, and \( d \) and \( c \) on the other hand, are symmetric. Hence, their relative order is immaterial. Finally, we can apply the same argument we applied to \( f \) to \( a + be \) so that \( a \) and \( b, e \) are not interleaved. These considerations leave several orders possible. One is simply obtained by listing the variables in the order in which they appear in the given expression for \( f \):

\[ a < b < c < d < e. \]

The resulting BDD is shown in Figure 6.26. The BDD is clearly optimal because each variable labels exactly one node.

4. For the following functions:

\[ f = a + b'c \]
\[ g = b'c + d \]
\[ h = b + c' + d \]

(a) Draw BDDs \( F, G, \) and \( H \) with complement edges using the variable ordering

\[ a \leq b \leq c \leq d; \]

(b) Compute \( \text{ite}(F,G,H); \)

(c) Draw the corresponding BDD with complement edges;

(d) Compute \( \text{ite}\_\text{constant}(F,G,H). \)

Note that you have to figure out the details of how to deal with complement edges in the \( \text{ite} \) algorithm.

**Solution.** The BDD for the operands and the result is shown in Figure 6.27. The computation of \( \text{ite}(F,G,H) \) proceeds as follows.

\[
I = (a, \text{ITE}(1,G,H), \text{ITE}(p',G,H)) \\
= (a, G, (b, \text{ITE}(0,d,1), \text{ITE}(c,q,r))) \\
= (a, G, (b, 1, (c, \text{ITE}(1,1,d), \text{ITE}(0,d,1)))) \\
= (a, G, (b, 1, (c, 1, 1))) \\
= (a, G, (b, 1, 1)) \\
= (a, G, 1)
\]

The computation of \( \text{ITE}\_\text{constant}(F,G,H) \) proceeds as follows.

\[
I = (a, \text{ITE}\_\text{constant}(1,G,H), \text{ITE}\_\text{constant}(p',G,H)) \\
= (a, \text{non}\_\text{constant}, \text{ITE}\_\text{constant}(p',G,H)) \\
= \text{non}\_\text{constant}
\]

This happens because the result from the positive branch is known to be \( G \), and \( G \) is known not to be constant.
5. Write pseudo code for the APPLY operation, defined here as a procedure that will take two BDDs and a Boolean connective operator as arguments. Assume the operation is specified by passing the name of a function that actually computes the desired operation. Write the function needed to compute the OR of two BDDs.

Solution. The pseudo code for APPLY is shown in Figure 6.28. The pseudo code for the OR of two BDDs is shown in Figure 6.29. Notice that we could return 1 if \( F \geq G' \). However, this test would not be possible in constant time.

6. For the following functions:

\[
\begin{align*}
f &= b + cd \\
g &= b + cd \\
h &= a'c + b
\end{align*}
\]

(a) Draw BDDs \( F, G, \) and \( H \) with complement edges using the variable ordering

\[ a \leq b \leq c \leq d; \]

(b) Compute \( \text{ite}(F, G, H) \) by putting the triple in standard form first;
(c) Draw the resulting BDD with complement edges.

Solution. The BDD for the operands and the result is shown in Figure 6.30. The computation of \( \text{ite}(F, G, H) \) proceeds as follows.

\[
\begin{align*}
l &= \text{ite}(F, G, H) \\
   &= \text{ite}(F, 1, H) \\
   &= \text{ite}(H, 1, F)
\end{align*}
\]
Chapter 6. Binary Decision Diagrams (BDDs)

\[ = (a, \text{ITE}(b, 1, F), \text{ITE}(g, 1, F)) \]
\[ = (a, (b, \text{ITE}(1, 1, 1), \text{ITE}(0, 1, p)), (b, \text{ITE}(1, 1, 1), \text{ITE}(c, 1, p))) \]
\[ = (a, (b, 1, p), (b, 1, (c, \text{ITE}(1, 1, d), \text{ITE}(0, 1, 0)))) \]
\[ = (a, F, (b, 1, 0))) \]
\[ = (a, F, (b, 1, c)) \]
\[ = (a, F, g) \]

Note that in this case the sharing between the operands and the result is detected by looking at the unique table.

Part III

Models of Sequential Systems