Chapter 2

PRELIMINARIES

In this chapter, some definitions and basic notations of Boolean algebra and reduced ordered *Binary Decision Diagrams* (BDDs) are given, as far as they are used in the following chapters. First, Boolean functions are defined and an important method for their decomposition is explained. Then a formal definition of BDDs is given.

2.1 Notation

This section explains general notations used throughout this book. Often sets and power sets are considered. The notation for the power set of a given set M is

$$2^M = \{S | S \subseteq M\}.$$

$\mathbb{N}$ denotes the set of natural numbers *not* including zero, i.e.

$$\mathbb{N} = \{1, 2, \ldots\}.$$

Variables which are assumed to have values in $\mathbb{N}$ are most of the time denoted by the letters i, j, k, m and n. Then, if the range of these variables is given by the context, sometimes a specification like "$n \in \mathbb{N}$" is omitted for simplicity.

When giving a result which expresses a both-way implication "if and only if" between a left and a right side of the statement, often the notation "iff" will be used as abbreviation, e.g.

$$a = \sqrt{b} \text{ iff } a^2 = b.$$

Functions usually are denoted using the identifiers f, g, and h. A function f is given as a mapping from a domain X -domain Y. Domain and co-domain are stated in advance, e.g. $f\colon X \to Y$. The mapping then is defined with an expression of the form

$$f(x) = \text{an expression using } x$$

or

$$(x_1, x_2, \ldots, x_k) \mapsto \text{an expression using } x_1, x_2, \ldots, x_k$$

(e.g., in the case of a function $f\colon \mathbb{N}^2 \to \mathbb{N}$, the function may be given in the form $(x_1, x_2) \mapsto x_1 \cdot x_2$).

However, in the case of captions of figures etc. a shorter notation may be used to save space: the function above might be given in the short form

$$f = x_1 \cdot x_2.$$

This is done only if domain and co-domain are clear or given by the context (e.g. if f above is already known to be a Boolean function, it is clear from the short notation that $f\colon \{0,1\}^2 \to \{0,1\}$).

2.2 Boolean Functions

Let $\mathbf{B}$:=$\{0, 1\}$ and $n \in \mathbb{N}$. Boolean variables, typically denoted by Latin letters, e.g. x, y, z are bound to values in $\mathbf{B}$. Variables are referred to by subscripts which are from the set $\{1, 2, \ldots, n\}$, e.g.

$$x_1, x_2, \ldots, x_n.$$

To denote the set $\{x_1, x_2, \ldots, x_n\}$ of "standard" variables we use the notation X_n. Later, in Chapter 4, also the notation $X_j^i = \{x_i, x_{i+1}, \ldots, x_{j-1}, x_j\}$ will be used to refer to several subsets of X_n. The following is an introduction of notations defining Boolean functions.

Definition 2.1 *Let $m, n \in \mathbb{N}$. A mapping*

$$f\colon \mathbf{B}^n \to \mathbf{B}^m$$

is called a Boolean function*. In the case of $m = 1$ we say f is a* single-output function*, otherwise f is called a* multi-output function*.*

These terms are used because Boolean functions are used to describe the digital logic of a *circuit*. A circuit transforms *inputs*, i.e. a vector of incoming Boolean signals to a vector of *outputs*, thereby following a certain logic. This logic can be described by a Boolean function.

Other properties of a circuit (e.g. critical path delay or area requirement) need a more sophisticated representation (e.g. as BDD which is a special form of a graph). Let $f\colon \mathbf{B}^n \to \mathbf{B}^m$ be a Boolean function. To put emphasis on the arity n of f, we may choose to write $f^{(n)}$ instead of f. This notation will be used for functions in general, e.g. later on we use it to denote variable orderings $\pi^{(n)}$ (see Section 2.4). Sometimes we may even write $f^{(n,m)}$ to reflect the whole signature of a (Boolean) function f.

A multi-output function f: $\mathbf{B}^n \to \mathbf{B}^m$ can be interpreted as a family of m single-output functions $(f_i^{(n)})_{1 \leq i \leq m}$. The f_i's are called *component functions.*

To achieve a standard, in this book the set of variables of a Boolean function $f^{(n)}$ will always be assumed to be X_n. If not stated otherwise, Boolean functions are assumed to be *total* (*completely specified*), i.e. there exists a defined function value for every vector of input variables. The Boolean functions constantly mapping every variable to 1 (to 0) are denoted one (zero), i.e.

$$\begin{aligned} \text{one}: \quad & \mathbf{B}^n \to \mathbf{B}^m; \quad (x_1, x_2, \ldots, x_n) \mapsto 1, \\ \text{zero}: \quad & \mathbf{B}^n \to \mathbf{B}^m; \quad (x_1, x_2, \ldots, x_n) \mapsto 0. \end{aligned}$$

A Boolean variable x_i itself can be interpreted as a first example of a Boolean function

$$x_i\colon \mathbf{B}^n \to \mathbf{B}; \ (a_1, a_2, \ldots, a_i, \ldots, a_n) \mapsto a_i.$$

This function is called the *projection function* for the i-th component.

DEFINITION 2.2 *The* complement *of a Boolean variable x_i is given by the mapping*

$$\overline{x}_i\colon \ \mathbf{B}^n \to \mathbf{B}; \ (a_1, a_2, \ldots, a_i, \ldots, a_n) \mapsto \overline{a}_i$$

where $\overline{a}_i = 1$ iff $a_i = 0$.

An interesting class of Boolean functions are (partially) symmetric functions. Later, in Chapter 3, algorithms for BDD minimization will be presented which exploit (partial) symmetry to reduce run time.

DEFINITION 2.3 *Let $f\colon \mathbf{B}^n \to \mathbf{B}^m$ be a multi-output function. Two variables x_i and x_j are called* symmetric*, iff*

$$\begin{aligned} & f(x_1, \ldots, x_i, \ldots, x_j, \ldots, x_n) \\ & \quad = \ f(x_1, \ldots, x_{i-1}, x_j, x_{i+1}, \ldots, x_{j-1}, x_i, x_{j+1}, \ldots, x_n). \end{aligned}$$

Symmetry is an equivalence relation which partitions the set X_n into disjoint classes $S_1, \ldots, S_k$ called the *symmetry sets.* A function f is called *partially symmetric*, iff it has at least one symmetry set S with $|S| > 1$. If a function f has only one symmetry set set $S = X_n$, then it is called *totally symmetric.*

2.3 Decomposition of Boolean Functions

DEFINITION 2.4 *Let f: $\mathbf{B}^n \to \mathbf{B}^m$ be a Boolean function. The* cofactor *of f for $x_i = c$ ($c \in \mathbf{B}$) is the function $f_{x_i=c}$: $\mathbf{B}^n \to \mathbf{B}^m$. For all variables in X_n it is defined as*

$$\begin{aligned} & f_{x_i=c}(x_1, x_2, \ldots, x_{i-1}, x_i, x_{i+1}, \ldots, x_n) \\ = \ & f(x_1, x_2, \ldots, x_{i-1}, c, x_{i+1}, \ldots, x_n). \end{aligned}$$

A cofactor of f is the function derived from f by fixing a variable of f to a value in $\mathbf{B}$. Formally, a cofactor of $f^{(n)}$ has the same arity n. In contrast to all variables different from x_i, the variable x_i is not free in the cofactor $f_{x_i=c}$. Hence the cofactor does not depend on this variable (see Definition 2.5).

Despite the generality of the last definition covering multi-output functions, sometimes only the cofactors of single-output functions f: $\mathbf{B}^n \to \mathbf{B}$ are of interest. When a multi-output function $f^{(n,m)} = (f_i^{(n)})_{1 \le i \le m}$ is given, we often consider the cofactors of the component functions f_i only. These cofactors then are single-output functions of arity n. A cofactor of a multi-output function f can be interpreted as a family of cofactors of the component functions of f.

A cofactor $f_{x_i=c}$ is sometimes called a *direct* cofactor to emphasize that there is only one variable bound to a value in $\mathbf{B}$. This opposes to a cofactor in more than one variable. E.g., for $k \le n$, $x_{i_1}, \ldots, x_{i_k} \in X_n$ and $c_1, \ldots, c_k \in \mathbf{B}$, the function $f_{x_{i_1}=c_1, x_{i_2}=c_2, \ldots, x_{i_k}=c_k}$ is a cofactor in multiple variables. This cofactor is equivalent to several direct cofactors, e.g. to

$$(f_{x_{i_1}=c_1, x_{i_2}=c_2, \ldots, x_{i_{k-1}}=c_{k-1}})_{x_{i_k}=c_k}.$$

In general it is equivalent to

$$(f_{x_{i_1}=c_1, x_{i_2}=c_2, \ldots, x_{i_{j-1}}=c_{j-1}, x_{i_{j+1}}=c_{j+1}, \ldots, x_{i_k}=c_k})_{x_{i_j}=c_j}$$

for any $1 \le j \le k$. A cofactor in multiple variables is uniquely determined regardless of the order in which we fix these variables. Hence, these cofactors can also be thought of being obtained by simultaneously fixing all the involved variables. To obtain increased readability, sometimes a "|" sign is used to separate the function symbol from the list of variable bindings, e.g. we write $f_j|_{x_{i_1}=c_1}$ for a cofactor in a component function f_j.

DEFINITION 2.5 *Let f: $\mathbf{B}^n \to \mathbf{B}^m$ be a Boolean function and let $x_i \in X_n$. Then function f is said to* essentially depend *on x_i iff*

$$f_{x_i=0} \neq f_{x_i=1}.$$

The set of variables which f essentially depends on, is called the support *of f and is denoted* $\mathrm{support}_{n,m}(f)$*. Usually n, m are clear from the context and hence we simply write* $\mathrm{support}(f)$*.*

Formally, $\mathrm{support}_{n,m}$ is a mapping $\{f \mid f\colon \mathbf{B}^n \to \mathbf{B}^m\} \to 2^{X_n}$. If not stated otherwise, a given Boolean function $f\colon \mathbf{B}^n \to \mathbf{B}^m$ is always defined over the variable set X_n and always $\mathrm{support}(f) = X_n$ is assumed. The next definition characterizes cofactors of a Boolean function which are derived by fixing at least one variable of the support of the function.

Definition 2.6 *Let $f\colon \mathbf{B}^n \to \mathbf{B}^m$ be a Boolean (single or multi-output) function over X_n. A cofactor $f_{x_{i_1}=a_1, x_{i_2}=a_2, \ldots, x_{i_k}=a_k}$ with $x_{i_1}, x_{i_2}, \ldots, x_{i_k} \in X_n$, $(a_1, a_2, \ldots, a_k) \in \mathbf{B}^k$ is called* true *iff*

$$\{x_{i_1}, x_{i_2}, \ldots, x_{i_k}\} \cap \mathrm{support}(f) \neq \emptyset.$$

Note that true cofactors of a function cannot be equivalent to the function itself.

Often it is more useful to consider a set of cofactors of a Boolean function rather than considering just one particular cofactor. This is reflected by the next definition.

Definition 2.7 *Let $f\colon \mathbf{B}^n \to \mathbf{B}^m$; $f = (f_i^{(n)})_{1 \leq i \leq m}$ be a Boolean multi-output function essentially depending on all its input variables and let $I \subseteq X_n$. The set of non-constant cofactors of f with respect to the variables in I is denoted* $\mathrm{cof}_{n,m}(f, I)$*. Formally, a function* $\mathrm{cof}_{n,m}$ *is given as*

$$\mathrm{cof}_{n,m}\colon \{f \mid f\colon \mathbf{B}^n \to \mathbf{B}^m\} \times 2^{X_n} \to 2^{\{f \mid f\colon \mathbf{B}^n \to \mathbf{B}^m\}};$$

$$(f, \{x_{i_1}, \ldots, x_{i_k}\}) \mapsto$$
$$\{f_i|_{x_{i_1}=a_1, \ldots, x_{i_k}=a_k} \text{ non-constant } \mid 1 \leq i \leq m, (a_1, \ldots, a_k) \in \mathbf{B}^k\}$$

for $1 \leq k \leq n$.

The set $\mathrm{cof}_{n,m}(f, I)$ is the set of all distinct (non-constant and single-output) cofactors of f (f is interpreted as a family of m n-ary single-output functions) with respect to all variables in I. Note that this is not a multiset, hence functionally equivalent cofactors are eliminated and thus do not contribute to $|\mathrm{cof}_{n,m}(f, I)|$.

If n and m are clear from the context, we simply write $\mathrm{cof}(f, I)$.

Next, we restrict this set to contain only cofactors that are *true* cofactors of one of the single-output functions f_i.

Definition 2.8 *Let* $f\colon \mathbf{B}^n \to \mathbf{B}^m$*;* $f = (f_i^{(n)})_{1\le i\le m}$ *be a Boolean multi-output function essentially depending on all its input variables and let* $I \subseteq X_n$*. The set of non-constant true cofactors of* f *with respect to the variables in* I *is denoted* $\mathrm{tcof}_{n,m}(f, I)$*. Formally, a function* $\mathrm{tcof}_{n,m}$ *is given as*

$$\mathrm{tcof}_{n,m}\colon\ \{f \mid f\colon\ \mathbf{B}^n \to \mathbf{B}^m\} \times 2^{X_n} \to 2^{\{f\mid f\colon\ \mathbf{B}^n\to\mathbf{B}^m\}};$$

$$\begin{array}{l}(f, \{x_{i_1}, \ldots, x_{i_k}\}) \mapsto \\ \quad \{\, f_i|_{x_{i_1}=a_1,\ldots,x_{i_k}=a_k} \ \textit{non-constant and true cofactor of } f_i \mid \\ \quad\quad 1 \le i \le m, (a_1, \ldots, a_k) \in \mathbf{B}^k \,\}\end{array}$$

for $1 \le k \le n$*.*

Let $f_i \neq f_j$ be two distinct single-output functions in the family $(f_i^{(n)})_{1\le i\le m}$. Note that a true cofactor in f_i can be functionally equivalent to a cofactor of f_j that is not true. In other words, the cofactors in the set $\mathrm{tcof}_{n,m}(f, I)$ are not required to be true in *every* single-output function, it is only required that at least one such single-output function exists.

Again if n and m are given from the context, we simply write $\mathrm{tcof}(f, I)$.

The following well-known theorem [Sha38] allows to decompose Boolean functions into "simpler" sub-functions.

Theorem 2.9 *Let* $f\colon\ \mathbf{B}^n \to \mathbf{B}^m$ *be a Boolean function (over* X_n*). For all* $x_i \in X_n$ *we have:*

$$f = x_i \cdot f_{x_i=1} + \overline{x}_i \cdot f_{x_i=0}. \tag{2.1}$$

It is straightforward to see that the sub-functions obtained by subsequent application of Theorem 2.9, called the *Shannon decomposition*, to a function f are uniquely determined. Note that this ensures the well-definedness of cofactor set definitions.

2.4 Reduced Ordered Binary Decision Diagrams

Many applications in VLSI CAD make use of reduced ordered *Binary Decision Diagrams* (BDDs) as introduced by [Bry86]:

A BDD is a *graph-based* data structure. Redundant nodes in the graph, i.e. nodes not needed to represent f, can be eliminated. BDDs allow a unique (i.e. canonical) representation of Boolean functions. At the same time they allow for a good trade-off between efficiency of manipulation and compactness. Compared to other techniques to represent Boolean functions, e.g. truth tables or Karnaugh maps, BDDs often require much less memory and faster algorithms for their manipulation do exist.

In the following, a formal definition of BDDs is given. We start with purely *syntactical* definitions by means of *Directed Acyclic Graphs* (DAGs). First, single-rooted *Ordered Binary Decision Diagrams* (OBDDs) are defined. This definition is extended to multi-rooted graphs, yielding *Shared* OBDDs (SBDDs). Next, the *semantics* of SBDDs is defined, clarifying how Boolean functions are represented by SBDDs.

After that, reduction operations on SBDDs are introduced which preserve the semantics of an SBDD. This leads to the final definition of reduced SBDDs that will be called BDDs for short in this book.

Finally, some definitions and notations are given which allow to discuss various graph-oriented properties of BDDs, among them paths in BDDs and their (expected) length.

Examples are given to illustrate the formal definitions where appropriate.

2.4.1 Syntactical Definition of BDDs

DEFINITION 2.10 *An* Ordered Binary Decision Diagram *(OBDD) is a pair (π, G) where π denotes the variable ordering of the OBDD and G is a finite DAG $G = (V, E)$ (V denotes the set of* vertices *and E denotes the set of* edges *of the DAG) with exactly one root node (denoted root) and the following properties:*

- *A node in V is either a non-terminal node or one of the two terminal nodes in $\{\mathbf{1}, \mathbf{0}\}$.*

- *Each non-terminal node v is labeled with a variable in X_n, denoted* $\mathrm{var}(v)$*, and has exactly two child nodes in V which are denoted* $\mathrm{then}(v)$ *and* $\mathrm{else}(v)$.

- *On each path from the root node to a terminal node the variables are encountered at most once and in the same order.*

More precisely, the variable ordering π of an OBDD is a bijection

$$\pi\colon \{1, 2, \ldots, n\} \to X_n$$

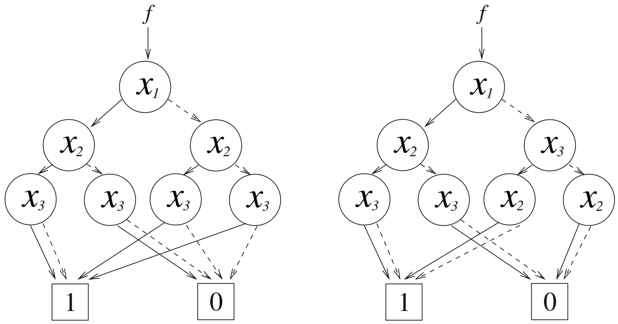

Figure 2.1. Representation of $f = x_1 \cdot x_2 + \overline{x}_1 \cdot x_3$ by an OBDD and an unordered BDD.

where $\pi(i)$ *denotes the i-th variable in the ordering. The above condition "in the same order" states that for any non-terminal node* v *we have*

$$\pi^{-1}(\mathrm{var}(v)) < \pi^{-1}(\mathrm{var}(\mathrm{then}(v)))$$

iff $\mathrm{then}(v)$ *is also a non-terminal node and*

$$\pi^{-1}(\mathrm{var}(v)) \quad < \quad \pi^{-1}(\mathrm{var}(\mathrm{else}(v)))$$

iff $\mathrm{else}(v)$ *is also a non-terminal node.*

Even though this might look a bit "over-formal", the notation is required in the following to prove the correctness of the algorithms.

EXAMPLE 2.11 *In Figure 2.1 two different types of binary decision diagrams are depicted. Solid lines are used for the edges from* v *to* $\mathrm{then}(v)$ *whereas dashed lines indicate an edge between* v *and* $\mathrm{else}(v)$*. In both diagrams the variables in* $\{x_1, x_2, x_3\}$ *are encountered at most once on every path. Whereas the right diagram is not ordered since the variables are differently ordered along distinct paths, the left one respects the ordering* $\pi(1) = x_1, \pi(2) = x_2, \pi(3) = x_3$*. Both diagrams represent the function* $f\colon \mathbf{B}^3 \to \mathbf{B}$*;* $(x_1, x_2, x_3) \mapsto x_1 \cdot x_2 + \overline{x}_1 \cdot x_3$*, as will be explained in a later section.*

Where appropriate, we will speak of an OBDD over X_n to put emphasis on the set of variables used as node labels. However, if not stated otherwise, an OBDD is always assumed to be an OBDD over X_n.

Note that OBDDs are *connected* graphs, as all nodes must be connected via at least one path to the (only) root node: to see this, assume the graph consists of more than one connected component. But then, due to the graph being finite and acyclic, there must exist a second root node for the second component graph. This contradicts the assumption that the graph is single-rooted. Let root $\in V$ denote the one root node of the OBDD.

Formally, a *function* var: $V \setminus \{\mathbf{1}, \mathbf{0}\} \rightarrow X_n$ maps non-terminal nodes to variables and *functions* then: $V \setminus \{\mathbf{1}, \mathbf{0}\} \rightarrow V \setminus \{\text{root}\}$ and else: $V \setminus \{\mathbf{1}, \mathbf{0}\} \rightarrow V \setminus \{\text{root}\}$ map non-terminal nodes to child nodes. Both functions are well-defined and total. If not stated otherwise, we will consider surjective functions var, i.e. every variable in X_n appears as label in the OBDD. This corresponds to the convention to consider Boolean functions which essentially depend on every variable in X_n. In general none of these functions is required to be injective, i.e. several nodes can have the same label and share the same child nodes. Let $G = (V, E)$ be the underlying graph of an OBDD $(\ldots, G)$ and let $v \in V \setminus \{\mathbf{1}, \mathbf{0}\}$. We call then$(v)$ the 1-child and else(v) the 0-child of v.

For an edge $e \in E$ $(E \subseteq V \times V)$, we denote the *type* of the edge with $\mathrm{t}(e)$, i.e. we have $\mathrm{t}(e) = 1$ for an edge $e = (v, \mathrm{then}(v))$ (called a 1-edge) and $\mathrm{t}(e) = 0$ for an edge $e = (v, \mathrm{else}(v))$ (called a 0-edge).

Definition 2.12 *A* Shared *OBDD (SBDD) is a tuple* (π, G, O). *G is a rooted, possibly multi-rooted DAG* (V, E) *which consists of a finite number of graph components. These components are OBDDs, all of them respecting the same variable ordering* π. $O \subseteq V \setminus \{\mathbf{1}, \mathbf{0}\}$ *is a finite set of output nodes* $O = \{o_1, o_2, \ldots, o_m\}$. *An SBDD has the following properties:*

- *A node in V is either a non-terminal node or one of the two terminal nodes in* $\{\mathbf{1}, \mathbf{0}\}$.
- *Every root node of the component OBDD graphs must be contained in O (but not necessarily vice versa).*

Example 2.13 *An example of an SBDD is given in Figure 2.2. The nodes pointed to by f_1, f_2 and f_3 are output nodes. Note that every root node is an output node (pointed to by f_2 and f_3). An output node must not necessarily be a root node (see the node pointed to by f_1).*

Also note that in SBDDs multiple graphs can share the same node, a property which helps to save nodes and to reduce the size of the diagram. In contrast to the OBDD in Figure 2.1, an SBDD represents multiple Boolean functions in only one diagram, as will be explained later.

Where appropriate, again we speak of an SBDD over X_n to clarify the set of node labels.

If there is only one component in G and if O is a singleton (consisting of the one root node of G), an SBDD specializes to an OBDD. Later, in

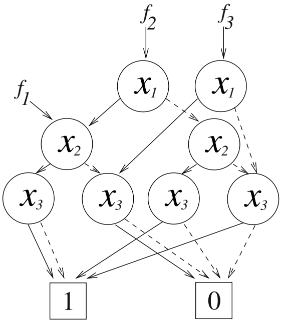

Figure 2.2. A shared OBDD (SBDD).

the case that for some reason single-rooted diagrams must explicitly be excluded, or to put emphasis on the multi-rooted case, the term "shared BDD" will be used where a BDD is a "reduced" SBDD (see Definition 2.16 and also Remark 1). Since SBDDs are not necessarily reduced, the terms "SBDD" and "shared BDD" should not be mixed up.

The idea behind the set O is to declare additional non-terminal, non-root nodes as nodes representing Boolean functions. This will be clarified in the next section when the semantics of BDDs is defined. Note that also SBDDs have at most two terminal nodes which are shared by the components.

2.4.2 Semantical Definition of BDDs

DEFINITION 2.14 *An SBDD* $(\dots, G, O)$*, over* X_n *with* $O = \{o_1, o_2, \dots, o_m\}$ *represents the multi-output function* $f := (f_i^{(n)})_{1 \le i \le m}$ *defined as follows:*

- *If* v *is the terminal node* **1***, then* $f_v = \text{one}$*, if* v *is the terminal node* **0***, then* $f_v = \text{zero}$*.*
- *If* v *is a non-terminal node and* $\text{var}(v) = x_i$*, then* f_v *is the function*

$$\begin{aligned} & f_v(x_1, x_2, \dots, x_n) \\ & \quad = \quad x_i \cdot f_{\text{then}(v)}(x_1, x_2, \dots, x_n) + \overline{x}_i \cdot f_{\text{else}(v)}(x_1, x_2, \dots, x_n). \end{aligned}$$

- *For* $1 \le i \le m$*,* f_i *is the function represented by the node* o_i*.*

The expression $f_{\text{then}(v)}$ ($f_{\text{else}(v)}$) denotes the function represented by the child nodes then(v) (else(v)). At each node of the SBDD, essentially a Shannon decomposition (see Theorem 2.9) is performed. In this, an SBDD recursively splits a function into simpler sub-functions. The first

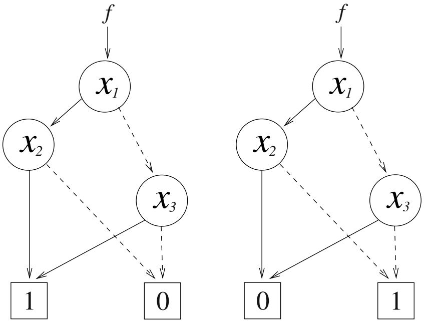

Figure 2.3. Representation of $f = x_1 \cdot x_2 + \overline{x}_1 \cdot x_3$ and $\overline{f} = x_1 \cdot \overline{x}_2 + \overline{x}_1 \cdot \overline{x}_3$.

definitions of binary decision diagrams are due to [Lee59] and [Ake78a]. *Complemented Edges* (CEs) are an important extension of the basic BDD concept and have been described in [Ake78b, Kar88, MB88, MIY90, BRB90]. The idea is to use the close similarity between the BDD representing a function f and the BDD representing its complement. For example see Figure 2.3 where a left BDD for $f = x_1 \cdot x_2 + \overline{x}_1 \cdot x_3$ and a right BDD for $\overline{f} = x_1 \cdot \overline{x}_2 + \overline{x}_1 \cdot \overline{x}_3$ is given. The diagrams are identical except for interchanged terminal nodes.

A CE is an ordinary edge that is tagged with an extra bit (*complement bit*): This bit is set to indicate that the connected sub-graph must be interpreted as the complement of the formula that the sub-graph represents. CEs allow to represent both a function and its complement by the same node, modifying the edge pointing to that node instead. As a consequence, only one constant node is needed. Usually the node **1** is kept, allowing the function zero to be represented by a CE to **1**.

Note that it is not necessary to store the complement bit as an extra element of the node structure. Instead, a smart implementation technique can be used which exploits the memory alignment used in present computer systems: modern CPUs require allocated objects to reside at a memory address that is evenly divisible by some small constant, e.g. often this is the constant two or four. Consequently, the least significant bit of a pointer word is irrelevant for address calculation. Therefore it can be used to store the complement bit (provided that an appropriate bit mask is applied before address calculation). Hence using CEs does not cause any memory overhead. SBDDs with CEs are used today in many modern BDD packages.

For the sake of a more understandable presentation of the achieved results, in this book “classical”, unmodified SBDDs without CEs are used in most of the examples and their illustrations (with the exception

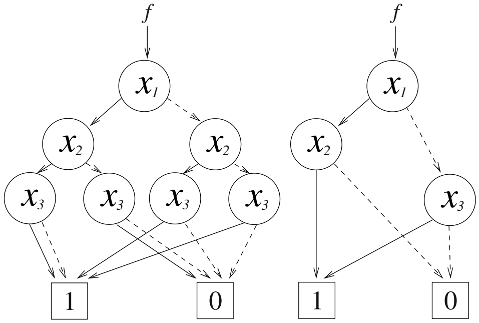

Figure 2.4. Two different SBDDs for f: $(x_1, x_2, x_3) \mapsto x_1 \cdot x_2 + \overline{x}_1 \cdot x_3$.

of Section 5.1 where CEs are discussed explicitly). However, all results presented throughout the book hold for BDDs with CEs and can easily be transferred to BDDs without CEs.

Note that in case of using CEs special attention must be drawn to maintain a so-called "canonical form", i.e. to achieve uniqueness of the representation of Boolean functions. This will be addressed again in Section 2.4.3 where the unique, irreducible form of BDDs is introduced.

2.4.3 Reduction Operations on SBDDs

In the previous sections, we developed syntax and semantics of a special form of graphs which are representing Boolean functions. However, even if the considered variable ordering is fixed, still there exist several possibilities of representing a given function: in Figure 2.4 we see two different SBDDs respecting the same variable ordering $\pi^{-1}(x_1) < \pi^{-1}(x_2) < \pi^{-1}(x_3)$, representing the same function f: $\mathbf{B}^3 \rightarrow \mathbf{B}$; $(x_1, x_2, x_3) \mapsto x_1 \cdot x_2 + \overline{x}_1 \cdot x_3$. Next, we will give reduction operations on SBDDs which transform an SBDD into an irreducible form, while the function represented by the SBDD is preserved. This is a crucial technique leading to an SBDD respecting a given variable ordering which is unique up to graph isomorphism and of minimal size.

A *reduced* SBDD then is the final form of binary decision diagrams that will be considered throughout this book, called BDD.

DEFINITION 2.15 *Given an SBDD $(\ldots, G, O)$, there are the following two* reduction rules *which can be applied to $G = (V, E)$:*

- **Deletion Rule:** *Let $v \in V$ be a node with*

$$\text{then}(v) = \text{else}(v) =: v'.$$

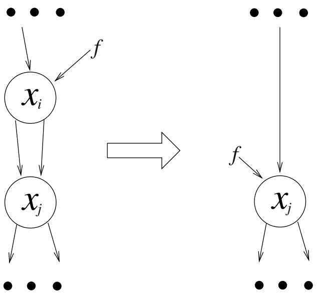

Figure 2.5. Deletion Rule for SBDD-reduction.

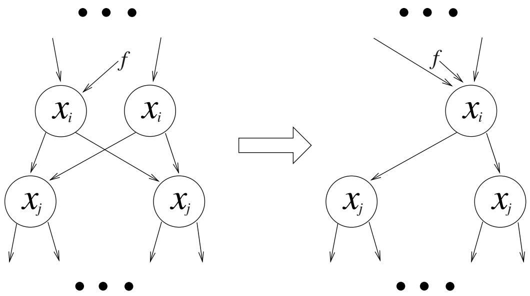

Figure 2.6. Merging Rule for SBDD-reduction.

Then the reduction $\mathrm{del}(v, v')$ *is applicable to* G*, resulting in a graph* $G[\mathrm{del}(v, v')]$ *obtained by a) redirecting all edges that are pointing to* v *to the node* v' *and b) inserting* v' *into* O *iff* $v \in O$ *and c) deleting* v *from* V *and* O*.*

- **Merging Rule:** *Let* $v, v' \in V$ *be two nodes with*

  1) $\mathrm{var}(v) = \mathrm{var}(v')$,

  2) $\mathrm{then}(v) = \mathrm{then}(v')$, *and*

  3) $\mathrm{else}(v) = \mathrm{else}(v')$.

  Then the reduction $\mathrm{merge}(v, v')$ *is applicable to* G*, resulting in a graph* $G[\mathrm{merge}(v, v')]$ *obtained by a) redirecting all edges pointing to* v *to the node* v' *and b) inserting* v' *into* O *iff* $v \in O$ *and c) deleting* v *from* V *and* O*.*

In both rules with action b) we ensure that output nodes do not "vanish". In Figures 2.5 and 2.6 the application of both reduction rules is illustrated. It is straightforward to see that the application of one of the two reduction operations del and merge does not change the function represented by the affected SBDD: first, with the Deletion Rule, redundant nodes are deleted. Let f_v denote the function represented by node v and let $\mathrm{var}(v) = x$. Provided that the Deletion Rule applies, we have $\mathrm{then}(v) = \mathrm{else}(v) =: v'$. Then, by Definition 2.14,

$$\begin{aligned} f_v &= x \cdot f_{\mathrm{then}(v)} + \overline{x} \cdot f_{\mathrm{else}(v)} \\ &= x \cdot f_{v'} + \overline{x} \cdot f_{v'} \\ &= (x + \overline{x}) f_{v'} = f_{v'}, \end{aligned}$$

hence deletion of v and redirection of all ingoing edges to the functionally equivalent node v' preserves the function represented by the diagram.

Second, with subsequent application of the Merging Rule isomorphic sub-graphs are identified. Provided that the Merging Rule applies, we have $\mathrm{var}(v) = \mathrm{var}(v') := x$ and the equivalence of the 1-children (0-children) of v and v'. Functional equivalence of the nodes v and v' follows directly from Definition 2.14: it is

$$\begin{aligned} f_v &= x \cdot f_{\mathrm{then}(v)} + \overline{x} \cdot f_{\mathrm{else}(v)} \\ &= x \cdot f_{\mathrm{then}(v')} + \overline{x} \cdot f_{\mathrm{else}(v')} \\ &= f_{v'}, \end{aligned}$$

thus using v' instead of v on every path of the BDD and discarding v collapses the graph but the represented function is not changed.

DEFINITION 2.16 *An SBDD is called* reduced *iff there is no node where one of the reduction rules (i.e., the Deletion or the Merging Rule) applies.*

The term “reduced” is used here since it has become common in the literature. This is done despite the fact that the older and more accurate term “irreducible” known from the paradigm of rewriting systems could be applied as well.

REMARK 1 *In the following, only reduced SBDDs are considered and for simplicity, they are called BDDs (or shared BDDs when explicitly excluding single-rooted BDDs).*

The *size of a BDD* $F = (\ldots, G, \ldots)$ is the number of nodes in the underlying graph G, denoted $|F|$ (or sometimes also $|G|$). The following theorem [Bry86] holds:

THEOREM 2.17 *BDDs are a canonical representation of Boolean functions, i.e. the BDD-representation of a given Boolean function with respect to a fixed variable ordering is unique up to isomorphism and is of minimal size.*

This property of BDDs is useful especially when checking the equivalence of two Boolean functions represented as a BDD. If the functions are indeed equivalent, their representation must be isomorphic. Actually, in modern, efficient BDD packages [BRB90, DB98, Som01, Som02] equivalent functions are represented by the same graph. Once the BDDs for the functions have been built, equivalence checking can be done in constant time (comparing the address of the root nodes of the representing graphs).

In a similar manner, it can be decided in constant time whether a Boolean function is satisfiable or not, once the BDD representing the function has been built: it suffices to check whether the BDD is the one for the constant zero function or not. In the latter case, the function must be satisfiable.

Moreover, it is known that computing a satisfying assignment of the n input variables can be done in time which is linear in n [Bry86].

The result of Theorem 2.17 is reflected in the next definition introducing a convenient notation to refer to the *one* BDD representing a Boolean function f with respect to a variable ordering π: we use the fact that the set of all BDDs representing a given Boolean function $f^{(n,m)}$ can be decomposed into equivalence classes of isomorphic BDDs, i.e. we have one class for every ordering.

DEFINITION 2.18 *Let $f^{(n,m)}$ be a Boolean function and let $\pi^{(n)}$ a variable ordering. Then* $\mathrm{BDD}_{n,m}(f,\pi)$ *denotes the equivalence class of BDDs representing f and respecting π. If n, m are clear, we simply write* $\mathrm{BDD}(f,\pi)$.

We may find it convenient to *identify* a class $\mathrm{BDD}(f,\pi)$ with an arbitrary *representative*, i.e. we speak of *the* BDD $\mathrm{BDD}(f,\pi)$ instead of the class of all BDDs isomorphic to the (chosen) representative. Formally, if $\overline{F}$ is a class of BDDs, we identify $\overline{F}$ with F.

An example of this would be using $\mathrm{BDD}(f,\pi)$ as parameter of a function which is expecting a BDD as argument. This simplification is harmless, as long as the following holds: the resulting function value must be preserved with respect to changes of the class representative. Instead of declaring a BDD with $F \in \mathrm{BDD}(f,\pi)$, this will be done in the form $F := \mathrm{BDD}(f,\pi)$. This is a relaxed notation, expressing that F is assigned a certain, fixed class representative. This notation is used to support complete abstraction from the details of graph isomorphism.

The introduced terminology directly transfers to BDDs with CEs. However, as has already been mentioned before in Section 2.4.2, special care has to be taken here to maintain a canonical form. The reason is the existence of several functional equivalences of distinct, reduced

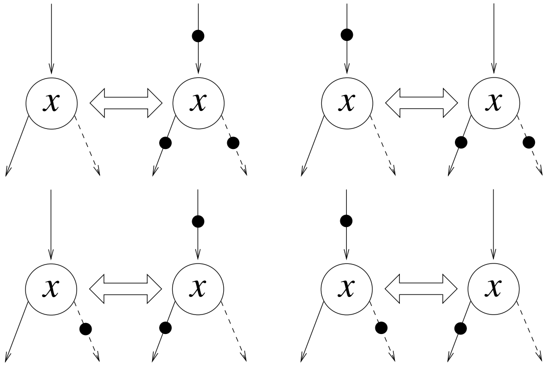

Figure 2.7. Pairs of functions represented as BDDs with CEs that are functional equivalent.

BDDs with CEs as illustrated in Figure 2.7. A dot on an edge indicates a complemented edge (otherwise it is a regular edge).

As a remedy, it must be constrained where CEs are used. To achieve a unique representation it suffices to restrict the 1-edge of every node to be a regular edge. Thus, in Figure 2.7 always the left member of each functionally equivalent pair is chosen. It can be shown that this already guarantees a canonical form. All function-preserving reduction operations which follow this constraint result in a unique BDD with CEs which also respects this condition.

2.4.4 Variable Orderings of BDDs

A problem with BDDs however is their *sensitivity to variable ordering.* To illustrate this, an example is reviewed which has been given in [Bry86].

EXAMPLE 2.19 *Let $n \in \mathbb{N}$ be even and let f: $\mathbf{B}^n \to \mathbf{B}$;*

$$(x_1, x_2, \ldots, x_n) \mapsto x_1 \cdot x_2 + x_3 \cdot x_4 + \ldots + x_{n-1} \cdot x_n.$$

A BDD for f respecting the variable ordering $\pi(1) = x_1, \pi(2) = x_2, \ldots,$ $\pi(n-1) = x_{n-1}, \pi(n) = x_n$ is given in Figure 2.8: the left BDD is of size n. Since f obviously depends on all n variables, this is the optimal size.

However, the right BDD for f respecting the variable ordering $\pi(1) = x_1, \pi(2) = x_3, \ldots, \pi(n/2) = x_{n-1}$, $\pi(n/2+1) = x_2, \pi(n/2+2) = x_4, \ldots, \pi(n) = x_n$ is of exponential size: it is straightforward to see that the graph is of size $2^{(n/2)+1} - 2$.

That is, depending on the variable ordering the size of a BDD may vary from linear to exponential in n, the number of input variables.

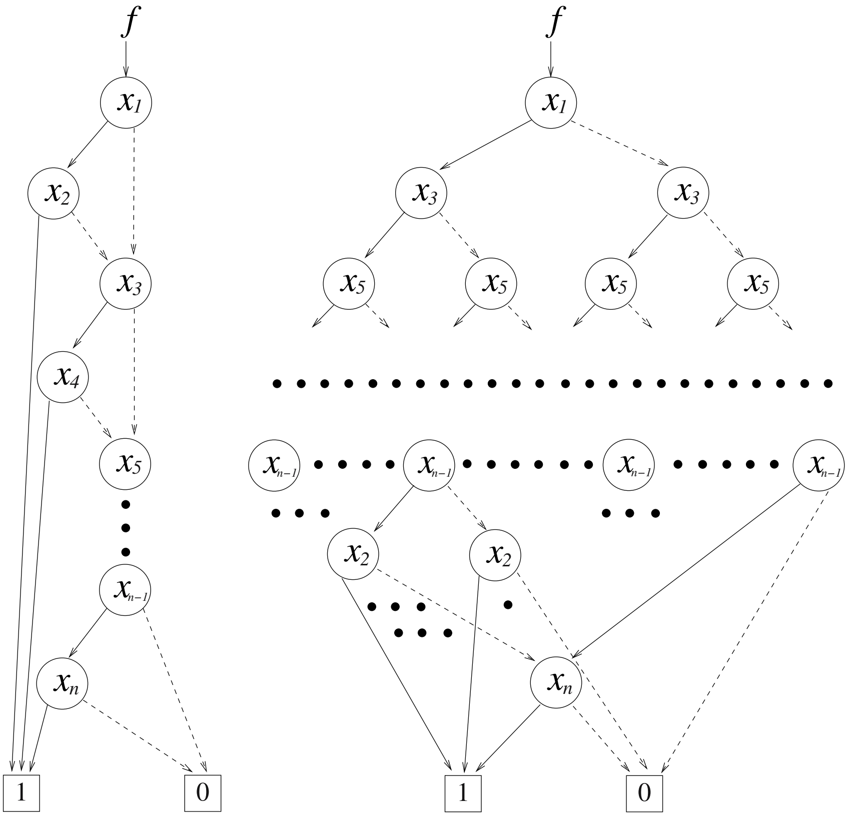

Figure 2.8. Two BDDs for $f = x_1 \cdot x_2 + x_3 \cdot x_4 + \ldots + x_{n-1} \cdot x_n$.

To describe approaches to minimization of BDD size (as will be done in Chapters 3 and 4), a formalism expressing variable orderings, changes of these orderings and movements of variables is necessary. This section introduces the notation used throughout this book for this purpose.

In some cases, when considering the variable ordering of a given BDD, the ordering is not needed explicitly. E.g. this is the case, when a BDD is given with respect to an initial ordering: considering methods for BDD size minimization, we are often interested in expressing (or restricting) the effect of the changes from one ordering to another. In this case we are not interested in the initial ordering from which a heuristic or an exact reordering method has been started.

In such cases we omit stating the ordering and thus in all examples, if not stated otherwise, the "natural" ordering defined by

$$\pi(i) = x_i \quad (i \in \{1, 2, \ldots, n\})$$

is assumed. To express movements (shifts) of variables in the ordering, i.e. changes in the position of a variable and exchanging a variable with another variable, often *permutations* have been used. E.g. [FS90], used bijections

$$\pi\colon \{1, 2, \ldots, n\} \to \{1, 2, \ldots, n\}$$

to express exchanges of variables. Here, variables x_i are referred to by their subscript i. A certain disadvantage is that both positions in the ordering and variable subscripts are denoted using natural numbers in $\{1, 2, \ldots, n\}$, hence they can be mixed up quite easily.

In [DDG00] bijections

$$\pi\colon X_n \to X_n$$

have been used instead. Both formalisms, although mainly designed to express *changes* in an ordering, also allow to express a variable ordering: for that we assume π is applied to the natural ordering, the result is then assumed to be the current ordering.

However, a formalism which directly gives the current position of each variable can be easier to read in most cases. Hence, instead of denoting orderings as permutations, throughout the book the mapping π from positions to variables is used, as already introduced in Definition 2.10. For the outlined historical reasons (see above), this bijection still is named π.

The next definition will be used later in Chapter 3 to express sets of orderings which respect a certain condition.

Definition 2.20 *Let $I \subseteq X_n$. Then $\Pi_n(I)$ denotes the set*

$$\Pi_n(I) = \{\pi\colon \{1, 2, \ldots, n\} \to X_n \mid \{\pi(1), \pi(2), \ldots, \pi(|I|)\} = I\}.$$

A mapping $\pi \in \Pi_n(I)$ is a variable ordering whose first $|I|$ positions constitute I. Normally we omit the arity n, writing $\Pi(I)$ instead of $\Pi_n(I)$, if n is given from the context.

This allows us to focus the attention to variable orderings with the following property: an ordering $\pi^{(n)} \in \Pi_n(I)$ partitions the set of variables X_n into a partition (L, R): let $(\pi, G, \ldots)$ be a BDD over X_n. Nodes with a label in $L = I$ reside in the upper part of G, i.e. in levels $1, 2, \ldots, |I|$. Nodes with a label in $R = X_n \setminus I$ reside in the lower part of G in levels $|I| + 1, |I| + 2, \ldots, n$. The definition also allows us to force the last $|I|$ positions of an ordering to constitute I: this holds for all $\pi \in \Pi(X_n \setminus I)$.

Here, the term "level" has been used informally. In the next section a formal definition of the term "BDD level" will be given.

2.4.5 BDD Levels and their Sizes

In this section additional notations are given which are needed to formally refer to parts of the BDD, to their sizes and to minimal sizes.

Definition 2.21 *A* level *of a BDD over X_n (or, equivalently, a BDD level) is a set containing all non-terminal nodes labeled with one particular variable x_i in X_n. Let $F = (\pi^{(n)}, \ldots)$ be a BDD over X_n. If $x_i = \pi(k)$, we speak of the "x_i-level" as well as the "k-th level" or "level k".*

Definition 2.22 *Let*

$$\text{nodes}_n\colon\ \{F \mid F \textit{ is a BDD over } X_n\} \times X_n \to 2^{\{v \in V \mid (\ldots,(V,E),\ldots) \textit{ is a BDD}\}};$$

$$\text{nodes}_n(F, x_i) = \{v \mid v \in V, \text{var}(v) = x_i \textit{ where } F = (\ldots,(V,E),\ldots)\}.$$

We straightforwardly extend this definition to also cover sets of variables, i.e. we define

$$\text{nodes}_n\colon\ \{F \mid F \text{ is a BDD over } X_n\} \times 2^{X_n} \to 2^{\{v \in V \mid (\ldots,(V,E),\ldots) \text{ is a BDD}\}};$$

$$\text{nodes}_n(F, X) = \bigcup_{x_i \in X} \text{nodes}_n(F, x_i).$$

For simplicity, this extension is denoted with the same function symbol. If n is clear from the context, we omit the subscript n, writing nodes(F, x_i) and nodes(F, X) instead of nodes$_n(F, x_i)$ and nodes$_n(F, X)$. The term nodes(F, x_i) denotes the set of nodes in the x_i-level of F whereas the term nodes(F, X) denotes the set of nodes in F labeled with a variable in $X \subseteq X_n$.

To express sizes of levels, i.e. the number of nodes in a level, we also introduce the following function.

Definition 2.23 $\text{label}_n\colon\ \{F \mid F \textit{ is a BDD over } X_n\} \times X_n \to \mathbb{N};$

$$\text{label}_n(F, x_i) = \left|\text{nodes}_n(F, x_i)\right|,$$

and $\text{label}_n\colon\ \{F \mid F \textit{ is a BDD over } X_n\} \times 2^{X_n} \to \mathbb{N};$

$$\text{label}_n(F, X) = \left|\text{nodes}_n(F, X)\right|.$$

As before, the subscript n is ommited if n is given by the context and the same function symbol is used both for the basic and the extended function. Note that the size of the k-th level can be expressed as label$(F, \pi(k))$. To be able to refer to the set of nodes in the k-th level in a BDD, we introduce the following definition.

Definition 2.24

$$\text{level: } \{F \mid F \textit{ is a BDD}\} \times \mathbb{N} \to 2^{\{v \in V \mid (\ldots,(V,E),\ldots) \textit{ is a BDD}\}};$$

$$\text{level}(F,k) = \{v \mid v \in V, \text{var}(v) = \pi(k) \textit{ where } F = (\pi,(V,E),\ldots)\}.$$

Let $F = (\ldots,\ldots,O)$ be a BDD. To express the set of nodes in parts of F covering (the output nodes situated at) several levels we introduce the following notations. Let

$$F_j^i = \bigcup_{i \le k \le j} \text{level}(F,k)$$

and let

$$O_j^i = O \cap F_j^i.$$

Due to the ordering restriction imposed on the variables of a BDD, it is possible to levelize each BDD graph illustration, i.e. to rectify the graph such that all nodes with the same label appear at the same level of height in the graph. Later, in Chapter 3, we will need to express the minimal number of nodes in BDDs or parts of BDDs. This can be done using the following very flexible definition.

Definition 2.25 *Let f be an n-ary Boolean function and let $I, J \subseteq X_n$.*

$$\text{min\_cost}_f(I,J) = \min_{\pi \in \Pi(J)} \left(\text{label}(\text{BDD}(f,\pi), I)\right),$$

i.e., min_cost$_f(I,J)$ *denotes the minimal number of nodes labeled with a variable in I under all BDDs representing f with a variable ordering whose first $|J|$ elements constitute J. If the function f is given from the context, we omit it, writing* min_cost(I,J) *for short.*

If $J = I$, all orderings are considered which change the order of variables within the upper part of the BDD in levels $1, 2, \ldots, |I|$ such that all nodes in this part remain labeled with a variable in I.

The term min_cost(I,I) expresses the size of the upper part for a "best" of all these orderings, i.e. the minimal size of the upper part with respect to a partition $(I, X_n \setminus I)$ of the set X_n.

If $J = X_n \setminus I$, all orderings are considered which change the order of variables within the *lower* part of the BDD in levels $n - |I| + 1, n - |I| +$

$2, \ldots, n$ such that all nodes in this part remain labeled with a variable in I.

The term $\text{min\_cost}(I, X_n \setminus I)$ expresses the size of the *lower* part for a "best" of all these orderings, i.e. the minimal size of the lower part with respect to a partition $(X_n \setminus I, I)$ of the set X_n.

Instead, we can also consider a partition $(I, X_n \setminus I)$, again considering all orderings which change the order of variables within the upper part of the BDD in levels $1, 2, \ldots, |I|$ and then express the minimal size of the *lower part* of the BDD under all these orderings: this would be expressed by the term $\text{min\_cost}(X_n \setminus I, I)$.

Yet, besides these possibilities, the definition allows to express even more sophisticated minimal sizes of BDD parts. But throughout this book, only the above forms will be needed and used.

Formally, min_cost_f is a function

$$\text{min\_cost}_f\colon\ 2^{X_n} \times 2^{X_n} \to \mathbb{N}.$$

The well-definedness follows from Theorem 2.17 which ensures that, for $I \subseteq X_n$, the term $\text{label}(\text{BDD}(f, \pi), I)$ is uniquely determined by f and π.

2.4.6 Implementation of BDD Packages

In the previous sections, the concept of BDDs has been introduced. In practice, the success of BDD-based algorithms relies on the efficient implementation of this concept. This section describes the basic implementation techniques used today by modern BDD packages. For more details see [BRB90, Som01].

In Section 2.4.3 it was explained why for typical applications like a functional equivalence check only reduced BDDs are of interest. For this reason it is desirable to avoid generating unreduced BDDs during the operation of a BDD package. A way of achieving this is to check whether a node representing a particular function already exists. This is done before the creation of a node representing this function. For a node v, the function represented by v, f_v, is uniquely determined by the tuple

$$(\text{var}(v), \text{then}(v), \text{else}(v)),$$

containing as its elements all arguments of the Shannon decomposition (see Theorem 2.9) carried out at v:

$$f_v = x \cdot f_{\text{then(v)}} + \overline{x} \cdot f_{\text{else(v)}}$$

where $x = \text{var}(v)$. Of course it would be too time-consuming to compare the tuple of a new node to all tuples of already existing nodes. Hence,

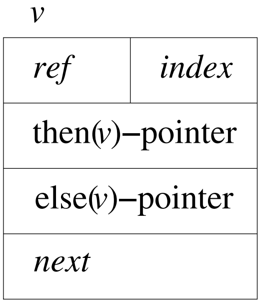

Figure 2.9. Node structure.

a global hash table, called the *unique table* is used which allows to find the node v for a tuple (x, v_1, v_0) in constant time. A hash function is computed on the tuple returning an index into an array of bins each storing the first node for that hash value. All other nodes with the same hash value are stored in a collision list. Usually, a next-pointer in the node structure is used to implement this linked list. Other pointers stored at each node are the pointers to the 1-child and the 0-child. Note that only forward-pointers are used to form the DAG, as backward-pointers would increase the node structure and cause too much memory overhead. Besides this, also the variable index *index* (plus flags, i.e. the complement bit, see Section 2.4.3) and a reference count *ref* are stored in the node structure (see Figure 2.9).

The reference count *ref* gives the information how often a node is used at the moment. That way it is possible to free a node if it is not used anymore. This allows the efficient usage of the memory. The count *ref* is updated dynamically whenever the number of references from other nodes or returned user functions changes. If *ref* has reached its maximum value, it is frozen, i.e. the node is not deleted during program run and exists until program termination.

If *ref* reaches the value zero for a node v, the node is called a *dead node*. The reference counts of the descendants of v need to be decreased recursively. However, the memory of the dead node v is not freed immediately, because there might still exist an entry in the so-called *computed table* which points at v.

The computed table is a global cache storing the intermediate and final results of the recursive algorithms which operate on the DAG of the BDD. A result in this context means a node of the DAG which is the root of a sub-graph representing the computed function.

The idea is to trade memory vs. run time: if a result can be found in the cache, it has been computed before and it is avoided to compute it again. As the number of results can be high during a program run with many BDD operations, the computed table would soon grow too large if implemented as a standard hash table. Therefore usually it is implemented as a *hash-based cache*.

Though the result of a new computation is stored in the cache (again, in form of a tuple pointing to a node via hashing), the handling is different from that of the unique table: rather than appending a new entry to a collision chain, an entry is simply overwritten if a collision occurs. That way a good trade-off between memory requirement and run time improvement can be achieved. Note that it is no problem if a result node is a dead node. In this case a *reclaim* operation, increasing the reference count, re-establishes the node. The reclaimed node is not a dead node anymore.

The reference count also allows for an efficient memory management of the unique table: if a new node is created causing the unique table to become too full, then there are two possible ways of remediation: as a first possibility a *garbage collection* can be performed. This is only done if the number of dead nodes exceeds a certain threshold. Otherwise freeing the dead nodes would not result in a sufficient amount of memory to be recycled, not making the garbage collection worthwile. During garbage collection, dead nodes are freed and entries in the computed table pointing to dead nodes are deleted. Second, if not enough dead nodes are available, both the unique table array and the computed table cache are increased in size, typically by a factor of two. As a consequence, all entries must be assigned a new valid position. This happens using the hash function whose return values have now changed due to the increase of the table size. This process is called *rehashing*. During rehashing, typically also an implicit garbage collection is performed.

Since the tables are increased by a factor of two every time, the rehashing operation is involved only logarithmically often. If both methods to obtain more memory, garbage collection and table size increase with rehashing, fail, the system returns a NULL pointer to the user. At this point it is up to the user to save space by freeing nodes.

Another important implementation aspect used by modern BDD packages is the use of an array of unique tables, one per level $i = 1, \ldots n$. Index i serves to locate the hash table which stores all nodes for level i. Then all nodes of level i can be traversed by stepping through all collision chains, each starting at a hash table array position.

2.4.7 Basic Minimization Algorithm

Algorithms for the minimization of BDDs make use of basic techniques such as *variable swaps* and the *sifting* algorithm. This method has been described in [Rud93] and exploits the only local complexity of the swap operation [ISY91] to achieve good run times. This section gives both basic techniques as the algorithms for BDD optimization presented in Chapters 3, 4, and 5 are based on them.

2.4.7.1 Variable swap

This section describes how a swap of adjacent variables can be implemented such that it is performed with a run time which is proportional to the sizes of the neighboring levels at which the variables are situated, i.e. a *local* behavior.

A problem which is encountered when trying to achieve a local behavior during a swap is the following: assume that a variable swap results in the change of the function represented by a BDD node v. As a consequence, all previous references to v in the DAG are not valid anymore and need to be corrected. In Section 2.4.6 it has already been described that modern BDD packages do not maintain back-pointers (from child nodes to parent nodes) in order to reduce memory requirement. Hence, to find all references to v from nodes situated above, we would have to traverse large parts of the DAG, and even then the problem to patch the references from user functions still remains.

For this reason it is necessary to preserve the function represented at each node involved in a swap. This is achieved as follows: instead of constructing a new node in another part of the computer's memory, the data of the node in question (i.e. its tuple, see Section 2.4.6) is overwritten with new values in a function preserving manner. These new values result from the change in the variable ordering. Besides avoiding the problem described above, another advantage of this schema is that no time-consuming memory allocations are needed.

In detail, this is done as follows. Assume a natural variable ordering and assume that variables $x_i = \pi(i)$ and $x_{i+1} = \pi(i+1)$ are swapped. Let v be a node situated at level i and let v_1, (v_0) denote the 1-child (0-child) of v.

Let f_v denote the function represented by v and let f_1 (f_0) denote the positive (negative) cofactor of f_v with respect to x_i. Clearly, f_1 (f_0) is represented by v_1 (v_0). Further, let f_{11}, (f_{10}) denote the positive (negative) cofactor of f_1 with respect to x_{i+1}. Similarly, let f_{01}, (f_{00}) denote the positive (negative) cofactor of f_0 with respect to x_{i+1}. Let v_{11} be the node representing f_{11}. This is either the 1-child of v_1 (if

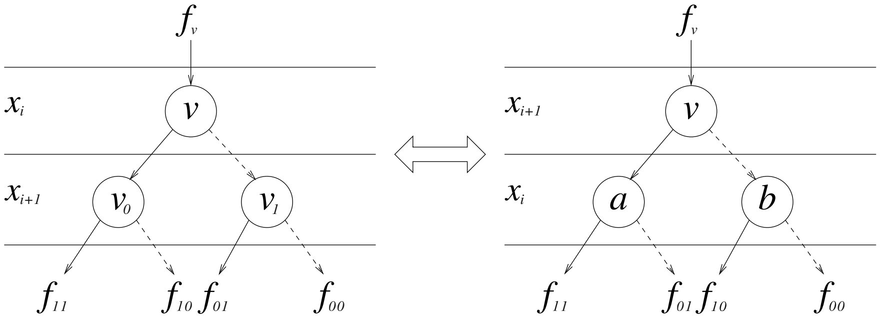

Figure 2.10. Variable swap.

x_{i+1} is tested at v_1) or simply v_1, otherwise. Similarly, let v_{10}, and v_{01}, v_{00} be the nodes representing f_{10}, and f_{01}, f_{00}, respectively. The left side of Figure 2.10 illustrates the most general case where all cofactors are distinct functions. If some of the cofactors f_{11}, f_{10}, f_{01}, and f_{00} are functionally equivalent, the respective representing nodes have been collapsed by a merge operation before and thus less nodes are involved in the swap than depicted in Figure 2.10.

Node v is represented by the tuple (x_i, v_1, v_0). Performing the swap of x_i and x_{i+1}, v is overwritten by $(x_{i+1}, (x_i, v_{11}, v_{01}), (x_i, v_{10}, v_{00}))$. Inspecting the paths through v shows that the new variable ordering (i.e., x_{i+1} “above” x_i) is established (see the right side of Figure 2.10). Moreover, this modification preserves the function represented by v:

$$\begin{aligned}
&(x_{i+1}, (x_i, v_{11}, v_{01}), (x_i, v_{10}, v_{00})) \\
&\quad = \quad (x_{i+1}, x_i \cdot f_{11} + \overline{x}_i \cdot f_{01}, x_i \cdot f_{10} + \overline{x}_i \cdot f_{00}) \\
&\quad = \quad x_{i+1} \cdot (x_i \cdot f_{11} + \overline{x}_i \cdot f_{01}) + \overline{x}_{i+1} \cdot (x_i \cdot f_{10} + \overline{x}_i \cdot f_{00}) \\
&\quad = \quad x_i \cdot x_{i+1} \cdot f_{11} + \overline{x}_i \cdot x_{i+1} \cdot f_{01} + x_i \cdot \overline{x}_{i+1} \cdot f_{10} + \overline{x}_i \cdot \overline{x}_{i+1} \cdot f_{00} \\
&\quad = \quad x_i \cdot (x_{i+1} \cdot f_{11} + \overline{x}_{i+1} \cdot f_{10}) + \overline{x}_i \cdot (x_{i+1} \cdot f_{01} + \overline{x}_{i+1} \cdot f_{00}) \\
&\quad = \quad x_i \cdot f_1 + \overline{x}_i \cdot f_0 \\
&\quad = \quad f_v
\end{aligned}$$

The situation after the swap is illustrated on the right side of Figure 2.10. Nodes v_0 and v_1 will only vanish, if no other (external or user) reference besides those from node v existed. Nodes a and b must be newly created only if nodes with their tuples did not already exist, otherwise they are simply retrieved from the unique table (see Section 2.4.6).

2.4.7.2 Sifting Algorithm

In 1993, Rudell presented an effective algorithm to reduce the size of a BDD, the *sifting algorithm* [Rud93]. Every variable is moved up and down in the variable ordering, i.e. the relative order of the other variables is preserved. At each position the resulting BDD size is recorded and in the end the variable is moved back to a best position, i.e. one which yielded the smallest BDD size. The variable movements are performed by swaps of variables which are adjacent in the variable ordering, e.g. for $i < j$ swapping $\pi(i)$ and $\pi(i{+}1)$, $\pi(i{+}1)$ and $\pi(i{+}2)$ . . . , finally swapping $\pi(j-1)$ and $\pi(j)$ moves $\pi(i)$ to the j-th position in the ordering. Note that π changes with each swap.

In Section 2.4.7.1 it has been shown that a swap of adjacent variables only affects the graph structure of the two levels involved in the swap. Since the number of nodes which have to be touched directly transfers to the run time of the method, this *locality* of the swap operation is a main reason for the efficiency of the sifting algorithm: these exchanges are performed very quickly since only edges must be redirected within these two levels. Moreover, changes in the graph structure can often be established simply by updates of the node data, thus saving the cost of many memory allocations. In this, sifting is based on effective local operations. Moreover, further reducing the number of swaps obviously yields reductions in the run time of the method.

In the past, several methods of achieving reductions in the number of variable swaps have been suggested. A trivial method for example is to start with moving the variable to the *closest end first* when moving a variable to all other positions in the relative order of the variables.

A much more sophisticated method is the use of lower bounds during sifting, which will be discussed in more detail in Chapter 4.

Another improvement suggested is the idea to start with the variable situated at the level with the largest number of nodes: this helps to reduce the overall BDD size as early as possible, thus reducing the complexity of all subsequent steps.

Summarized, the classical sifting algorithm works as follows:

1 The levels are sorted according to their sizes. The largest level is considered first.

2 For each variable:

   (a) The variable is first moved downwards if it is situated closer to the bottommost variable, otherwise it is first moved upwards. Moving the variable means exchanging it repeatedly with its successor variable if moved downwards or with its predecessor variable if moved upwards. This process stops, when the variable has become the bottommost variable (if moved downwards) or the topmost variable (if moved upwards).

(b) The variable is moved into the opposite direction of the previous step, i.e. if in the previous step the variable has been moved downwards, now it is moved upwards until it has become the topmost variable. If, however, in the previous step the variable has been moved upwards, now it is moved downwards until it has become the bottommost variable.

(c) In the previous steps the BDD size resulting from every variable swap has been recorded. Now the variable is moved back to the closest position among those positions which led to a minimal BDD size.

The algorithm is given in Figure 2.11. For $i < j$, a call sift_down(i, j) moves a variable from level i down to j (the other procedure calls have similar semantics). We start with the largest level first, hence in lines (5), (7), and (9), $sl[i]$ denotes the number of the largest unprocessed level. In line (9) the tested condition is a check whether the way down is shorter than the way up. In that case, the algorithm goes down first. In line (16) the algorithm moves the variable back to a best position. This is usually done by following a sequence of recorded moves in reverse order.

Note that the run time of sifting increases, if the BDD becomes large in intermediate steps of the algorithm. To prevent high run times, moving a variable into a specific direction can be cancelled if the BDD size exceeds a certain limit, e.g. a limit of twice the size of the initial BDD.

2.4.8 Evaluation with BDDs

Besides the ability of BDDs to represent a Boolean function, BDDs can be also used to actually *implement an evaluation* of the function.

While evaluating functions with BDDs, one has to consider *paths* in BDDs, starting at one of the output nodes and ending at a terminal node. Sometimes also paths to an inner node are considered. Since BDDs essentially are DAGs, they inherit the usual standard notations considering paths in graphs. Paths in BDDs, as is common for graphs, are denoted as alternating sequence of nodes v_i and edges e_i, i.e. $(v_1, e_1, \dots, e_{k-1}, v_k)$. The length of a path p is the number of non-terminal nodes occurring on p, denoted $\lambda(p)$. Next, an evaluation of a BDD with respect to an assignment is defined in operational terms.

```
sifting(BDD $F$, int $n$)
  proc
    sort level numbers by descending level sizes and store
    them in array $sl$;
    for $i := 1$ to $n$ do
      if $sl[i] = 1$ then
        sift_down($i, n$);
      else if $sl[i] = n$ then
        sift_up($i, 1$);
      else if $(sl[i] - 1) > (n - sl[i])$ then
        sift_down($i, n$);
        sift_up($n, 1$);
      else
        sift_up($i, 1$);
        sift_down($1, n$);
      end–if
      sift_back();
    end–for
  end–proc
```

Figure 2.11. Original sifting-algorithm.

An assignment $b = (b_1, \ldots, b_n) \in \mathbf{B}^n$ denotes the function

$$b\colon\ 2^{X_n} \to \mathbf{B}^n; x_i \mapsto b_i \quad (1 \leq i \leq n).$$

DEFINITION 2.26 *An* evaluation *of a BDD $(\ldots, (V, E), O)$ with respect to an assignment $b = (b_1, \ldots, b_n) \in \mathbf{B}^n$ starts at one of the output nodes in O and traverses the path along the edges in E which are chosen according to the values assigned to the variables by b. Thereby all variables which are not tested along the traversed path are ignored.*

The evaluation is said to reach *a node $v \in V$ if v occurs on the traversed path. The evaluation is said to* stop *at node v if v is the last node of the traversed path.*

Due to the BDD semantics as a graph where a Shannon decomposition is carried out at each node, we have $f(b_1, b_2, \ldots, b_n) = 1$ iff the evaluation stops at $\mathbf{1}$ and $f(b_1, b_2, \ldots, b_n) = 0$ iff the evaluation stops at $\mathbf{0}$.

EXAMPLE 2.27 *Consider the left BDD given in Figure 2.12. The evaluation for $b = (0, 0, 1)$ starts at the output node for f which is the root node of the BDD. Assignment b assigns x_1 to 0, x_2 to 0, and x_3 to 1. According to these values, the path along the corresponding edges labeled*

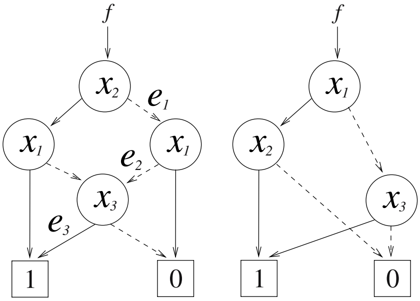

Figure 2.12. Two BDDs for f: $(x_1, x_2, x_3) \mapsto x_1 \cdot x_2 + \overline{x}_1 \cdot x_3$.

e_1, e_2, and e_3 is chosen (1-edges are depicted with solid lines, 0-edges with dashed lines). This path finally reaches the terminal node labeled 1, and indeed the function value $f(0,0,1)$ equals 1.

Note that the path along e_1, e_2, and e_3 is of maximal length three in the left BDD whereas the right BDD has a maximal path length of only two.

Let us consider a BDD respecting a variable ordering π. Sometimes we may choose to assign values to only the first few variables in the ordering π, thus considering a possibly shorter prefix $a = (b_1, \ldots, b_k)$ $(k \leq n)$ of a (full) assignment b. The operational semantics of an evaluation directly transfers to this situation in complete analogy. Evaluation of a stops at a (possibly non-terminal) node v, representing the cofactor $f_{x_1=b_1,\ldots,x_k=b_k}$ for which we also write f_a.

2.4.9 Paths in BDDs

The optimization of BDDs with respect to different aspects of their paths will be subject to further discussion in Chapter 5. Thereby CEs (see Sections 2.4.2 and 2.4.3) will be explicitly considered to distinguish paths to zero from paths to one. Formally BDDs with CEs have to be represented by a edge-labeled graph in order to explicitly represent edges from a node v to then(v) and else(v), respectively. This is necessary as the following example illustrates.

Example 2.28 *Consider the projection function for variable x_1 shown in Figure 2.13. Both outgoing edges of node v lead to the terminal **1**. Therefore they would correspond to a single edge in the graph structure without labels and the complement could not be properly associated to the edge leading to* else(v)*. This ambiguity is removed using the edge-labeled graph that contains two edges.*

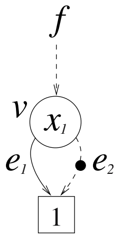

Figure 2.13. BDD with CEs for $f(x_1) = x_1$.

The following definitions are given for the more difficult case of BDDs with CEs only. For an edge $e \in E$ the attribute CE(e) is true, iff e is a CE. The edges e_i occurring on a path in a BDD may be complemented or non-complemented. An (implicit) edge e pointing to the root node has to be considered to represent a function. Such an implicit (possibly complemented) edge into the BDD is used to represent f or $\overline{f}$, respectively. This edge is not explicitly denoted.

The predecessors of a node w are split into those having a CE to w and those having a regular edge to w, given by two sets (the sets are not always disjoint):

$$\begin{aligned}\mathcal{M}_1(w) &:= \{v : w \text{ can be reached from } v \text{ via a regular edge}\}\\ \mathcal{M}_0(w) &:= \{v : w \text{ can be reached from } v \text{ via a CE}\}\end{aligned}$$

Regarding the following terminology for paths in BDDs, it should be noted that for a BDD with CEs the implicit edge representing a function has to be considered.

Definition 2.29

Two paths

$$\begin{aligned}p_1 &= (v_0, d_0, v_1, d_1, \ldots, d_{l-1}, v_l),\\ p_2 &= (w_0, e_0, w_1, e_1, \ldots, e_{l-1}, w_l)\end{aligned}$$

with $v_i, w_i \in V$ and $e_i, f_i \in E$ are identical*, iff*

$$\begin{array}{rcl}\forall i \in \{0,\ldots,l\} & v_i & = w_i,\\ \forall i \in \{0,\ldots,l-1\} & d_i & = e_i,\\ \forall i \in \{0,\ldots,l-1\} & \mathrm{CE}(d_i) & = \mathrm{CE}(e_i).\end{array}$$

Otherwise the two paths are called different*.*

Definition 2.30 *A path $p = (v_0, e_0, v_1, e_1, \ldots, e_{l-1}, v_l)$ is called* complemented *iff it leads from v_0 to v_l via an odd number of CEs, i.e.*

$$|\{e : (e \textit{ is an edge in } p) \wedge \mathrm{CE}(e)\}| = 2i + 1$$

for some $i \in \mathbb{N}$. Otherwise the path is called regular*.*

Definition 2.31 *A path $p = (v_0, e_0, v_1, e_1, \ldots, e_{l-1}, v_l)$ is called a* **1**-path *("one-path")* from v_0 *iff the path is regular and $v_l = \mathbf{1}$. A path $p = (v_0, e_0, v_1, e_1, \ldots, e_{l-1}, v_l)$ is called a* **0**-path *("zero-path")* from v_0 *iff the path is complemented and $v_l = \mathbf{1}$.*

2.4.9.1 Number of Paths

Notation 1 *Let f be a Boolean function. $P_1(\mathrm{BDD}(f, \pi))$ denotes the number of all different* **1***-paths from any of the outputs to the terminal* **1** *with respect to the variable ordering π. $P_0(\mathrm{BDD}(f, \pi))$ denotes the number of all different* **0***-paths from any of the outputs with respect to the variable ordering π. To denote the number of all paths in* $\mathrm{BDD}(f, \pi)$*, we use the short notation*

$$\alpha(\mathrm{BDD}(f, \pi)) = P_1(\mathrm{BDD}(f, \pi)) + P_0(\mathrm{BDD}(f, \pi)).$$

Example 2.32 *The BDD for the well-known odd-parity or EXOR-function f: $\mathbf{B}^n \rightarrow \mathbf{B}$; $(x_1, \ldots x_n) \mapsto x_1 \oplus \ldots \oplus x_n$ is of linear size $2n + 1$. The corresponding BDD is shown in Figure 2.14. Nonetheless the number of paths is exponential in n and, even worse, all BDDs representing the EXOR-function have a number of paths exponential in n. Due to the symmetry of the EXOR-function with respect to all variables, the structure of the BDD is independent of the variable ordering.*

Example 2.32 shows that the number of paths can grow exponentially in n, the number of input variables, even if the size of the BDD grows only linear in n.

When computing the number of paths in a BDD (see Section 5.1 and in particular Section 5.3) an exponential run time should be avoided. An appropriate formula for the number of paths in a BDD F can be derived from the Shannon decomposition (Theorem 2.9). The following recurrent equation expresses the number of paths starting at a node v and ending at a terminal node (this quantity denoted $\alpha(v)$):

$$\alpha(v) = \begin{cases} 1, & v \in \{\mathbf{1}, \mathbf{0}\} \\ \alpha(\mathrm{then}(v)) + \alpha(\mathrm{else}(v)), & \mathrm{else} \end{cases} \tag{2.2}$$

The formula simply states that $\alpha(v)$ is one if v is a terminal node (there is one path of length zero from v to v). Otherwise, the paths starting at

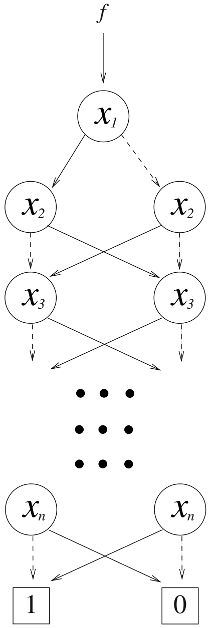

Figure 2.14. BDD for Example 2.32.

v and ending at a terminal node can be partitioned into the paths via the 1-child and those via the 0-child, due to the Shannon decomposition carried out at each node. The sum of the number of paths in these two partitions yields the total number of paths.

It is straightforward to give an algorithm computing the number of paths during a graph traversal (an example of the recursive computation of the α-values is illustrated later in Section 5.3). The number of paths in a shared BDD F representing a Boolean multi-output function $f = (f_i)_{1 \leq i \leq m}$ can be computed as the sum of the α-values of the output nodes representing the m single-output functions:

$$\alpha(F) = \sum_{i=1}^{m} \alpha(o_i) \tag{2.3}$$

where o_i is the output node representing f_i. Note that an output node o_i might be used by multiple, functionally equivalent single-output functions, as circuits sometimes repeat an output signal several times. In other words, paths emerging from output nodes used for more than one function are counted several times accordingly.

2.4.9.2 Expected Path Length

An important problem of VLSI CAD is *simulation-based verification*: here, a circuit is simulated to check whether the design fullfills its specification. The main difference of cycle-based *functional simulation* to classical simulation is that only values of output and latch ports are computed. Applications in this field repeatedly evaluate logic functions with different input vectors. This can be done using BDDs as described in Section 2.4.8.

During evaluation, a path is traversed. Thereby the algorithm moves from the start node of the path to the end of the path, following the respectiv edges. Pointers to the respective next node on the path must be dereferenced repeatedly. Hence, the evaluation time is linear in the length of the path. In BDD-based functional simulation, the average evaluation time (and hence the total simulation time) is proportional to the *Expected Path Length* (EPL) in the BDD. Next, a formal definition of the expected path length follows. EPL expresses the expected number of variable tests needed to evaluate a BDD with respect to an input assignment along a path from an output node to one of the terminal nodes, as explained and defined above in Section 2.4.8.

Let F be a BDD and let p_i be the i-th path in an enumeration of all paths from output nodes to one of the terminal nodes in F. Let $\mathrm{pr}(p_i)$ be the probability of an evaluation traversing path p_i. Then for the EPL of F, denoted $\epsilon(F)$, we have

$$\epsilon(F) = \sum_{i=1}^{\alpha(F)} \lambda(p_i) \cdot \mathrm{pr}(p_i). \tag{2.4}$$

In this formula, a path p is weighted with the probability of being chosen during evaluation. Minimizing $\epsilon(F)$ means shortening the path lengths with a high probability, thus minimizing the expected path length or, in other words, the average evaluation time.

Equation (2.4) is not suitable for use by an efficient algorithm to compute EPL, as $\alpha(F)$ can grow exponentially in n, even if the size of the BDD only grows linear in n: a function and the respective BDD with this property has already been given in Example 2.32.

Next, a formula is given which is useful for computing EPL in a time proportional to the BDD size: the following equation expresses the expected number of variable tests for an evaluation starting from a node v and ending at one of the terminal nodes (this quantity denoted $\epsilon(v)$).

Thereby the probability that a variable x is assigned to a value $b \in \mathbf{B}$ is denoted by $\mathrm{pr}(x = b)$.

$$\epsilon(v) = \begin{cases} 0, & v \in \{\mathbf{1}, \mathbf{0}\} \\ 1 \quad + \quad \mathrm{pr}(\mathrm{var}(v) = 1) \cdot \epsilon(\mathrm{then}(v)), & \text{else} \\ \quad\;\; + \quad \mathrm{pr}(\mathrm{var}(v) = 0) \cdot \epsilon(\mathrm{else}(v)) & \end{cases}$$

This formula simply states that $\epsilon(v)$ is zero if v is already a terminal node. Otherwise, evaluations starting from v are either via the 1-child or the 0-child of v. Hence, $\epsilon(v)$ is built by

1) summing up the respective ϵ-values of the child nodes of v weighted with the probability of the respective child node being chosen, and

2) adding one since the expected length of all paths starting at v must be one larger than that of the child nodes of v: this is due to the additional variable test at v.

Again it is straightforward to compute $\epsilon(F)$ during a graph traversal. An example of the recursive computation of the ϵ-values is illustrated later in Section 5.2 by the left BDD in Figure 5.8. In analogy to Equation (2.3), the EPL for a shared BDD F representing a Boolean multi-output function $f = (f_i)_{1 \le i \le m}$ can be computed by use of the ϵ-values of the output nodes representing the m single-output functions:

$$\epsilon(F) = \frac{1}{m} \sum_{i=1}^{m} \epsilon(o_i)$$

where o_i is the output node representing f_i. Again, note that an output node o_i might be used by multiple, functionally equivalent single-output functions.

2.4.9.3 Average Path Length

In Section 5.3, optimization of BDDs with respect to the following aspect of their paths will be discussed. The motivation for this is the mapping of optimized BDDs into fast multiplexor-based circuits.

Let F be a BDD and let $\lambda(F)$ denote the sum of the lengths of the paths in F. Again, let p_i be the i-th path in an enumeration of all paths from output nodes to the terminal node in F. Then $\lambda(F)$ can be expressed as

$$\lambda(F) = \sum_{i=1}^{\alpha(F)} \lambda(p_i). \tag{2.5}$$

Equation (2.5) can be seen as a special instance of Equation (2.4) where all path probabilities $\mathrm{pr}(p_i) = 1$.

The *Average Path Length* (APL) of F is denoted $\overline{\lambda}(F)$ and is defined as

$$\overline{\lambda}(F) = \frac{\lambda(F)}{\alpha(F)}. \tag{2.6}$$

Equation (2.6) can be seen as a special case of Equation (2.4), defining the expected path length: the two equations coincide, if in Equation (2.4) the term $\mathrm{pr}(p_i)$ is fixed to $\frac{1}{\alpha(F)}$ for each p_i.

A recurrent equation that is more useful for the efficient computation of the sum of the lengths of the paths is derived from the Shannon decomposition. This equation is given in the next result.

Lemma 2.33 *For a node v in a BDD, let $\lambda(v)$ denote the sum of the lengths of all paths starting at v and ending at a terminal node. Then we have*

$$\lambda(v) = \begin{cases} 0, & v \in \{\mathbf{1}, \mathbf{0}\} \\ \lambda(\mathrm{then}(v)) + \alpha(\mathrm{then}(v)) & \\ \quad + \lambda(\mathrm{else}(v)) + \alpha(\mathrm{else}(v)), & \textit{else} \end{cases} \tag{2.7}$$

Proof. In the case of $v \in \{\mathbf{1}, \mathbf{0}\}$ there is nothing to show. Now let v be an inner node. Let $p = (v, e_1, \mathrm{then}(v), \ldots, t)$ be a path from v via $\mathrm{then}(v)$ onto a terminal node t. Further, let $p' = (\mathrm{then}(v), \ldots, t)$ be a path coinciding with p except that we start at $\mathrm{then}(v)$ instead. It is $\lambda(p) = \lambda(p') + 1$, i.e. the length of every path via $\mathrm{then}(v)$ is increased by one if started at v instead. There are $\alpha(\mathrm{then}(v))$ such paths. Consequently, the sum of the lengths of all paths from v via $\mathrm{then}(v)$ onto a terminal node must be $\lambda(\mathrm{then}(v)) + \alpha(\mathrm{then}(v))$. An analogous argument holds for paths via $\mathrm{else}(v)$. The set of all paths starting from v and ending at a terminal node can be partitioned into those via $\mathrm{then}(v)$ and those via $\mathrm{else}(v)$. But then the required result already follows. □

Similar to Equations (5.8) and (2.3), the sum of path lengths for a shared BDD F representing a Boolean multi-output function $f = (f_i)_{1 \le i \le m}$ can be computed by use of the λ-values of the output nodes representing the m single-output functions:

$$\lambda(F) = \sum_{i=1}^{m} \lambda(o_i)$$