Symbolic Representation and Manipulation of Discrete Functions

Diploma Thesis

Jörn Ossowski

RHEINISCHE FRIEDRICH-WILHELMS-UNIVERSITÄT BONN
Institute of Computer Science I

4th November 2004

universitätbonn

Acknowledgements

This work would not have been possible without the support of Professor Dr. Christel Baier, under whose supervision I got the opportunity to explore a new field with a chance to do real science.

Special thanks go to Frank, Jasmin, Marcus, Rolf, Sascha, Tobias and Wolfgang for providing this work with ideas and criticism.

I want to thank my girl-friend Claudia for her endless patience and understanding. Thanks to my brother Martin and all who helped to make this thesis possible.

Lastly, and most importantly, I would like to express my sincere appreciation to my parents. To them I dedicate this thesis.

0 Contents

Chapter 1 Introduction **1**

Chapter 2 Shared Binary Decision Diagrams **3**

2.1. Notations and Definitions 3
2.2. Binary Decision Diagrams 4
2.3. Ordered Binary Decision Diagrams 6
2.3.1. Reduced Ordered Binary Decision Diagrams 8
2.3.1.1. Binary Boolean Operators 10
2.3.1.2. Test for Equality 13
2.3.1.3. The Boolean Satisfiability Problem 14
2.3.1.4. Cofactors 14
2.3.1.5. Negation 14
2.3.1.6. Asymptotic Complexity 14
2.3.1.7. The Variable Ordering Problem 16
2.3.2. Shared Ordered Binary Decision Diagrams 18
2.3.2.1. Test for Equality 20
2.3.2.2. Negation 20
2.3.2.3. Attributed Edges 20
2.3.3. SOBDD with Negative Edges 21
2.3.3.1. Algorithms on SOBDDs with negative edges . . 24
2.3.3.2. Negation 24
2.3.3.3. Asymptotic Complexity 26
2.4. Algebraic Computation 26

2.4.1. Integer Computation 26
2.4.1.1. Benchmark 27
2.4.2. Matrix Representation 28

Chapter 3 Algebraic Binary Decision Diagrams 31

3.1. Syntax and Semantics 31
3.2. Algorithms 32
3.3. Algebraic Computation 33
3.3.1. Basic Algebraic Operations 34
3.3.1.1. Benchmark 34
3.3.2. Matrix Representation 35
3.3.2.1. Benchmark 36

Chapter 4 Normalized Algebraic Binary Decision Diagrams 39

4.1. Definition and Semantics 39
4.2. Canonicity 44
4.3. Algorithms 55
4.4. Algebraic Computation 57
4.4.1. Basic Algebraic Computation 57
4.4.1.1. Benchmark 57
4.4.2. Matrix Representation 59
4.4.2.1. Benchmark 60
4.5. Implementation 60
4.5.1. Influence of λ_ε and τ_ε 61
4.5.2. Storing the node parameters 61

Chapter 5 Experimental Results 63

5.1. Algebraic Computation 63
5.2. Matrix Representation 66

Chapter 6 Conclusion and Perspective 69

1 Introduction

Symbolic representation and manipulation of discrete functions over Boolean variables plays an important role in a wide range of applications, e.g. symbolic model checking, computer-aided design (CAD) and very large scale integration (VLSI). Many problems can be expressed in terms of operations over finite domains. These domains can be understood by their binary encoding. This could be seen as a generalization of the problem to find appropriate data structures for Boolean functions. If Boolean functions can be represented and manipulated in an efficient way many complex problems can be solved symbolically.

There are a lot of different ways to represent Boolean functions. Table 1.1 shows the truth table representation of the function

$$f(x, y, z) = y \vee z.$$

Algorithms to manipulate truth tables have the complexity $\Omega(2^n)$ for n input variables. Another representation, which has the same complexity, are binary decision trees. Figure 1.1 shows a decision tree for the function $f(x, y, z) = y \vee z$. To evaluate the value of the function the tree has to be traversed from the root to a drain.

Binary Decision Diagrams (BDDs) [Lee 1959, Akers 1978] provide a more efficient method for representing and manipulating Boolean functions than binary

x	y	z	$f(x, y, z)$
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Table 1.1: Truth Table

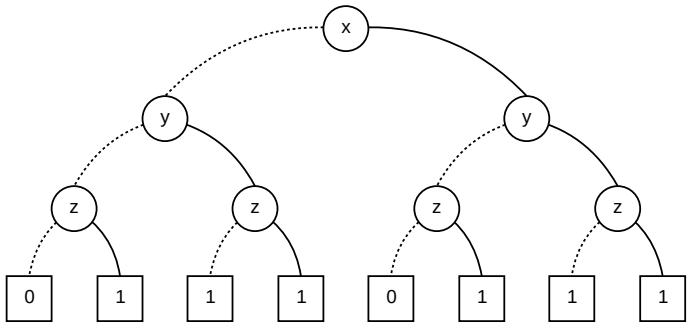


Figure 1.1: Binary Decision Tree

decision trees and truth tables do. With the use of BDDs various complex problems can be solved.

Ordered Binary Decision Diagrams (OBDDs) [Bryant 1986] are BDDs with some restrictions that assure a canonical form to represent Boolean functions. With this property the equality of functions can be checked without difficulty.

Some problems cannot be represented by their Boolean encoding in an efficient way. That was the reason why Algebraic Decision Diagrams[1] (ADDs) [Bahar 1993] were developed. These are BDDs that can contain multiple drains and thus can be used for algebraic computation.

But even ADDs cannot represent discrete Boolean functions with a lot of different values efficiently. Another variant that tries to solve this problem are Edge-Valued BDDs (EVBDDs) [Lai 1992]. The idea behind this variant is that the addition of a constant value to a function f does not influence the structure of an EVBDD that represents f. This concept can be expanded to a data structure that is invariant under scalar multiplication and translation which leads to Normalized ADDs (NADDs). NADDs are structural equivalent to Factored Edge-Valued BDDs (FEVBDDs) [Tafertshofer 1997]. The main difference relies on how the functions are normalized. A detailed comparison can be found in the succeeding paper. Like EVBDDs and FEVBDDs NADDs do not store the values of a function inside the drains, the values will be calculated on the way to them.

In this thesis the ability of algebraic computation of different OBDD variants will be discussed and compared. The second and third chapter will give an introduction to Shared OBDDs (SOBDDs) and ADDs. One of the major achievements of this diploma thesis was the development of NADDs, which will be explained in Chapter 4. In Chapter 5 we will examine the differences and advantages of all introduced OBDD variants. The advantages of NADDs with respect to algebraic computations will also be shown there.

[1] similar to Multi-Terminal Binary Decision Diagrams (MTBDDs) [Clarke 1996]

2 Shared Binary Decision Diagrams

Binary Decision Diagrams (BDDs) can be used as a data structure for symbolic representation of discrete functions over Boolean variables. This chapter will give the definition of Shared Ordered Binary Decision Diagrams (SOBDDs) and will explain basic manipulations on them.

2.1. Notations and Definitions

Definition 2.1.1. Let $\mathcal{Z} = \{z_1, \ldots, z_n\}$ be a finite set of Boolean variables. An *evaluation* of $\mathcal{Z}$ is a map

$$\eta : \mathcal{Z} \to \{0, 1\}$$

that assigns a value $\eta(z) \in \{0, 1\}$ to each variable $z \in \mathcal{Z}$.
$\mathrm{Eval}(\mathcal{Z})$ identifies the set of all evaluations of $\mathcal{Z}$.

Let $\overline{a} = (a_1, \ldots, a_n) \in \{0, 1\}^n$ and $\overline{z} = (z_{i_1}, \ldots, z_{i_n}) \in \mathcal{Z}^n$ with pairwise different z_{i_j}, then $[\overline{z} = \overline{a}]$ represents the evaluation $\eta \in \mathrm{Eval}(\mathcal{Z})$ with

$$\eta(z_{i_j}) = a_j, \ j = 1, \ldots, n.$$

□

Notation 2.1.2. Let $\mathcal{Z}$ be defined as in Definition 2.1.1,
$\overline{b} = (b_1, \ldots, b_r) \in \{0, 1\}^r$ and $\overline{z} = (z_{i_1}, \ldots, z_{i_r}) \in \mathcal{Z}^r$ with pairwise different z_{i_j}.
The *assignment*

$$\eta \left[\overline{z} = \overline{b}\right] \in \mathrm{Eval}(\mathcal{Z})$$

is defined by

$$\eta \left[\overline{z} = \overline{b}\right](z) = \begin{cases} b_j & \text{if } z \in \{z_{i_1}, \ldots, z_{i_r}\} \text{ with } z = z_{i_j} \\ \eta(z) & \text{otherwise.} \end{cases}$$

Definition 2.1.3. Let $\mathbb{K}$ be a set. A $\mathbb{K}$-*function* over $\mathcal{Z}$ is a map

$$f : \mathrm{Eval}(\mathcal{Z}) \to \mathbb{K}.$$

The set of all Boolean functions over $\mathcal{Z} = \{z_1, \ldots, z_n\}$ will be called $\mathbb{K}(\mathcal{Z})$ or $\mathbb{K}(z_1, \ldots, z_n)$. The special case $\mathbb{K} = \{0, 1\}$ identifies the *switching functions*. The set of all switching functions will be called $\mathbb{B}(\mathcal{Z})$ or $\mathbb{B}(z_1, \ldots, z_n)$.

□

Definition 2.1.4. Let $\overline{z}$ and $\overline{b}$ be as in Notation 2.1.2 and $f \in \mathbb{K}(\mathcal{Z})$. The *cofactor* of f related to $\overline{z}$ is defined by:

$$f|_{\overline{z}=\overline{b}} \in \mathbb{K}(\mathcal{Z})$$

where

$$f|_{\overline{z}=\overline{b}}(\eta) = f(\eta\left[\overline{z} = \overline{b}\right]).$$

□

Definition 2.1.5. A *variable ordering* over $\mathcal{Z}$ is an ordered tuple

$$\pi = (z_{i_1}, \ldots, z_{i_n}),$$

that contains every variable $z_i \in \mathcal{Z}$ exactly once. A variable ordering π defines an order relation over variables in a canonical way. For every two variables $z_{i_j}, z_{i_k} \in \pi$ the following holds:

$$z_{i_j} <_\pi z_{i_k} \Leftrightarrow j < k.$$

□

2.2. Binary Decision Diagrams

Binary decision diagrams are a variant of decision trees. The reason why decision diagrams are a more compact representation for Boolean functions than decision trees, is that the functions of certain subtrees can coincide and therefore be represented by a single node of a BDD.

An example for this is shown in Figure 2.1. Each non-terminal node v is labeled with its variable name $var(v)$ and has successors directed towards its two children $succ_0(v)$ (shown as a dashed line) and $succ_1(v)$ (shown as a solid line) corresponding to the value assigned to the variable. Each drain d is labeled with its value $value(d)$.

BDDs are in practice an efficient data structure for Boolean function representation, but they are not efficient for every function. It will be shown that there is no efficient data structure for all Boolean functions.

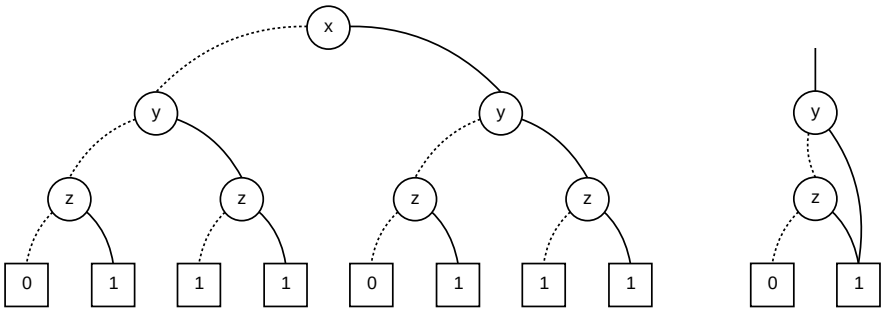


Figure 2.1: Decision Tree and a corresponding Binary Decision Diagram

Theorem 2.2.1. The number of switching functions over $\mathcal{Z}$ equals

$$|\mathbb{B}(\mathcal{Z})| = 2^{2^{|\mathcal{Z}|}}.$$

Proof. Exactly $|B|^{|A|}$ maps $A \to B$ exist for every two finite sets A, B. Let $B = \{0, 1\}$ then the following holds:

$$|\text{Eval}(\mathcal{Z})| = \text{number of maps } \mathcal{Z} \to \{0, 1\} = 2^{|\mathcal{Z}|}$$

and with this:

$$|\mathbb{B}(\mathcal{Z})| = \text{number of maps Eval}(\mathcal{Z}) \to \{0, 1\} = 2^{|\text{Eval}(\mathcal{Z})|} = 2^{2^{|\mathcal{Z}|}}$$

□

Theorem 2.2.2. For each universal data structure functions with a representation of exponential size exist.

Proof. Let C be the number of functions $f \in \mathbb{B}(z_1, \ldots, z_n)$ that can be represented with at most 2^{n-1} bits. Then it holds:

$$C \leq \sum_{i=0}^{2^{n-1}} 2^i = 2^{2^{n-1}+1} - 1 < 2^{2^{n-1}+1}.$$

This shows that at least

$$2^{2^n} - 2^{2^{n-1}+1} = 2^{2^{n-1}+1} \cdot (2^{2^n - 2^{n-1} - 1} - 1) = 2^{2^{n-1}+1} \cdot (2^{2^{n-1}-1} - 1)$$

functions need more than 2^{n-1} bits to be represented. □

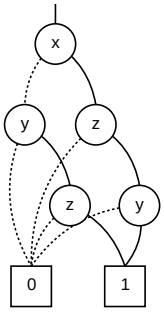


Figure 2.2: BDD violating the variable ordering condition

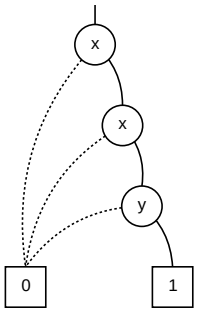


Figure 2.3: BDD violating the Read Once condition

2.3. Ordered Binary Decision Diagrams

The main concept for representing switching functions will be explained by means of BDDs. For the purpose of this thesis, only Bryant's Ordered BDDs (OBDDs) [Bryant 1986] are of relevance. These require a fixed variable ordering[1] π such that on every path the variables occur in the same order as in the variable ordering π.

Another important requirement for an OBDD, besides the variable ordering, is the Read Once condition. No variable may occur more than once on every possible path of an OBDD[2].

Definition 2.3.1. [**Ordered Binary Decision Diagram**]**:** Let π be a variable ordering of $\mathcal{Z}$. A π-OBDD is a tuple

$$\mathcal{B} = (V, V_I, V_T, succ_0, succ_1, var, value, v_0)$$

that contains:

- a finite set of nodes $V = V_I \cup V_T$ with $V_I \cap V_T = \emptyset$ (V_I contains the inner and V_T the terminal nodes),

[1] See Figure 2.2 for a BDD violating the variable ordering condition.
[2] See Figure 2.3 for a BDD that violates the Read Once condition.

- functions $succ_0$ and $succ_1$

  $$succ_0, succ_1 : V_I \to V$$

  that map every inner node to its successors,
- a function $var : V_I \to \mathcal{Z}$ that yields a labeling of the nodes with variables,
- a map $value : V_T \to \{0, 1\}$ that assigns a value to a terminal node and
- a root node[3]: $v_0 \in V$.

Additionally, for all $v \in V_I$ and $b \in \{0, 1\}$ the following must hold:

$$var(v) <_\pi var(succ_b(v)) \text{ if } succ_b(v) \in V_I.$$

□

Notation 2.3.2. In the following indices for the components of an OBDD are used when dealing with two or more OBDDs. For instance:

$$\mathcal{B}_i = (V_i, V_{I_i}, V_{T_i}, succ_{0_i}, succ_{1_i}, var_i, value_i, v_{0_i})$$

The complexity of algorithms on OBDDs is measured with respect to their size.

Definition 2.3.3. [**Size of an OBDD**]**:** The size of a π-OBDD $\mathcal{B}$ is defined by its number of nodes:

$$|\mathcal{B}| = |V|.$$

□

The following definition (*Shannon Expansion*) uses a bottom-up strategy to define the function represented by a given OBDD.

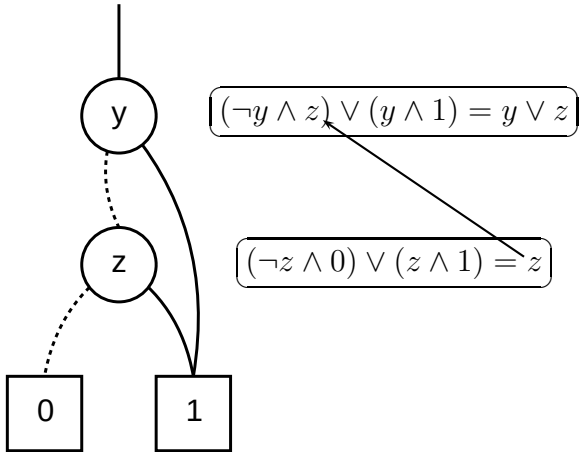
Definition 2.3.4. [**Function of an OBDD**]**:** Let $\mathcal{B}$ be an OBDD with a variable set $\mathcal{Z}$. A function $f_v \in \mathbb{B}(\mathcal{Z})$ will be assigned to every node v with

$$f_v = \begin{cases} value(v) & v \in V_T \\ (\neg z \wedge f_{succ_0(v)}) \vee (z \wedge f_{succ_1(v)}) & v \in V_I \text{ with } var(v) = z. \end{cases}$$

[3]Nodes that are not reachable from a root node do not belong to the OBDD. This holds for every further BDD variant.

□

Example 2.3.5. The following BDD represents the function $y \vee z$ which can be calculated with the Shannon Expansion in a bottom-up strategy.



■

With this definition it is possible to define algorithms to manipulate OBDDs. But a very important condition is missing to assure the uniqueness[4] of OBDD representations for a given switching function and a fixed variable ordering.

2.3.1. Reduced Ordered Binary Decision Diagrams

In Figure 2.1 the decision tree has redundant nodes while the corresponding Binary Decision Diagram has none. Note, that any decision tree could also be interpreted as an Ordered Binary Decision Diagram.

Definition 2.3.6. An OBDD $\mathcal{B}$ is called *reduced* if for every two nodes $v_1, v_2 \in V$ of $\mathcal{B}$ the following holds:

$$v_1 \neq v_2 \quad \Rightarrow \quad f_{v_1} \neq f_{v_2}$$

□

This definition ensures that there are no redundant nodes in a ROBDD. In the following it will be assumed that no OBDD has redundant drains. To obtain this, different drains with the same value have to be merged into one. With two *reduction rules* every such OBDD can be transferred into a ROBDD.

[4]Figure 2.1 shows two different OBDDs representing the same function.

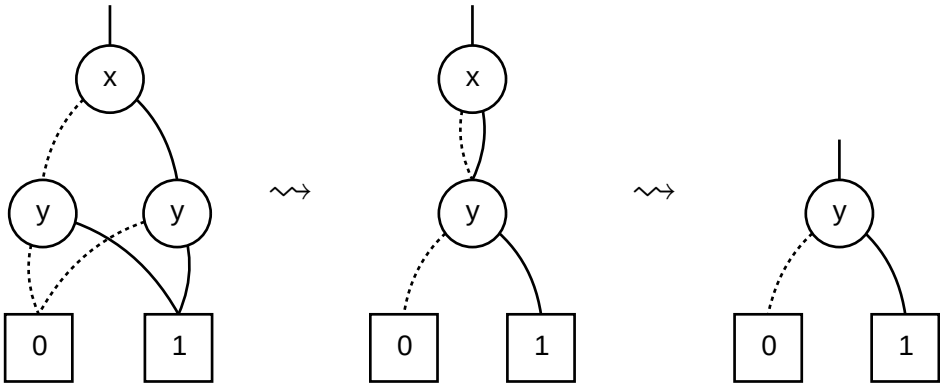


Figure 2.4: Reduction rules applied from bottom to top level

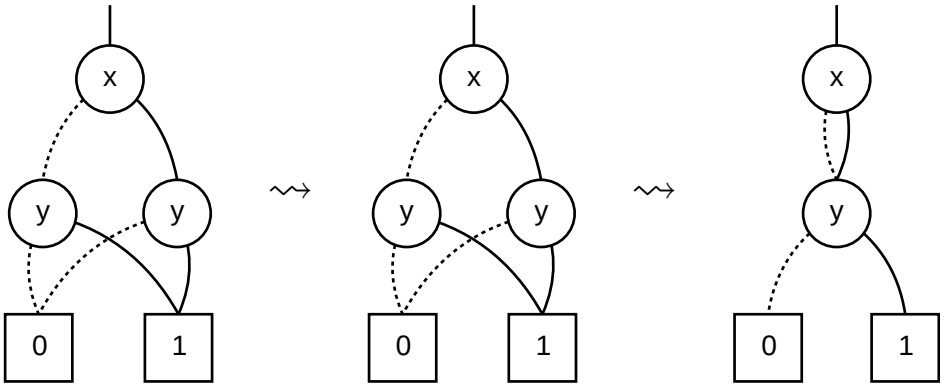


Figure 2.5: Reduction rules applied from top to bottom level

- **Elimination rule:** If for node $v \in V_I$ holds:

  $$succ_0(v) = succ_1(v).$$

  Eliminate v and redirect all incoming edges to $succ_0(v)$.

- **Isomorphism rule:** If for two nodes $v_1, v_2 \in V$ with $v_1 \neq v_2$ holds:

  $$succ_b(v_1) = succ_b(v_2)\ \forall b \in \{0, 1\}.$$

  Eliminate v_2 and redirect all incoming edges to v_1.

These rules should be applied from the bottom to the top level of the OBDD (see Figure 2.4). Otherwise the OBDD could still contain redundant nodes (see Figure 2.5). In this case more than one traversation is needed to obtain reducedness.

This observation is the reason why all algorithms compute their values from

the bottom to the top level. The reduction rules can be applied before creating redundant nodes. Thus, the reducedness of an OBDD can be ensured without additional traversation.

2.3.1.1. Binary Boolean Operators

Now it is feasible to define manipulations on (R)OBDDs. The following observation is the base of all OBDD-based binary Boolean operators.

Lemma 2.3.7. Let $f_1, f_2 \in \mathbb{B}(\mathcal{Z})$, $z \in \mathcal{Z}$ and **op** be a binary Boolean connector (e.g. conjunction, disjunction, implication,...). Then the following holds:

$$\boxed{f_1 \ \mathbf{op} \ f_2 = \big(\neg z \wedge (f_1|_{z=0} \ \mathbf{op} \ f_2|_{z=0})\big) \vee \big(z \wedge (f_1|_{z=1} \ \mathbf{op} \ f_2|_{z=1})\big)}$$

Proof. With the definition of cofactors it can directly be seen that for every $b \in \{0, 1\}$:

$$(f_1 \ \mathbf{op} \ f_2)|_{z=b} = f_1|_{z=b} \ \mathbf{op} \ f_2|_{z=b}.$$

For a z-node v, $f_v = f_1 \ \mathbf{op} \ f_2$ can be expressed with the Shannon Expansion:

$$\begin{aligned} f_v &= f_1 \ \mathbf{op} \ f_2 \\ &= (\neg z \wedge (f_1 \ \mathbf{op} \ f_2)|_{z=0}) \vee (z \wedge (f_1 \ \mathbf{op} \ f_2)|_{z=1}) \\ &= (\neg z \wedge (f_1|_{z=0} \ \mathbf{op} \ f_2|_{z=0})) \vee (z \wedge (f_1|_{z=1} \ \mathbf{op} \ f_2|_{z=1})) \end{aligned}$$

□

Lemma 2.3.7 can be used to describe an algorithm which applies a binary Boolean connector to two OBDDs, called APPLY. Algorithm 1 describes a version of the APPLY algorithm, which usually creates redundant nodes (see Figure 2.6).

To obtain a ROBDD, it is necessary to reduce the OBDD after applying an operator to two OBDDs. But with small modifications the algorithm will create reduced OBDDs (see Algorithm 2).

Note that Algorithm 2 still creates a ROBDD even if the input contains a non-reduced OBDD. But it is reasonable to apply this algorithm on ROBDDs only because the computation time depends on the size of the used input BDDs.

Lemma 2.3.8. For two ROBDDs $\mathcal{B}_1, \mathcal{B}_2$, a Boolean operator **op** has the complexity

$$\Theta(|\mathcal{B}_1| \cdot |\mathcal{B}_2|).$$

□

The integration of the elimination and isomorphic rule (as in Algorithm 2) can be used for any kind of algorithm on OBDDs. In the following, this modification will be used as the **find_or_add** function (see Algorithm 3).

Algorithm 1 APPLY(v_1, v_2,**op**)

Input: Node v_1 of a π-OBDD $\mathcal{B}_1$, node v_2 of a π-OBDD $\mathcal{B}_2$ and operator **op**
Output: Node v of a π-OBDD with $f_v = f_{v_1}$ **op** f_{v_2}

if $v_1 \in V_{T_1}$ and $v_2 \in V_{T_2}$ **then**
  $b \leftarrow value_1(v_1)$ **op** $value_2(v_2)$
  return $\boxed{b}$
else
  $z \leftarrow \min\{var_1(v_1), var_2(v_2)\}$
  $w_0 \leftarrow \text{APPLY}(v_1|_{z=0}, v_2|_{z=0}, \textbf{op})$
  $w_1 \leftarrow \text{APPLY}(v_1|_{z=1}, v_2|_{z=1}, \textbf{op})$
  $v \leftarrow$ new z-node v with $succ_0(v) = w_0$ and $succ_1(v) = w_1$
  return v
end if

Algorithm 2 APPLY(v_1, v_2,**op**) with modifications

Input: Node v_1 of a π-ROBDD $\mathcal{B}_1$, node v_2 of a π-ROBDD $\mathcal{B}_2$ and operator **op**
Output: Node v of a π-ROBDD with $f_v = f_{v_1}$ **op** f_{v_2}

if $v_1 \in V_{T_1}$ and $v_2 \in V_{T_2}$ **then**
  if $\exists d \in V_T$ with $value(d) = value_1(v_1)$ **op** $value_2(v_2)$ **then**
    return $\boxed{d}$
  else
    $b \leftarrow$ new drain with $value(b) = value_1(v_1)$ **op** $value_2(v_2)$
    return $\boxed{b}$
  end if
else
  $z \leftarrow \min\{var_1(v_1), var_2(v_2)\}$
  $w_0 \leftarrow \text{APPLY}(v_1|_{z=0}, v_2|_{z=0}, \textbf{op})$
  $w_1 \leftarrow \text{APPLY}(v_1|_{z=1}, v_2|_{z=1}, \textbf{op})$
  if $w_0 = w_1$ **then**
    return w_0
  else if $\exists v \in V_I$ with $var(v) = z$, $succ_0(v) = w_0$ and $succ_1(v) = w_1$ **then**
    return v
  else
    $v \leftarrow$ new z-node v with $succ_0(v) = w_0$ and $succ_1(v) = w_1$
    return v
  end if
end if

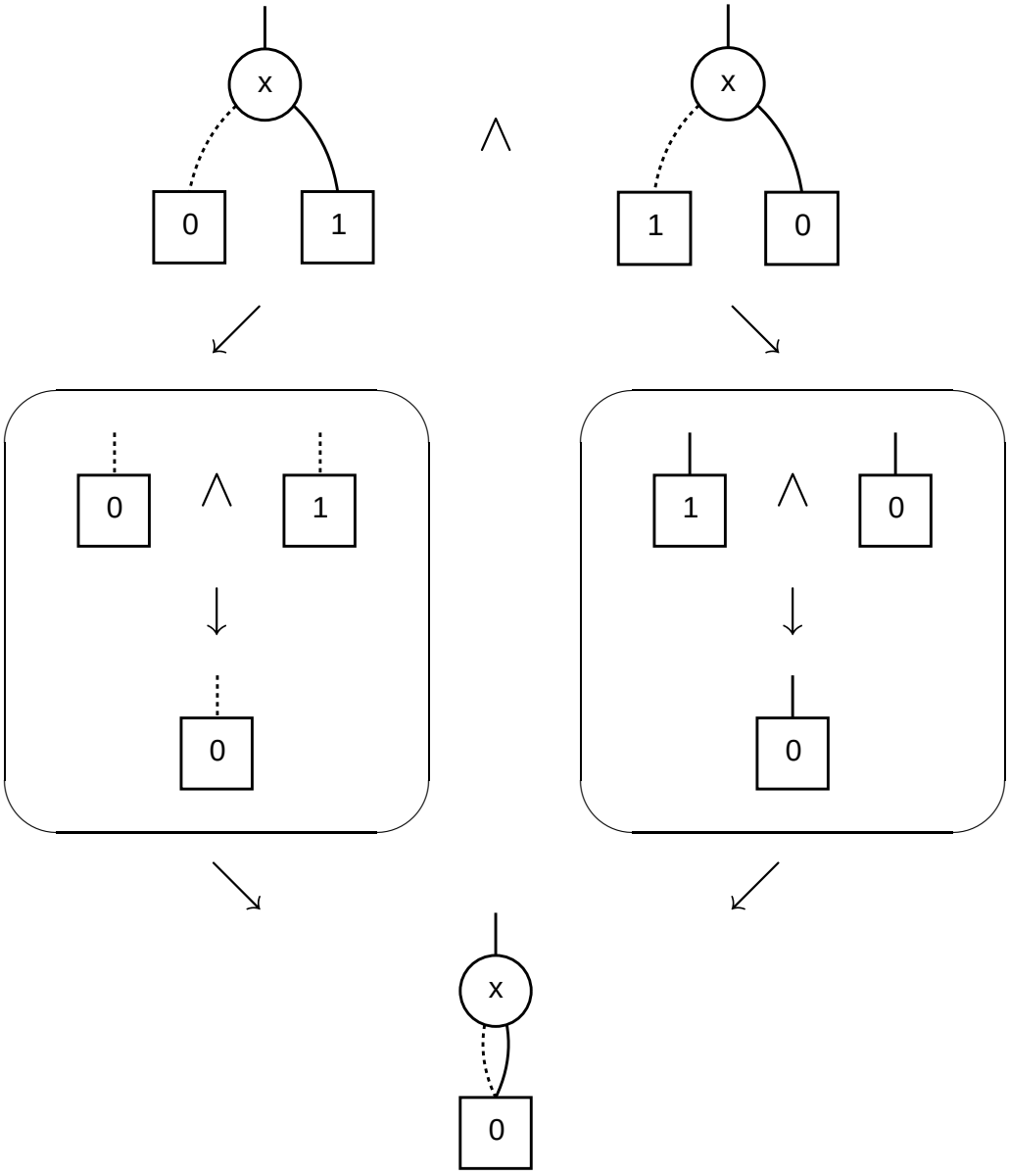


Figure 2.6: Conjunction of two ROBDD functions creates redundant nodes

Algorithm 3 find_or_add(z, w_0, w_1)

Input: Variable $z \in \mathcal{Z}$ and nodes w_0 and w_1 of a π-OBDD $\mathcal{B}$
Output: Node v of a π-ROBDD

if $w_0 = w_1$ **then**
  return w_0
else if $\exists v \in V_I$ with $var(v) = z$, $succ_0(v) = w_0$ and $succ_1(v) = w_1$ **then**
  return v
else
  $v \leftarrow$ new z-node v with $succ_0(v) = w_0$ and $succ_1(v) = w_1$
  return v
end if

2.3.1.2. Test for Equality

A very important operation on functions is the test for equality. This test can be performed by applying the operator $\leftrightarrow$ to the (R)OBDDs of two functions and verifying whether the result equals 1. Unfortenately, this test has the complexity $\Theta(|\mathcal{B}_1| \cdot |\mathcal{B}_2|)$, $\mathcal{B}_1$ and $\mathcal{B}_2$ being the corresponding OBDDs.

For two π-ROBDDs $\mathcal{B}_1$ and $\mathcal{B}_2$ with the same variable ordering $\pi = (z_1, \ldots, z_n)$ this test can be done according to Algorithm 4 in time

$$\Theta(\min\{|\mathcal{B}_1|, |\mathcal{B}_2|\}).$$

Algorithm 4 EQUAL(v_1, v_2)

Input: Node v_1 of a π-ROBDD $\mathcal{B}_1$, node v_2 of a π-ROBDD $\mathcal{B}_2$
Output: $f_{v_1} \leftrightarrow f_{v_2} \equiv 1$

if $v_1 \in V_{T_1}$ and $v_2 \in V_{T_2}$ **then**
  return $value_1(v_1) \leftrightarrow value_2(v_2)$
else if $v_1 \in V_{I_1}$ and $v_2 \in V_{I_2}$ **then**
  if $var_1(v_1) \neq var_2(v_2)$ **then**
    return false
  end if
  $z \leftarrow var_1(v_1)$
  return EQUAL$(v_1|_{z=0}, v_2|_{z=0}) \wedge$ EQUAL$(v_1|_{z=1}, v_2|_{z=1})$
else
  return false
end if

2.3.1.3. The Boolean Satisfiability Problem

The Boolean **Satisfiability** problem (SAT) is computationally a hard decision problem with many important applications in computer science. A propositional logical formula is said to be satisfiable if there is an assignment for its free variables that makes the formula true. SAT is NP-complete. This was the first problem known to be NP-complete proven by Stephen Cook [Cook 1971]. However, for a given ROBDD function this problem can be solved in constant time[5].

$$f \text{ is satisfiable} \Leftrightarrow f \not\equiv 0$$

2.3.1.4. Cofactors

Another important operation on a function is building cofactors. To build the cofactor $f|_{z=b}$ of a function f the value of the variable z is set to b.

Example 2.3.9. Let $f = z_0 \oplus \neg z_1$

$$\begin{aligned} f|_{z_0=0} &= ((\neg z_0 \wedge \neg z_1) \vee (z_0 \wedge z_1))|_{z_0=0} \\ &= (\neg 0 \wedge \neg z_1) \vee (0 \wedge z_1) \\ &= 1 \wedge \neg z_1 \\ &= \neg z_1 \end{aligned}$$

■

Algorithm 5 describes how to build a cofactor for a given OBDD $\mathcal{B}$ in the complexity $O(|\mathcal{B}|)$.

2.3.1.5. Negation

The negation of an OBDD function can be done in constant time. For this purpose the values of the drains have to be switched (see Figure 2.7).

2.3.1.6. Asymptotic Complexity

The complexity of operators on ROBDDs can be seen in Table 2.1. The test for equality is a very important operation on OBDDs. A very simple idea for improving ROBDDs so that this test can be done in constant time leads to SOBDDs, which will be described in detail later. The basic idea is to store more than one function in a reduced graph by sharing nodes for common cofactors instead of storing different ROBDDs.

[5]Notice that a ROBDD generated from a propositional logical formula can have exponential size (see Lemma 2.3.8).

Algorithm 5 COFACTOR(v, z, b)

Input: Node v of a π-ROBDD $\mathcal{B}$, variable $z \in \mathcal{Z}$ and value $b \in \{0, 1\}$
Output: Node w with $f_w = f|_{z=b}$

```
if v ∈ V_T then
  return v
else if z < var(v) then
  return v
else if z = var(v) then
  return succ_b(v)
else
  w_0 ← COFACTOR(succ_0(v), z, b)
  w_1 ← COFACTOR(succ_1(v), z, b)
  return find_or_add(z, w_0, w_1)
end if
```

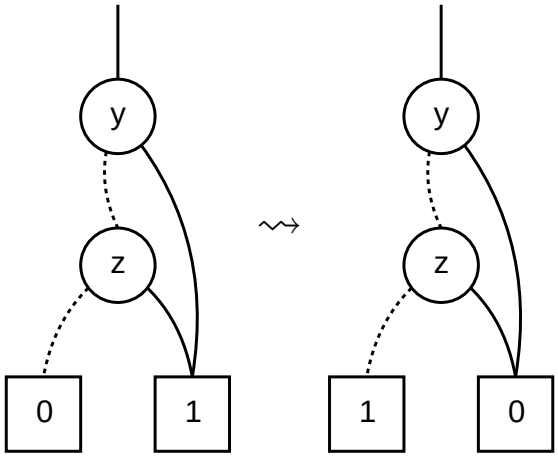


Figure 2.7: Before and after the application of the negating function

Operator on ROBDDs	Asymptotic Complexity
APPLY$(\mathcal{B}_1, \mathcal{B}_2, \mathbf{op})$	$\lvert\mathcal{B}_1\rvert \cdot \lvert\mathcal{B}_2\rvert$
EQUAL$(\mathcal{B}_1, \mathcal{B}_2)$	$\min\{\lvert\mathcal{B}_1\rvert, \lvert\mathcal{B}_2\rvert\}$
$\mathcal{B}\lvert_{z=b}$	$\lvert\mathcal{B}\rvert$
$\neg\mathcal{B}$	constant

Table 2.1: Complexity of operators on ROBDDs

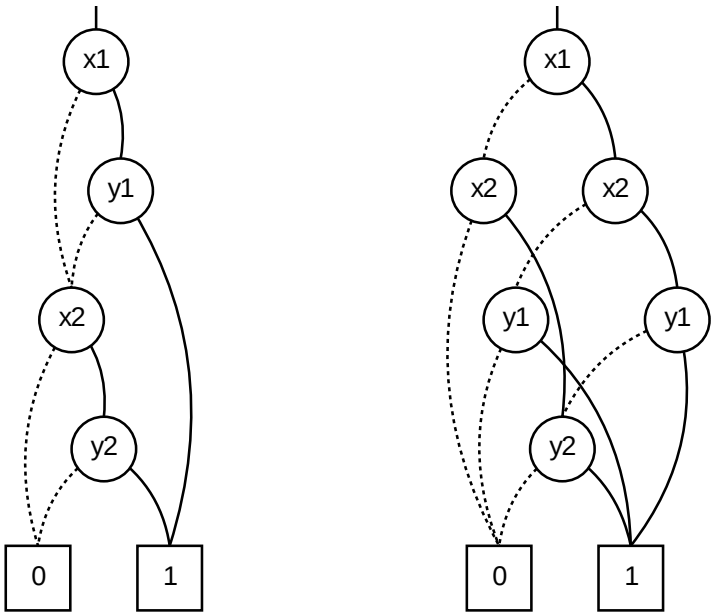


Figure 2.8: The same function represented by ROBDDs with different variable orderings

2.3.1.7. The Variable Ordering Problem

The APPLY algorithm (Algorithm 2) requires as input two OBDDs with the same variable ordering. It is obvious that manipulating OBDDs with arbitrary variable orderings is at least as hard as manipulating OBDDs with the same ordering.

The representation of a function with a ROBDD and given variable ordering π is optimal under all π-OBDDs but the variable ordering has a great influence on the size of an OBDD.

Figure 2.8 shows the influence of different variable orderings on the function $f(x_1, y_1, x_2, y_2) = (x_1 \wedge y_1) \vee (x_2 \wedge y_2)$. With the ordering $\pi = (x_1, y_1, x_2, y_2)$ the ROBDD on the left has six nodes[6] while the same function represented by the ROBDD on the right (with the ordering $\pi' = (x_1, x_2, y_1, y_2)$) has eight nodes.

Functions can be classified by their behaviour in reference to different variable orderings.

In Table 2.2 three different function classes can be seen. Unfortunately, the functions whose best OBDD representation has exponential size are not only of theoretical nature. They appear quite often in arithmetic computations

[6]drains are counted

Function Class	Complexity	
	Best	Worst
Symmetric	linear	quadratic
Integer Addition	linear	exponential
Integer Multiplication (middle bits)	exponential	exponential

Table 2.2: Complexity for different function classes

with OBDDs[7]. More detailed information about the complexity of integer addition and multiplication can be found in [Bryant 1991]. Theorem 2.2.2 implies that with each universal data structure for switching functions, such as ROBDDs, there are ill-natured functions[8]. Fortunately, there are also good-natured functions where the ROBDD size is invariant under different variable orderings.

Definition 2.3.10. A function $f \in \mathbb{B}(z_1, \ldots, z_n)$ is called totally symmetric if its values are invariant under all permutations $\sigma \in S_n$ of all assignments $\overline{a} \in \{0,1\}^n$, i.e. :

$$f([\overline{z} = \overline{a}]) = f([\overline{z} = \sigma(\overline{a})]).$$

□

Example 2.3.11. The parity function $f = \bigoplus_{1 \leq i \leq n} z_i$ is totally symmetric.

$$\begin{aligned} f\,[\overline{z} = \overline{a}] &= \bigoplus_{1 \leq i \leq n} a_i \\ &= a_1 \oplus \ldots \oplus a_n \\ &= a_{\pi(1)} \oplus \ldots \oplus a_{\pi(n)} \\ &= \bigoplus_{1 \leq i \leq n} a_{\pi(i)} \\ &= f\,[\overline{z} = \pi(\overline{a})] \end{aligned}$$

■

Lemma 2.3.12. Every ROBDD representation of a totally symmetric function $f \in \mathbb{B}(z_1, \ldots, z_n)$ has

$$O(n^2)$$

nodes.

[7] How OBDDs can be used for integer arithmetics will be shown in Section 2.4.

[8] These are functions with exponential sized ROBDDs with respect to any variable ordering

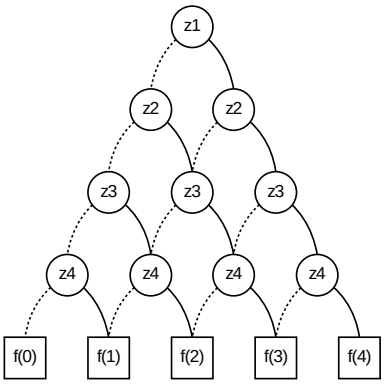


Figure 2.9: Construction of an OBDD representing a symmetric function

Proof. The values of a totally symmetric function are defined by the number of zeros and ones assigned to its variables. From this point of view a symmetric function $f : \{0,1\}^n \to \{0,1\}$ can also be seen as a function

$$f : \{0, \ldots, n\} \to \{0,1\}.$$

For any ordering π the π-OBDD representation of a symmetric function can be constructed as indicated in Figure 2.9 for four variables.

From this example it can be seen that level i contains at most i z_i-nodes. For n variables the number of nodes is bounded above by:

$$\begin{aligned} \underbrace{(n+1)}_{\text{number of drains}} + \sum_{i=1}^{n} i &= (n+1) + \frac{n \cdot (n+1)}{2} \\ &= \frac{(n+2) \cdot (n+1)}{2} \\ &= \frac{n^2 + 3n + 2}{2} \end{aligned}$$

□

The question if several functions have polynomial-sized ROBDDs for a common variable ordering cannot be answered efficiently. It is already very hard to find an optimal variable ordering[9] for one OBDD.

2.3.2. Shared Ordered Binary Decision Diagrams

Although the representation of two or more switching functions by separate (node disjoint) ROBDDs with appropriate variable orderings might be more

[9] An algorithm for the exact minimization can be found in [Friedman 1990]. The proof of the complexity for OBDDs and SOBDDs can be read in [Bollig 1996] and [Tani 1993].

compact than the representation in a single reduced decision graph, the manipulation of ROBDDs with different orderings is known to be computationally hard. For most Boolean connectives the best known algorithms rely on a transformation of the given ROBDDs into equivalent ROBDDs with the same ordering. For this reason the simultaneous representation of several switching functions in one Shared OBDD (SOBDD) turned out to be the most appropriate approach.

Definition 2.3.13. [**Shared Ordered Binary Decision Diagram**]**:** Let π be a variable ordering of $\mathcal{Z}$. A π-SOBDD is a tuple

$$\overline{\mathcal{B}} = (V, V_I, V_T, succ_0, succ_1, var, value, \overline{v})$$

that contains:

- a finite set of nodes $V = V_I \cup V_T$ with $V_I \cap V_T = \emptyset$ (V_I contains the inner and V_T the terminal nodes),
- functions $succ_0$ and $succ_1$

  $$succ_0, succ_1 : V_I \to V$$

  that map every inner node to its successors,
- a function $var : V_I \to \mathcal{Z}$ that yields a labeling of the nodes with variables,
- a map $value : V_T \to \{0, 1\}$ assigns a value to a terminal node and
- a tuple of root nodes: $\overline{v} = (v_1, \ldots, v_k) \in V^k$.

Additionally, for all $v \in V_I$ and $b \in \{0, 1\}$ the following, must hold:

$$var(v) <_\pi var(succ_b(v)) \text{ if } succ_b \in V_I$$

In the following SOBDDs are assumed to be reduced, i.e.

$$f_v \neq f_w \text{ if } v \neq w.$$

□

In the sequel, we shall discuss the complexity of several composition operators for ROBDDs represented as nodes of a SOBDD. In this context, we will use the following definition.

Definition 2.3.14. [**Size of a function**]**:** Let f be a switching function. The size of f with regard to variable ordering π is defined by

$$|f|_\pi = |\mathcal{B}|$$

$\mathcal{B}$ being the π-ROBDD with $f_{v_0} = f$. If π is understood from the context then the notation $|f|$ instead of $|f|_\pi$ will be used.

□

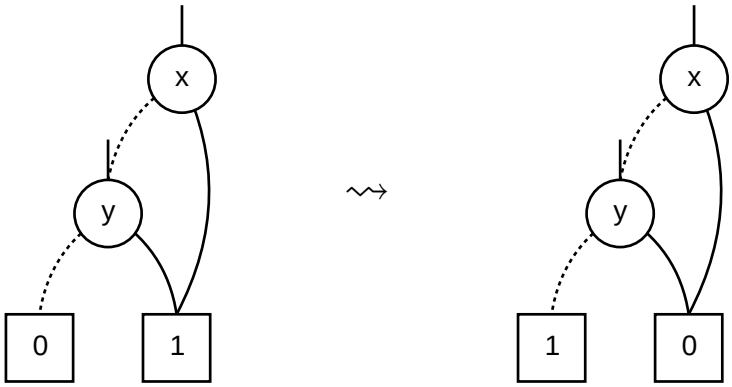


Figure 2.10: Switching the drains influences all functions

2.3.2.1. Test for Equality

Two functions are equal if they are represented by the same graph. Thus, the test for equality can be performed in constant time[10] while for ROBDDs an algorithm needs to traverse the whole diagram (see Algorithm 4).

2.3.2.2. Negation

The negation of a function in a SOBDD cannot be performed as in ROBDDs because switching the values of the two drains would influence all represented functions in the SOBDD (see Figure 2.10).

A simple way of negating a function f is based on APPLY:

$$\begin{aligned} 1 \oplus f &\equiv (\neg 1 \wedge f) \vee (1 \wedge \neg f) \\ &\equiv 1 \wedge \neg f \\ &\equiv \neg f \end{aligned}$$

An explicit formulation for APPLY$(1, f, \oplus)$ is shown in Algorithm 6. APPLY$(1, f, \oplus)$ and hence NEGATE(f), have the complexity $O(|f|)$.

2.3.2.3. Attributed Edges

The complexity of several operators on SOBDDs is shown in Table 2.3. As mentioned in Section 2.3.1.6 one of the advantages of SOBDDs is that the test for equality can be performed in constant time but the downside of SOBDDs is that negation cannot be carried out in constant time. To avoid this drawback *attributed edges* can be used.

[10] with the comparison of the root nodes

Algorithm 6 NEGATE(v)

Input: z-node v of a π-SOBDD $\mathcal{B}$
Output: $\neg f_v$

```
if v ∈ V_T then
  return ¬value(v)
else
  w_0 ← NEGATE(v|_{z=0})
  w_1 ← NEGATE(v|_{z=1})

  return find_or_add(z, w_0, w_1)
end if
```

Operator on SOBDDs	Asymptotic Complexity
APPLY($f_1, f_2, \mathbf{op}$)	$\|f_1\| \cdot \|f_2\|$
EQUAL(f_1, f_2)	**constant**
$f\|_{z=b}$	$\|f\|$
$\neg f$	$\|f\|$

Table 2.3: Complexity of operators on SOBDDs

Using attributed edges means information is stored on the edges of a node that influences the semantics of the represented function. In this case, the negation of a function is the information stored on the edges of a node. This type of attributed edges is called *negative edges* below.

A lot of other kinds of attributed edges for SOBDDs are possible[11].

2.3.3. SOBDD with Negative Edges

The most important thing to take care about when dealing with attributed edges is that uniqueness can easily be lost. In the case of SOBDDs this would cost the advantage of testing for equality in constant time. Therefore two restrictions will be used:

- only the one drain[12] exists
- an attribute can only be attached to the edge from f to $f|_{z=0}$ and to the root nodes $v_0 \in \overline{v_0}$

[11] See [Minato et al 1990] for more details.
[12] $f_{\text{one drain}} \equiv 1$

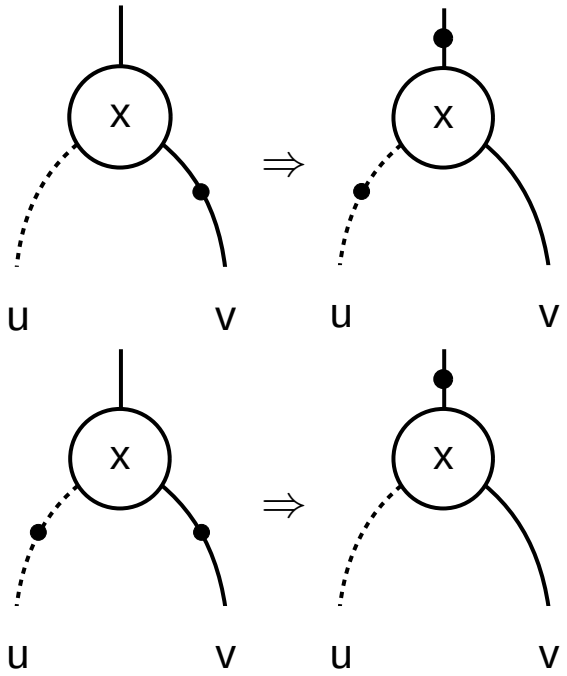


Figure 2.11: Transformation rules for negative edges

When reformulating the APPLY algorithm and other algorithms for SOBDDs with negative edges one has to take care about these two conditions. Figure 2.11 shows how situations in which the conditions are violated can be resolved.

Definition 2.3.15. [**SOBDD with Negative Edges**]**:** Let π be a variable ordering of $\mathcal{Z}$. A π-SOBDD with negative edges is a tuple

$$\overset{\circ}{\mathcal{B}} = (V, V_I, V_T, neg, succ_0, succ_1, var, \overline{\langle v, neg \rangle})$$

that contains[13]:

- a finite set of nodes $V = V_I \cup V_T$ with $V_I \cap V_T = \emptyset$ (V_I contains the inner and V_T the one drain),
- functions $succ_0$ and $succ_1$

  $$succ_0, succ_1 : V_I \to V$$

  that map every inner node to its successors,
- a map

  $$neg : V_I \to \{0, 1\}$$

[13]The map $value : V_T \to \{0, 1\}$ is not needed anymore because there is only the one drain.

- a function $var : V_I \to \mathcal{Z}$ that yields a labeling of the nodes with variables,
- a tuple of root node pairs:

$$\overline{\langle v, neg \rangle} = (\langle v_1, neg_1 \rangle, \ldots, \langle v_k, neg_k \rangle) \in (V \times \{0,1\})^k$$

Additionally, for all $v \in V_I$ and $b \in \{0,1\}$ the following must hold:

$$var(v) <_\pi var(succ_b(v)) \text{ if } succ_b \in V_I.$$

The reducedness as formalized in Definition 2.3.18 is also required.

□

The semantics of a SOBDD with negative edges is slightly different to the semantics of SOBDDs.

Definition 2.3.16. [**Function of a SOBDD with negative edges**]**:** Let $\overrightarrow{\mathcal{B}}$ be an π-SOBDD with negative edges and variable set $\mathcal{Z}$. Every node v will be assigned to a function $f_v \in \mathbb{B}(\mathcal{Z})$ with

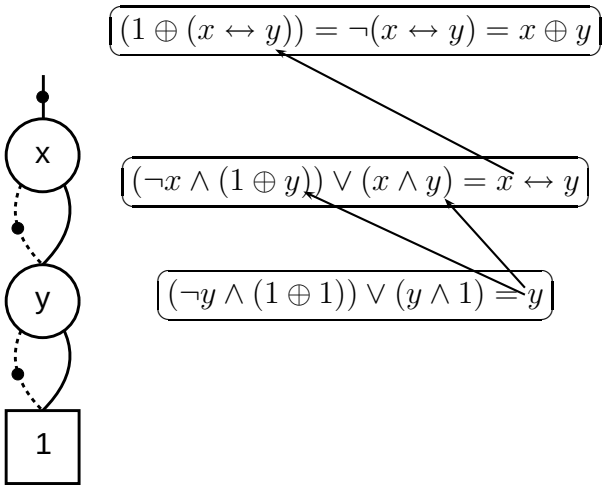
$$f_v = \begin{cases} 1 & v \in V_T \\ \left(\neg z \wedge (neg(v) \oplus f_{succ_0(v)})\right) \vee \left(z \wedge f_{succ_1(v)}\right) & v \in V_I \text{ with } var(v) = z. \end{cases}$$

Every root node tuple $\langle v_i, neg_i \rangle$ will be assigned to

$$f_{\langle v_i, neg_i \rangle} = neg_i \oplus f_{v_i}$$

□

Example 2.3.17.



■

The term reducedness is slightly different for this OBDD variant.

Definition 2.3.18. A π-SOBDD $\overrightarrow{\mathcal{B}}$ with negative edges is called reduced if for all nodes $v_1, v_2 \in V$ with $v_1 \neq v_2$ holds:

$$v_1 \neq v_2 \Rightarrow f_{v_1} \neq f_{v_2} \text{ and } f_{v_1} \neq \neg f_{v_2}.$$

□

The reduction rules have to be reformulated. The uniqueness of SOBDDs with negative edges is guaranteed because there is only the one drain. This ensures that negation is used the right way.

2.3.3.1. Algorithms on SOBDDs with negative edges

Algorithms for SOBDDs with negative edges can be described in a similar way to those on SOBDDs. For this purpose, the **find_or_add** function has to be modified (see Algorithm 7).

Algorithm 7 find_or_add$(z, \langle w_0, neg_0\rangle, \langle w_1, neg_1\rangle)$ on SOBDDs with negative edges

Input:	Variable $z \in \mathcal{Z}$ and tuples $\langle w_0, neg_0\rangle$ and $\langle w_1, neg_1\rangle$ with nodes w_0 and w_1 of a π-SOBDD $\overrightarrow{\mathcal{B}}$
Output:	Tuple $\langle v, neg\rangle$ with node $v \in V$ and $neg \in \{0, 1\}$

```
if ⟨w_0, neg_0⟩ = ⟨w_1, neg_1⟩ then
  return ⟨w_0, neg_0⟩
else if ∃v ∈ V_I with succ_b(v) = w_b and neg(v) = neg_0 ⊕ neg_1 then
  return ⟨v, neg_1⟩
else
  v ← z-node v with succ_b(v) = w_b and neg(v) = neg_0 ⊕ neg_1
  return ⟨v, neg_1⟩
end if
```

Now it is possible to rewrite every algorithm on SOBDDs to this OBDD variant. Algorithm 8 shows how this can be done for the APPLY algorithm.

2.3.3.2. Negation

The negative edges were introduced to increase the efficiency of SOBDDs. With the negative edges the negation of a SOBDD function can be performed in constant time.

Algorithm 8 APPLY($\langle v_1, neg_1\rangle, \langle v_2, neg_2\rangle$,**op**) on SOBDDs with negative edges

Input: Tuples $\langle v_1, neg_1\rangle$ and $\langle v_2, neg_2\rangle$ of a π-SOBDD $\overrightarrow{\mathcal{B}}$ and operator **op**

Output: Tuple $\langle v, neg\rangle$ of a π-SOBDD with $f_{\langle v,neg\rangle} = f_{\langle v_1,neg_1\rangle}$ **op** $f_{\langle v_2,neg_2\rangle}$

if $v_1 \in V_T$ and $v_2 \in V_T$ **then**
  $d \leftarrow \neg neg_1$ **op** $\neg neg_2$
  return $\langle v_1, \neg d\rangle$
else
  $z \leftarrow \min\{var_1(v_1), var_2(v_2)\}$

  $\left\langle s_{0_1}, neg_{s_{0_1}}\right\rangle \leftarrow \langle succ_0(v_1), neg_1 \oplus neg(v_1)\rangle$
  $\left\langle s_{0_2}, neg_{s_{0_2}}\right\rangle \leftarrow \langle succ_0(v_2), neg_2 \oplus neg(v_2)\rangle$
  $\langle w_0, neg_{w_0}\rangle \leftarrow$ APPLY($\left\langle s_{0_1}, neg_{s_{0_1}}\right\rangle, \left\langle s_{0_2}, neg_{s_{0_2}}\right\rangle$, **op**)

  $\left\langle s_{1_1}, neg_{s_{1_1}}\right\rangle \leftarrow \langle succ_1(v_1), neg_1\rangle$
  $\left\langle s_{1_2}, neg_{s_{1_2}}\right\rangle \leftarrow \langle succ_1(v_2), neg_2\rangle$
  $\langle w_1, neg_{w_1}\rangle \leftarrow$ APPLY($\left\langle s_{1_1}, neg_{s_{1_1}}\right\rangle, \left\langle s_{1_2}, neg_{s_{1_2}}\right\rangle$, **op**)

  return **find_or_add**($z, \langle w_0, neg_{w_0}\rangle, \langle w_1, neg_{w_1}\rangle$)
end if

Operator	Asymptotic Complexity for		
	ROBDDs	**SOBDDs**	**SOBDDs with n. e.**
APPLY	$\|f_1\| \cdot \|f_2\|$	$\|f_1\| \cdot \|f_2\|$	$\|f_1\| \cdot \|f_2\|$
EQUAL	$\min\{\|f_1\|, \|f_2\|\}$	constant	**constant**
COFACTOR	$\|f\|$	$\|f\|$	$\|f\|$
NEGATION	constant	$\|f\|$	**constant**

Table 2.4: Complexity of operators on different OBDD variants

2.3.3.3. Asymptotic Complexity

Table 2.4 shows the asymptotic complexity of all discussed OBDD variants. SOBDDs with negative edges combine the advantages of ROBDDs and SOBDDs.

In all following results SOBDDs with negative edges will be termed SOBDDs.

2.4. Algebraic Computation

Symbolic and Algebraic Computation (SAP), also known as Computer Algebra (CA), tries to automate mathematical computations of all sorts. Every problem that can be expressed as computations on Boolean functions can be represented as manipulations on OBDD functions.

2.4.1. Integer Computation

Every function $f \in \mathbb{K}(\mathcal{Z})$

$$f : \mathrm{Eval}(\mathcal{Z}) \to \mathbb{K}$$

can be seen as a function f':

$$f' : \mathrm{Eval}(\mathcal{Z}) \to \{0, \ldots, |\mathrm{Image}(f)| - 1\}.$$

An unsigned integer value $n > 0$ is encoded with its binary coding:

$$n = \sum_{i=0}^{\lfloor \log_2(n) \rfloor} a_i \cdot 2^i$$

with coefficients $a_i \in \{0, 1\}$.

For different function values there are different coefficients a_i. The different values of a coefficient a_i can be represented by an OBDD function[14] f_i. Thereby

[14] the function represented by an OBDD

a tuple $\overline{f} = (f_0, \ldots, f_k) \in \mathbb{B}(\mathcal{Z})^{k+1}$ of SOBDD functions[15] can be interpreted as a function $\overline{f}$

$$\overline{f} : \text{Eval}(z_1, \ldots, z_n) \rightarrow \{0, \ldots, 2^{k+1} - 1\}$$

with

$$\overline{f}(z_1, \ldots, z_n) = \sum_{i=0}^{k} f_i(z_1, \ldots, z_n) \cdot 2^i.$$

Example 2.4.1. Let $f(x) = 3x + 5$ then f_0, f_1, f_2, f_3 are defined by:

$$\begin{aligned} f_0(x) &= \neg x \\ f_1(x) &= 0 \\ f_2(x) &= \neg x \\ f_3(x) &= x \end{aligned}$$

To verify the result two different cases have to be considered.

$$\begin{aligned} f(0) &= f_0(0) \cdot 1 + f_1(0) \cdot 2 + f_2(0) \cdot 4 + f_3(0) \cdot 8 \\ &= 1 \cdot 1 + 0 \cdot 2 + 1 \cdot 4 + 0 \cdot 8 \\ &= 5 \end{aligned}$$

$$\begin{aligned} f(1) &= f_0(1) \cdot 1 + f_1(1) \cdot 2 + f_2(1) \cdot 4 + f_3(1) \cdot 8 \\ &= 0 \cdot 1 + 0 \cdot 2 + 0 \cdot 4 + 1 \cdot 8 \\ &= 8 \end{aligned}$$

■

With this encoding all basic arithmetic operations can be represented by SOBDDs.

2.4.1.1. Benchmark

To provide a comparison between different OBDD variants the OBDD sizes to represent certain functions will be compared.

[15] It can also be represented by a tuple of ROBDD functions but this would only make sense if different good variable orderings are known for the functions.

Let $\overline{x} = (x_0, \ldots, x_k)$ and $\overline{y} = (y_0, \ldots, y_k)$. The functions that serve as benchmarks are:

$$\begin{aligned} f_{\overline{x}} &= \sum_{i=0}^{k} x_i \cdot 2^i \\ f_{\overline{x}} + f_{\overline{y}} &= \left(\sum_{i=0}^{k} x_i \cdot 2^i\right) + \left(\sum_{i=0}^{k} y_i \cdot 2^i\right) \\ f_{\overline{x}} \cdot f_{\overline{y}} &= \left(\sum_{i=0}^{k} x_i \cdot 2^i\right) \cdot \left(\sum_{i=0}^{k} y_i \cdot 2^i\right) \end{aligned}$$

Table 2.5 shows the sizes of the benchmark functions represented by a SOBDD for different k. The π-SOBDDs for $f_{\overline{x}}$ and $f_{\overline{x}} + f_{\overline{y}}$ have polynomial size, while the π-SOBDD for the multiplication $f_{\overline{x}} \cdot f_{\overline{y}}$ has exponential size. For all functions the variable ordering

$$\pi = (x_k, y_k, \ldots, x_0, y_0)$$

was used. This is the optimal variable ordering for the addition ($f_{\overline{x}} + f_{\overline{y}}$). This can be seen if the function $f_{\overline{x}} + f_{\overline{y}}$ is split-up to its functions for every binary digit. Let f_i be the function that represents the i-th bit of $f_{\overline{x}} + f_{\overline{y}}$ and c_i the function that represents the add carry bit of function f_i. The function f_i can be calculated with the variables x_i and y_i and the add carry bit c_{i-1}, where $c_{-1} = 0$:

$$f_i = x_i \oplus y_i \oplus c_{i-1}.$$

The add carry bit c_i is calculated by:

$$c_i = \big(x_{i-1} \wedge y_{i-1}\big) \vee \big(c_{i-1} \wedge (x_{i-1} \vee y_{i-1})\big).$$

To calculate the function f_i the add carry bit functions c_j ($j = -1, \ldots, i-1$) are used. This can be seen when the functions are stored in a SOBDD these functions are shared for every further computation. This explains why the variable ordering

$$\pi' = (x_0, y_0, \ldots, x_k, y_k)$$

is not as good as the chosen one.

The multiplication has exponential size for every variable ordering. Empirical results show that the ordering π is well capable for this function.

2.4.2. Matrix Representation

SOBDDs can also be used for matrix representation. Therefore the entries of a matrix are encoded by Boolean variables x_i (for the rows) and y_j (for the columns).

k	SOBDD Nodes		
	$\lvert f_{\overline{x}} \rvert$	$\lvert f_{\overline{x}} + f_{\overline{y}} \rvert$	$\lvert f_{\overline{x}} \cdot f_{\overline{y}} \rvert$
0	2	4	3
1	3	9	12
2	4	14	45
3	5	19	153
4	6	24	475
5	7	29	1511
6	8	34	4674
7	9	39	14558
8	10	44	45054
9	11	49	139404
10	12	54	429911

Table 2.5: Size of the benchmark functions represented by a SOBDD

Example 2.4.2. The matrix

$$\begin{pmatrix} 1 & 1 & & \\ 1 & 1 & & \\ & & 3 & 3 \\ & & 3 & 3 \end{pmatrix}$$

can be expressed as truth tables

x_1	y_1	x_0	y_0	$f_0(\cdot)$
0	0	0	0	1
0	0	0	1	1
0	0	1	0	1
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	0
1	1	0	0	1
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1

x_1	y_1	x_0	y_0	$f_1(\cdot)$
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	0
1	1	0	0	1
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1

which are represented by the functions

$$f_0(x_1, y_1, x_0, y_0) = x_1 \leftrightarrow y_1$$
$$f_1(x_1, y_1, x_0, y_0) = x_1 \wedge y_1$$

■

Computations on matrices represented by SOBDDs cannot be reduced to basic algebraic operations. The variables do not code the values[16], they code the rows and columns of a matrix – the values are coded by the functions f_i. Thus SOBDDs are not a handy data structure for matrix operations. ADDs and NADDs, which are introduced later, are better suitable for that task.

[16] which is the case for the integer representation

3 Algebraic Binary Decision Diagrams

SOBDDs were introduced to represent functions over the domain $\{0, 1\}$. It was shown that it is also possible to represent functions over small finite domains. In computer algebra a lot of problems are expressed as matrix computations but as mentioned in Section 2.4.2 SOBDDs were shown to be not suitable for matrix manipulation. For this reason Algebraic Binary Decision Diagrams (ADDs) were developed, which will be introduced in this chapter.

3.1. Syntax and Semantics

ADDs are similar to SOBDDs without negative edges, the main difference being that ADDs can contain drains with arbitrary values.

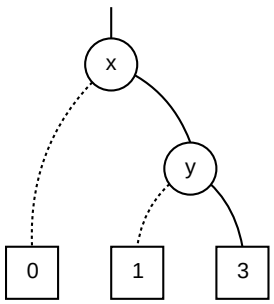
Definition 3.1.1. [**Algebraic Decision Diagram**]**:** Let π be a variable ordering of $\mathcal{Z}$. A π-ADD is as a π-SOBDD with one modification:

$$value : V_T \to \mathbb{R}.$$

□

The values of the ADD function are evaluated in the same way as in regular SOBDDs.

Example 3.1.2.



$$f(x,y) = \begin{cases} 0 & \text{for } \neg x \\ 1 & \text{for } x \wedge \neg y \\ 3 & \text{for } x \wedge y \end{cases}$$

■

Definition 3.1.3. [**Function of an ADD**]**:** Let $\mathcal{A}$ be an ADD with variable set $\mathcal{Z}$. A function $f_v \in \mathbb{B}(\mathcal{Z})$ is assigned to every node v:

$$f_v = \begin{cases} value(v) & v \in V_T \\ (1-z) \cdot f_{succ_0(v)} + z \cdot f_{succ_1(v)} & v \in V_I \text{ with } var(v) = z. \end{cases}$$

With that definition the function represented in Example 3.1.2 can be calculated as follows:

$$\begin{aligned} f(x,y) &= (1-x) \cdot f|_{x=0} + x \cdot f|_{x=1} \\ &= x \cdot \Big((1-y) \cdot f|_{[x=1,y=0]+y \cdot f|_{[x=1,y=1]}} \Big) + (1-x) \cdot 0 \\ &= x \cdot (y \cdot 3 + (1-y) \cdot 1) \\ &= x \cdot (2y+1) \end{aligned}$$

Theorem 3.1.4. [**ADDs are universal**]**:** For every function $f \in \mathbb{R}(\mathcal{Z})$ and every variable ordering π there exists a π-ADD root node $v \in \overline{v}$ with $f_v = f$.

Proof. The decision tree for f with respect to the variable ordering π is at the same time a π-ADD. □

The term reducedness can be used from OBDDs without modification. The reduction rules can be applied the same way.

In the following a π-ADD is assumed to be reduced.

3.2. Algorithms

Algorithms on ADDs are based on the same observation as described in Lemma 2.3.7. To create reduced ADDs the **find_or_add** function for SOBDDs can be applied without modifications.
Another important operation is the **find_or_add_drain** function which avoids the generation of redundant drains.

The algorithm for the multiplication will be described in detail. All other algorithms can be formulated analogously. Lemma 2.3.7 builds the base for all binary Boolean connectors for OBDDs. This can be expanded to any kind of algebraic operator.

Lemma 3.2.1. Let $f_1, f_2 \in \mathbb{R}(\mathcal{Z})$, $z \in \mathcal{Z}$ and **op** be an algebraic operator (e.g. addition, multiplication, maximum, ...). Then the following holds:

$$\boxed{f_1 \textbf{ op } f_2 = ((1-z) \cdot (f_1|_{z=0} \textbf{ op } f_2|_{z=0})) + (z \cdot (f_1|_{z=1} \textbf{ op } f_2|_{z=1}))}$$

Proof. The proof can be made with the observation from Lemma 2.3.7. For every $b \in \{0, 1\}$ holds:

$$(f_1 \textbf{ op } f_2)|_{z=b} = f_1|_{z=b} \textbf{ op } f_2|_{z=b}.$$

A z-node v with $f_v = f_1 \textbf{ op } f_2$ can be expressed through

$$\begin{aligned} f_v &= f_1 \textbf{ op } f_2 \\ &= ((1-z) \cdot (f_1 \textbf{ op } f_2)|_{z=0}) + (z \cdot (f_1 \textbf{ op } f_2)|_{z=1}) \\ &= ((1-z) \cdot (f_1|_{z=0} \textbf{ op } f_2|_{z=0})) + (z \cdot (f_1|_{z=1} \textbf{ op } f_2|_{z=1})) \end{aligned}$$

□

Algorithm 9 MULTIPLICATION(v_1, v_2)

Input: Nodes v_1, v_2 of a π-ADD $\mathcal{A}$
Output: Node v with $f_v = f_{v_1} \cdot f_{v_2}$

```
if (v1 ∈ V_T and value(v1) = 0) or (v2 ∈ V_T and value(v2) = 0) then
  return find_or_add_drain(0)
else if v1 ∈ V_T and v2 ∈ V_T then
  return find_or_add_drain(value(v1) · value(v2))
else
  z ← min{var(v1), var(v2)}

  w0 ←MULTIPLICATION(f_v1|_{z=0}, f_v2|_{z=0})
  w1 ←MULTIPLICATION(f_v1|_{z=1}, f_v2|_{z=1})

  return find_or_add(z, w0, w1)
end if
```

Algorithm 9 illustrates the multiplication on ADDs. The terminal case where one factor equals zero is checked just for performance reasons. Without this check, the algorithm does not create redundant nodes but a lot of traversing would be done without effect. The other terminal case, which is not checked here, would be the test if one multiplier equals one. The remaining subtree could be returned instead of traversing it without altering the values. Other terminal cases that will be explained for an advanced variant of ADDs in Section 4.3 are impossible on this basic variant of ADDs. Whenever the algorithm reaches two drains the new value will be calculated and the corresponding ADD function will be created from bottom to top.

3.3. Algebraic Computation

ADDs were introduced for algebraic computation, but opposed to their name they are not well suitable for basic algebraic operations.

k	ADD Nodes		
	$\lvert f_{\overline{x}} \rvert$	$\lvert f_{\overline{x}} + f_{\overline{y}} \rvert$	$\lvert f_{\overline{x}} \cdot f_{\overline{y}} \rvert$
0	3	6	6
1	7	18	24
2	15	44	93
3	31	98	352
4	63	208	1377
5	127	430	5358
6	255	876	21078
7	511	1770	83203
8	1023	3560	329908
9	2047	7142	1308670
10	4095	14308	5199280

Table 3.1: Size of the benchmark functions represented by an ADD

3.3.1. Basic Algebraic Operations

ADDs are a good data structure for function representation over small finite domains. With an increasing number of elements the size of the ADD can grow exponentially. This can easily be seen with the previously defined benchmark functions.

3.3.1.1. Benchmark

Table 3.1 shows the size of the benchmark functions represented by an ADD. This time, a slightly different variable ordering $\pi = (x_0, y_0, \ldots, x_k, y_k)$ was used. At a first glance, it is surprising that ADDs are far worse than SOBDDs to represent the benchmark functions. With a closer look thou, these results can be explained.

The function $f_{\overline{x}}$ has exactly 2^{k+1} different values. An ADD function that represents 2^{k+1} values is equivalent to a complete binary tree (see Figure 3.1).

The number of nodes for the function $f_{\overline{x}}$ can be calculated by:

$$|f_{\overline{x}}| = \sum_{i=0}^{k+1} 2^k = 2^{k+2} - 1$$

The chosen variable ordering $\pi = (x_0, y_0, \ldots, x_k, y_k)$ is well-suited for the function $f_{\overline{x}} + f_{\overline{y}}$. The computation of this function with ADDs is completely different from the computation with SOBDDs. In the ADD computation there is

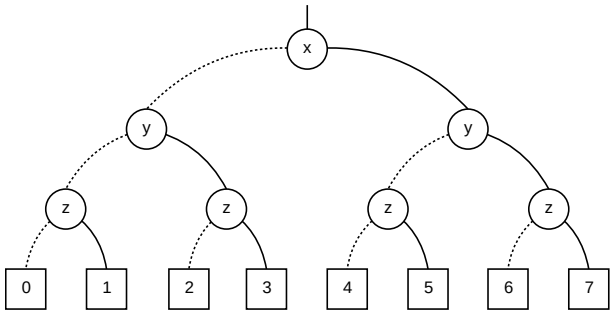


Figure 3.1: ADD function for $f_{\overline{x}}$ with $k = 2$

no add carry bit because the values are calculated as a number and not with their Boolean encoding. When two ADD functions are added, this can be seen as the sum that is calculated with the school method.

Example 3.3.1.

$$\begin{array}{cccccc} & 1 & 2 & 3 & 4 & 5 \\ + & 2 & 4_1 & 6_1 & 8 & 0 \\ \hline & 3 & 7 & 0 & 2 & 5 \end{array}$$

■

The add carry is needed to calculate the next digit. From that point of view it is clear that the variables x_i and y_i should be interleaved and ordered from x_0, y_0 to x_k, y_k.

The product function $f_{\overline{x}} \cdot f_{\overline{y}}$ has $\Theta(\frac{4^k}{k})$ values. This complexity can be proven with the prime number theorem [Corman 1990] together with the approximation for the prime sum. Thus it cannot be expected to find a good variable ordering for the multiplication function.

But basic algebraic operations are not the intended application for ADDs. They are designed for algebraic operations on matrices over a small domains.

3.3.2. Matrix Representation

With ADDs Matrix representation can be done straightforward. The idea to encode a matrix with ADDs is to have the input variables x_i coding the rows and y_i coding the columns.

Definition 3.3.2. Let $f^i \in \mathbb{K}(\mathcal{Z})$ be defined by:

$$f^i([\overline{x} = \overline{a}]) = a_i.$$

□

With Definition 3.3.2 an entry $a_{i,j}$ of a matrix $A \in \mathbb{K}^{n \times m}$ can be encoded with an ADD function $f_{a_{i,j}}$:

$$f_{a_{i,j}}(\overline{x}, \overline{y}) = a_{i,j} \cdot f^i(\overline{x}) \cdot f^j(\overline{y}).$$

The sum over all $a_{i,j}$ has to be calculated to represent the matrix A:

$$f_A(\overline{x}, \overline{y}) = \sum_{i=1}^{n} \sum_{j=1}^{m} f_{a_{i,j}}(\overline{x}, \overline{y}).$$

3.3.2.1. Benchmark

The main purpose of ADDs is the representation of matrices. The sizes to represent certain matrices will be compared to provide a comparison between this OBDD variant and the later introduced NADDs.

The used variable ordering $\pi = (x_1, y_1, \ldots, x_n, y_n)$ for matrix representation interleaves the variables for the rows (x_i) and columns (y_i). This has been proven to be a good variable ordering because every entry usually depends on the rows and columns of a matrix. The first matrix is the identity matrix

$$\mathbb{1}_k = (\delta_{i,j})_{1 \leq i,j \leq k} = \begin{cases} 1 & \text{for } i = j \\ 0 & \text{otherwise.} \end{cases}$$

This matrix was chosen to show how the size of the compared BDD variants vary for a small domain. $(\log_2(k) - 1) \cdot 3 + 5$ nodes are needed to represent $\mathbb{1}_k$.

As a second example, the Hilbert matrix $H_k = (\frac{1}{i+j-1})_{1 \leq i,j \leq k}$ was selected to provide a comparison for a larger domain. H_k is symmetric so that the symbolic representation of that matrix should have many coinciding nodes. The domain of the Hilbert matrix increases with growing k and thus the number of nodes to represent it. The number of different values for this matrix is $2k - 1$ and these values are ordered in a way so that a BDD representation is not very compact. At least $3k$ nodes are needed to represent H_k with $2k - 1$ different values.

The results for the first two benchmark matrices represented by an ADD can be seen in Table 3.2.

The last matrix is the sparse matrix ORANI678 taken from [Matrix Market]. This matrix represents the economic model of Australia with the data from 1968-1969. Figure 3.2 shows the shape of the matrix. The ORANI678 matrix has a much worse ADD representation. To represent the 90158 entries of this matrix 472187 ADD nodes are needed.

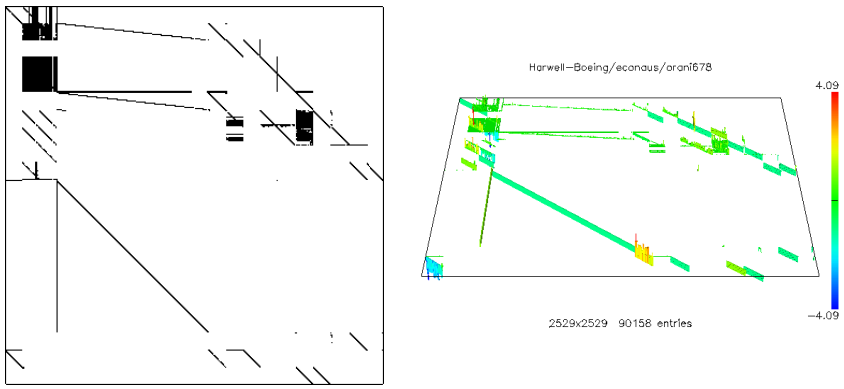


Figure 3.2: Structure and city plot of ORANI678

k	ADD Nodes	
	$\lvert\mathbb{1}_k\rvert$	$\lvert H_k\rvert$
2	5	6
4	8	18
8	11	44
16	14	98
32	17	208
64	20	430
128	23	876
256	26	1770
512	29	3560
1024	32	7142

Table 3.2: Size of the benchmark matrices represented by an ADD

4 Normalized Algebraic Binary Decision Diagrams

As seen in Chapter 3 ADDs are not a good data structure for basic algebraic operations. They can better be used for matrix operations.

In this chapter we introduce a variant of OBDDs, called Normalized Algebraic Binary Decision Diagrams (NADDs) which are designed to combine the advantages of SOBDDs and ADDs. Further on, we will show how they are designed and why they serve as a much better data structure for algebraic computation than SOBDDs and ADDs do.

4.1. Definition and Semantics

The idea of NADDs is based on attributed edges. Instead of trying to reduce the size of an OBDD with small additional effort while traversing the BDD[1], NADDs calculate a lot on their way to the drain.

A wide class of functions over Boolean variables can be viewed as linear functions[2]. E.g. a polynomial function

$$p(x) = \sum_{i=0}^{n} a_i x^i$$

over the Boolean variable x it holds:

$$\begin{aligned} p(x) = \sum_{i=0}^{n} a_i x^i &= a_0 + a_1 \cdot x + a_2 \cdot x^2 + \ldots + a_n \cdot x^n \\ &= a_0 + a_1 \cdot x + a_2 \cdot x + \ldots + a_n \cdot x \\ &= a_0 + x \cdot (a_1 + a_2 + \ldots + a_n) \\ &= a_0 + x \cdot \sum_{i=1}^{n} a_i \end{aligned}$$

For this reason, a scalar multiplication factor λ and a translation τ would be of great use to get a compact BDD representation for this kind of functions.

[1] like SOBDDs with negative edges

[2] with some restrictions

The intention of establishing NADDs is to achieve scalar multiplication and translation in constant time without losing the advantages (like uniqueness, etc.) of BDDs.

Definition 4.1.1. [(λ, τ)**-Algebraic Binary Decision Diagram**]**:** Let π be a variable ordering of $\mathcal{Z}$. A (λ, τ)-ADD is a tuple

$$\mathcal{A}_{(\lambda,\tau)} = (V, V_I, V_T, value, \lambda_0, \tau_0, succ_0, \lambda_1, \tau_1, succ_1, var, value, \overline{\langle \lambda, \tau, v \rangle})$$

that contains:

- a finite set of nodes $V = V_I \cup V_T$ with $V_I \cap V_T = \emptyset$ (V_I contains the inner nodes and V_T the drains),
- a map $value$:
$$value : V_T \to \mathbb{R}$$
- functions $succ_0$ and $succ_1$
$$succ_0, succ_1 : V_I \to V$$
that map every inner node to its successors,
- maps λ_i, τ_i with $i \in \{0, 1\}$
$$\lambda_i, \tau_i : V_I \to \mathbb{R}$$
- a function $var : V_I \to \mathcal{Z}$ that yields a labeling of the nodes with variables,
- a tuple of root node triples:
$$\overline{\langle \lambda, \tau, v \rangle} = \left(\left\langle \widehat{\lambda}_1, \widehat{\tau}_1, v_1 \right\rangle, \ldots, \left\langle \widehat{\lambda}_k, \widehat{\tau}_k, v_k \right\rangle \right) \in (\mathbb{R} \times \mathbb{R} \times V)^k$$

Additionally, for all $v \in V_I$ and $b \in \{0, 1\}$ the following must hold:

$$var(v) <_\pi var(succ_b(v)) \text{ if } succ_b \in V_I.$$

□

This modification of ADDs leads to an intuitive interpretation of a function represented by a (λ, τ)-ADD.

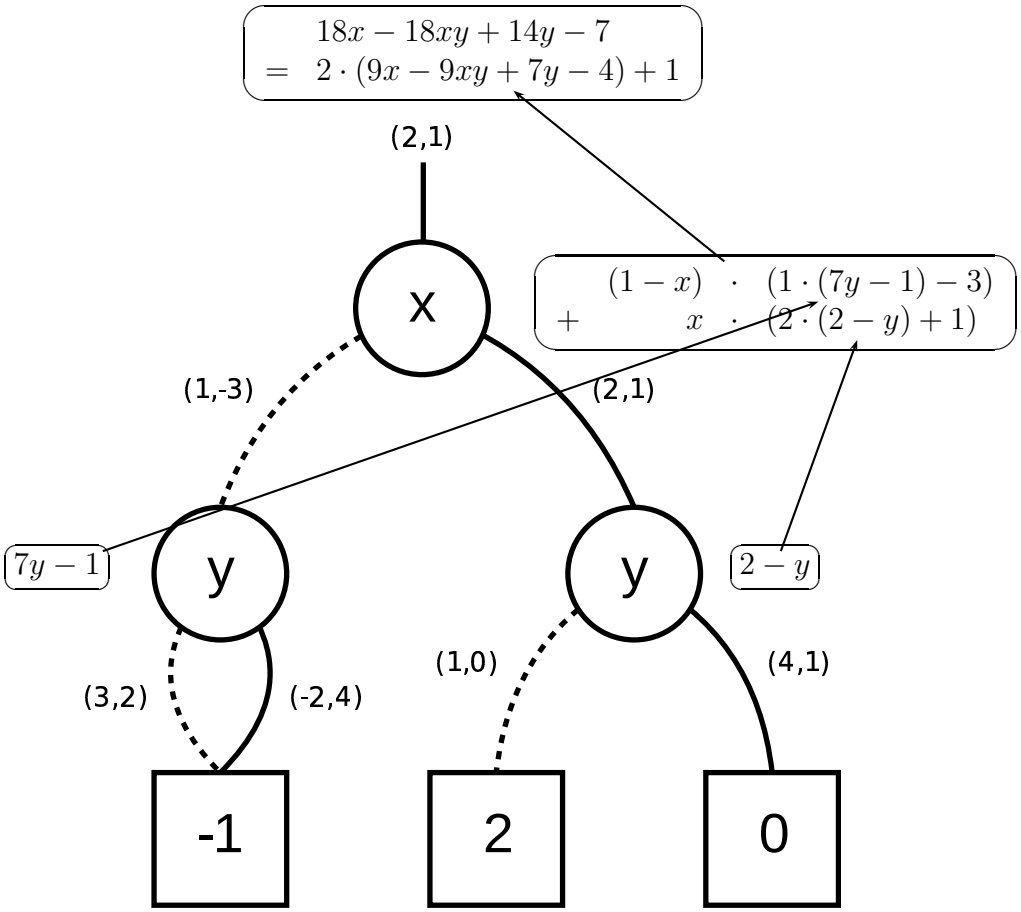


Figure 4.1: Bottom-up definition of a function represented by a (λ, τ)-ADD

Definition 4.1.2. [**Function of a** (λ, τ)**-ADD**]: Let $\mathcal{A}$ be a (λ, τ)-ADD with variable set $\mathcal{Z}$. A function $f_v \in \mathbb{R}(\mathcal{Z})$ is assigned to every node v:

$$f_v = \begin{cases} value(v) & v \in V_T \\ (1 - z) \cdot (\lambda_0(v) \cdot f_{succ_0(v)} + \tau_0(v)) & v \in V_I \text{ with } var(v) = z \\ + \quad z \cdot (\lambda_1(v) \cdot f_{succ_1(v)} + \tau_1(v)) & \end{cases}$$

Every root node tuple $\left\langle \widehat{\lambda}_i, \widehat{\tau}_i, v_i \right\rangle$ will be assigned to

$$f_{\left\langle \widehat{\lambda}_i, \widehat{\tau}_i, v_i \right\rangle} = \widehat{\lambda}_i \cdot f_{v_i} + \widehat{\tau}_i$$

□

Figure 4.1 shows a (λ, τ)-ADD and its represented function.

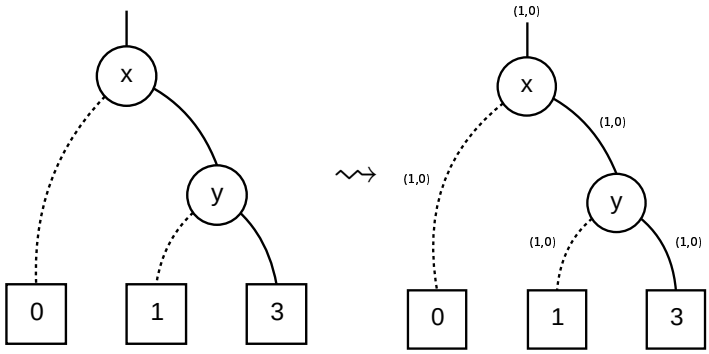
(λ, τ)-ADDs were defined over the field $\mathbb{R}$. Every other field $\mathbb{K}$ was also possible, but $\mathbb{R}$ was chosen to compare this new BDD variant against SOBDDs

and ADDs.

Theorem 4.1.3. [(λ, τ)**-ADDs are universal**]**:** For every function $f \in \mathbb{R}(\mathcal{Z})$ and every variable ordering π exists a π-(λ, τ)-ADD tuple $\langle \lambda, \tau, v \rangle$ with

$$\lambda \cdot f_v + \tau = f.$$

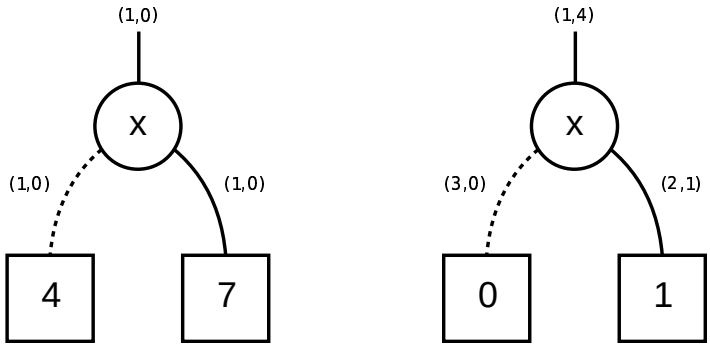
Proof. As shown in Theorem 3.1.4 ADDs are a universal data structure. Every ADD can be converted into a (λ, τ)-ADD.



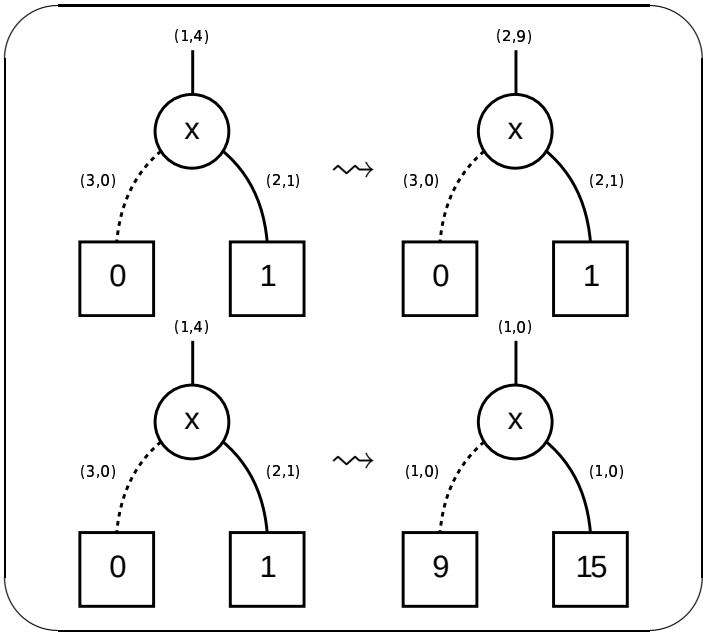
□

The test for equality on SOBDDs and ADDs can be done in constant time because SOBDDs and ADDs yield a unique data structure for any given switching respectively $\mathbb{R}$-function. Unfortunately this does not hold for (λ, τ)-ADDs.

Example 4.1.4. The function $f(x) = 3x + 4$ can be represented in different ways, e.g.:



The function $g(x) = 2 \cdot f(x) + 1$ should share the same nodes as $f(x)$. Instead of only changing the parameters λ and τ completely different nodes can be created.



■

As seen in Section 2.3.3 there are problems to get a unique data structure using attributed edges. The first step towards a unique data structure is to restrict the (λ, τ)-ADD. Multiple drains should be forbidden. But, which drain should be used?

The one drain could be taken[3] but two cases have to be differentiated:

- node $v \in V_I$: Calculation could be done as usual:

$$f = \lambda \cdot f_v + \tau.$$

- node $v \in V_T$: The value of a function pointing to the drain depends on both parameters λ and τ:

$$f = \lambda + \tau$$

  as we assume $value(v) = 1$.

It is much better to chose the zero drain, because no different cases have to be considered for inner nodes and the drain. The value of a function pointing to the zero drain depends on the translation parameter τ only, which makes the interpretation more intuitive. For a given function $f(x) = g(x) + b$ the parameter b is the translation of $g(x)$. If $g(x) = 0$ then f is the constant function b. Hence, every constant function can be seen as a translation from the zero function.

[3] like for SOBDDs with negative edges do

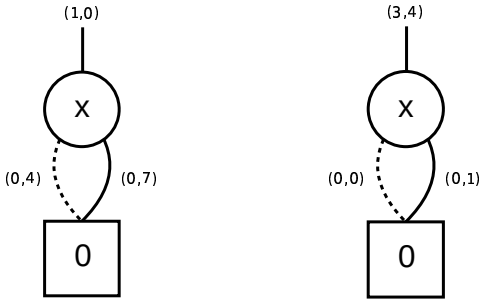


Figure 4.2: The zero drain does not ensure uniqueness

This restriction alone does not help to get a unique data structure (see Figure 4.2).

The next restriction is to normalize (λ, τ)-ADDs.

Definition 4.1.5. [**Normalized Algebraic Binary Decision Diagram**]**:** A (λ, τ)-ADD $\mathcal{A}_{(\lambda,\tau)}$ is called *normalized* if for every node $v \in V_I$ holds:

- $f_v(\eta) \in [0, 1]\ \forall \eta \in \text{Eval}(\mathcal{Z})$
- $\exists \eta \in \text{Eval}(\mathcal{Z})$ with $f_v(\eta) = 0$
- $\exists \zeta \in \text{Eval}(\mathcal{Z})$ with $f_v(\zeta) = 1$

A normalized (λ, τ)-ADD $\mathcal{A}_{(\lambda,\tau)}$ will be called NADD and denoted $\mathcal{A}_{[0,1]}$. Every NADD contains only the zero drain.

□

The main idea behind NADDs is that a lot of functions can possibly share the same nodes. The normalization of (λ, τ)-ADDs makes it possible to calculate the minimum and maximum value of a function in constant time whereas non-normalized (λ, τ)-ADDs have to traverse the whole diagram. This property ensures that a (λ, τ)-ADD $\mathcal{A}_{(\lambda,\tau)}$ can be converted into a NADD $\mathcal{A}_{[0,1]}$ with the complexity $\Theta(|\mathcal{A}_{(\lambda,\tau)}|)$. This conversion has to be done from the bottom to the top level and must be canonical to provide a unique data structure.

4.2. Canonicity

The idea behind converting a (λ, τ)-ADD into a NADD relies on modifying the parameters λ and τ of the incoming edges.

Theorem 4.2.1. If the root node $v \in V_I$ of a (λ, τ)-ADD function $f_{\langle\lambda,\tau,v\rangle}$ is multiplied by a scalar $\mu \in \mathbb{R}\backslash\{0\}$ and translated by $\nu \in \mathbb{R}$ the function

parameters λ and τ can be modified in such a way that the represented function does not change and the subgraphs of v do not have to be altered (see Figure 4.3).

Proof. Let v be a z-node. The scalar multiplication and translation of f_v leaves the subgraphs of v untouched. This can directly be seen from Definition 4.1.2:

$$\begin{aligned} f_{\langle\mu,\nu,v\rangle} &= \mu \cdot f_v + \nu \\ &= \mu \cdot \big(\quad (1-z) \cdot (\lambda_0(v) \cdot f|_{z=0} + \tau_0(v)) \\ &\qquad + \quad z \cdot (\lambda_1(v) \cdot f|_{z=1} + \tau_1(v)) \quad \big) + \nu \\ &= \quad (1-z) \cdot \big((\mu \cdot \lambda_0(v)) \cdot f|_{z=0} + (\mu \cdot \tau_0(v) + \nu)\big) \\ &\quad + \quad z \cdot \big((\mu \cdot \lambda_1(v)) \cdot f|_{z=1} + (\mu \cdot \tau_1(v) + \nu)\big) \end{aligned}$$

Let v' be the modified node with $\lambda_i(v') = \mu \cdot \lambda_i(v)$ and $\tau_i(v') = \mu \cdot \tau_i(v) + \nu$, $i = 0, 1$. All other parameters are taken from v without change.

f_v and $f_{v'}$ represent different functions provided that $(\mu, \nu) \neq (1, 0)$:

$$\begin{aligned} \mu \cdot f_v + \nu &= f_{v'} \\ \Leftrightarrow \qquad f_v &= \tfrac{f_{v'} - \nu}{\mu} \end{aligned}$$

Thus, the function parameters λ and τ cannot be taken for v'. But these parameters can be modified so that the represented functions become equal:

$$\begin{aligned} f_{\langle\lambda,\tau,v\rangle} &= \lambda \cdot f_v + \tau \\ &= \lambda \cdot \left(\frac{f_{v'} - \nu}{\mu}\right) + \tau \\ &= \frac{\lambda}{\mu} \cdot f_{v'} + \left(\tau - \frac{\lambda\nu}{\mu}\right) \\ &= f_{\left\langle \frac{\lambda}{\mu}, \tau - \frac{\lambda\nu}{\mu}, v \right\rangle} \end{aligned}$$

□

Theorem 4.2.1 can be generalized for every inner node. This ensures that a (λ, τ)-ADD can be converted into a NADD in a bottom up strategy.

NADDs are defined to be normalized. But which function has to be used to normalize a (λ, τ)-ADD?

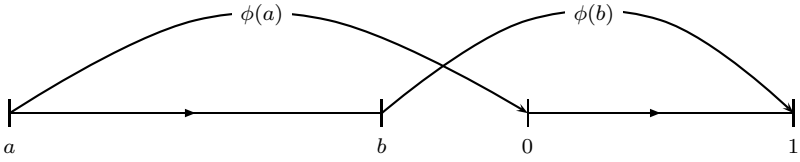
Example 4.2.2. Let $a, b \in \mathbb{R}$ with $a < b$:

$$\phi : [a, b] \to [0, 1]$$

is defined by

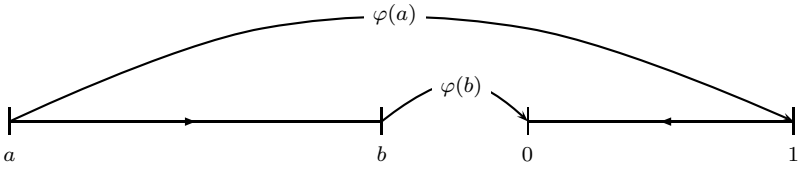
$$\phi(x) = \frac{x - a}{b - a} \text{ for all } x \in [a, b]\,.$$

Figure 4.3: Modifying the parameters without changing the function



This is not the only function that maps $[a, b]$ to $[0, 1]$. The function

$$\varphi(x) = \frac{b - x}{b - a}$$



achieves the same.

■

As seen in Example 4.2.2 normalization is not enough to provide a unique data structure. Other restrictions have to be made:

- $\phi'(x) = \frac{1}{b-a}$
- $\phi(a) = 0$
- $\phi(b) = 1$

Definition 4.2.3. Let a be the minimum and b the maximum value of a given function f with $a < b$. The normalization function $\phi_f : [a, b] \to [0, 1]$ is defined by:

$$\phi_f(x) = \frac{x - a}{b - a}.$$

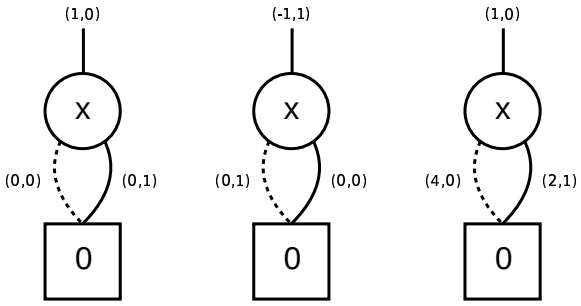


Figure 4.4: Different reduced normalized (λ, τ)-ADDs representing the same function

□

Lemma 4.2.4. The normalization function ϕ_f is invariant under translation, i.e. $\phi_f(x) = \phi_{f+\tau}(x + \tau)$.

Proof. Let f be the given function and $a, b \in \mathbb{R}$ with $a < b$ its minimum and its maximum respectively and $\tau \in \mathbb{R}$ the translation.

$$\begin{aligned}\phi_{f+\tau}(x+\tau) &= \frac{(x+\tau)-(a+\tau)}{(b+\tau)-(a+\tau)} \\ &= \frac{x-a}{b-a} \\ &= \phi_f(x)\end{aligned}$$

□

The negative edges were introduced to negate functions in a SOBDD in constant time. This could be done because a function and its negation were represented by the same subtree. This concept can be expanded on NADDs.

Definition 4.2.5. A NADD $\mathcal{A}_{[0,1]}$ is called *reduced* if for every two nodes $v_1, v_2 \in V$, for all scale factors $\lambda \in \mathbb{R}\backslash\{0\}$ and translations $\tau \in \mathbb{R}$ the following holds:

$$\boxed{v_1 \neq v_2 \Rightarrow f_{v_1} \neq \lambda \cdot f_{v_2} + \tau}$$

In the following, NADDs are assumed to be reduced.

□

But reducedness alone does not guarantee uniqueness. Figure 4.4 shows different reduced normalized (λ, τ)-ADDs that represent the same function.

The situation is similar to the one with SOBDDs with negative edges. There, the negation can only be applied to the zero successor of a node. This method can be extended to NADDs.

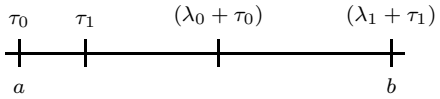
Every node $v \in V_I$ of a NADD is normalized, i.e.

- $\min\{f_v(\eta) | \eta \in \text{Eval}(\mathcal{Z})\} = 0$
- $\max\{f_v(\eta) | \eta \in \text{Eval}(\mathcal{Z})\} = 1$.

This way the maximum and minimum value of f_v can be calculated as follows. For simplification, we write λ_i instead of $\lambda_i(v)$ and τ_i instead of $\tau_i(v)$. Let:

$$a = \min\{(\lambda_0 + \tau_0), \tau_0, (\lambda_1 + \tau_1), \tau_1\}$$
$$b = \max\{(\lambda_0 + \tau_0), \tau_0, (\lambda_1 + \tau_1), \tau_1\}$$

The different parameter values can be interpreted as values in the interval $[a, b]$. For instance:



All possible λ_i and τ_i have to be combined in a unique way. This can be assured if the normalization function is invariant under scalar multiplication and translation. In Lemma 4.2.4 we have proven the translation invariance of the normalization function ϕ_f. Unfortunately, this does not hold for scalar multiplication.

Theorem 4.2.6. The normalization function is not invariant under scalar multiplication.

Proof. Let f be the given function and a, b with $a < b$ its minimum and its maximum respectively. If this function would be invariant under scalar multiplication then the following holds:

$$\phi_{\lambda \cdot f}(\lambda \cdot x) = \phi_f(x).$$

This holds for all $\lambda \in \mathbb{R}^{>0}$. For $\lambda = -1$ the minimum and maximum values are $-b$ and $-a$ respectively.

$$\phi_{-f}(-x) = \frac{b - x}{b - a} \neq \frac{x - a}{b - a} = \phi_f(x).$$

□

Theorem 4.2.6 shows that normalization cannot ensure uniqueness. But the parameters can be modified in a way that the normalization function becomes invariant under scalar multiplication. For this reason, additional constraints for NADDs have to be expressed.

Let $\langle \lambda, \tau, v \rangle$ be a NADD-tuple and $\lambda_i = \lambda_i(v), \tau_i = \tau_i(v)$.

- **0-Scalar:** $\langle \lambda, \tau, v \rangle$ with $v \in V_T \Leftrightarrow \lambda = 0$
- **λ_0-Positivity:** $v \in V_I \Rightarrow \lambda_0 \geq 0$
- **λ_1-Positivity:** $v \in V_I$ with $\lambda_0 = 0 \Rightarrow \lambda_1 \geq 0$
- **τ_0-Arrangement:** $v \in V_I$ with $\lambda_0 = \lambda_1 = 0 \Rightarrow \tau_0 \leq \tau_1$

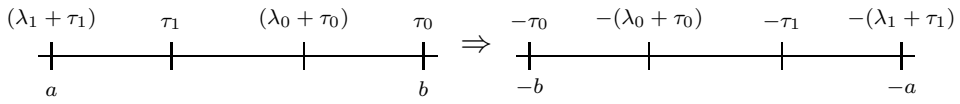
The first restriction ensures that constant functions are uniquely represented by the zero drain. All other limitations have to be made to guarantee that all possible λ_i and τ_i are handled the same way.

The case $\lambda_0 = \lambda_1 = 0$ and $\tau_0 = \tau_1$ is equivalent to the elimination rule (see page 9). Thus, this is not a possible instance for the parameters.

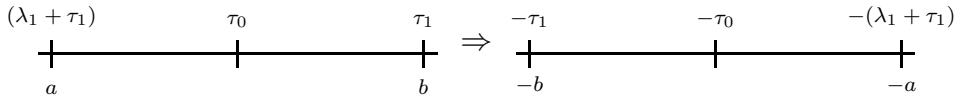
Example 4.2.7 shows how it can be guaranteed that λ_i and τ_i fulfill these restrictions.

Example 4.2.7. The parameters can be modified so that all limitations will be fulfilled whenever one of the restrictions is violated . This can be reached by multiplying all parameters with -1. Theorem 4.2.1 shows that this modification is valid.

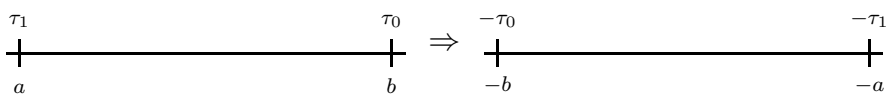
- **λ_0-Positivity:** $v \in V_I$ with $\lambda_0 < 0$



- **λ_1-Positivity:** $v \in V_I$ with $\lambda_0 = 0$ and $\lambda_1 < 0$



- **τ_0-Arrangement:** $v \in V_I$ with $\lambda_0 = \lambda_1 = 0$ and $\tau_0 > \tau_1$



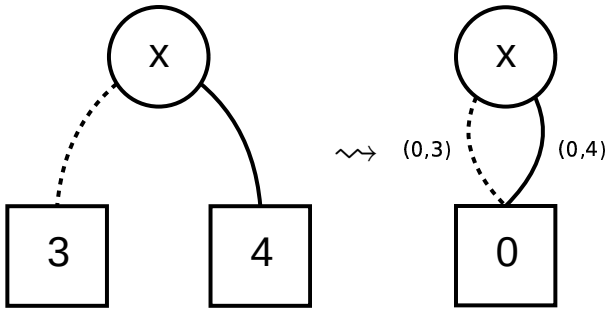
■

With these restrictions the normalization function is invariant under scalar multiplication and translation. In the following these limitations will be used for NADDs.

Theorem 4.2.8. [**NADDs are universal**]**:** For every function $f \in \mathbb{R}(\mathcal{Z})$ and every variable ordering π there exists a π-NADD tuple $\langle \lambda, \tau, v \rangle$ with

$$\lambda \cdot f_v + \tau = f.$$

Proof. As shown in Theorem 4.1.3 there exists a (λ, τ)-ADD function for f. All drains of the (λ, τ)-ADD can be converted into a (λ, τ)-ADD with only the zero drain.



The normalization function can be seen as scalar multiplication with following translation. With Theorem 4.2.1 this (λ, τ)-ADD can be converted into a NADD. □

Lemma 4.2.9. Let $f : \mathrm{Eval}(\mathcal{Z}) \to \mathbb{R}$ be a non-constant function and

$$a = \min f(\eta),\ b = \max f(\eta)$$

where η ranges over all evaluations for $\mathcal{Z}$.

Furthermore, let $\lambda, \tau \in \mathbb{R}$ and $g : \mathrm{Eval}(\mathcal{Z}) \to \mathbb{R}$ such that

$$\min_{\eta \in \mathrm{Eval}(\mathcal{Z})} g(\eta) = 0 \text{ and } \max_{\eta \in \mathrm{Eval}(\mathcal{Z})} g(\eta) = 1.$$

Then the following statements are equivalent:

(i)
$$f = \lambda \cdot g + \tau$$

(ii)
$$\begin{array}{rrcl} & \langle \lambda, \tau, g \rangle & = & \langle \quad b - a \ , a, \qquad \phi_f \circ f \ \rangle \\ \text{or} & \langle \lambda, \tau, g \rangle & = & \langle -(b-a), b, \underbrace{1 - (\phi_f \circ f)}_{=(1-\phi_f)\circ f} \rangle \end{array}$$

Proof. We assume that $f = \lambda \cdot g + \tau$ as in (i). f is a non-constant function and from this it follows that $\lambda \neq 0$ and $b - a \neq 0$.

- $\lambda > 0$:

$$\boxed{\begin{array}{lllllll} a & = & \min f(\eta) & = & \lambda \cdot 0 + \tau & = & \tau \\ b & = & \max f(\eta) & = & \lambda \cdot 1 + \tau & = & \lambda + \tau \end{array}} \Rightarrow \lambda = b - a$$

$$\begin{array}{rrcl} & (b-a) \cdot g(\eta) + a & = & f(\eta) \\ \Leftrightarrow & (b-a) \cdot g(\eta) & = & f(\eta) - a \\ \Leftrightarrow & g(\eta) & = & \frac{f(\eta)-a}{b-a} = \phi_f \circ f \end{array}$$

  Note that "$\Leftarrow$" proves the first case of (ii) $\Rightarrow$ (i).

- $\lambda < 0$:

$$\boxed{\begin{array}{lllllll} a & = & \min f(\eta) & = & \lambda \cdot 1 + \tau & = & \lambda + \tau \\ b & = & \max f(\eta) & = & \lambda \cdot 0 + \tau & = & \tau \end{array}} \Rightarrow \lambda = a - b = -(b - a)$$

$$\begin{array}{rrcl} & -(b-a) \cdot g(\eta) + b & = & f(\eta) \\ \Leftrightarrow & -(b-a) \cdot g(\eta) & = & f(\eta) - b \\ \Leftrightarrow & g(\eta) & = & \frac{b-f(\eta)}{b-a} = \varphi_f \circ f \end{array}$$

φ can be expressed through ϕ so that only one normalization function has to be used.

$$\begin{aligned} 1 - \phi_f(x) &= 1 - \frac{x-a}{b-a} \\ &= \frac{b-a-(x-a)}{b-a} \\ &= \frac{b-x}{b-a} \\ &= \phi_{-f}(-x) \\ &= \varphi_f(x) \end{aligned}$$

□

From Lemma 4.2.9 it follows that there are just two ways to split a function f into its normalized component g and capable λ and τ. It is essential to prove that only one of the decompositions is valid for NADDs, i.e. for every non-constant function $f = f_{\langle \lambda, \tau, v \rangle}$ holds either $f_v = \phi_f \circ f$ or $f_v = 1 - (\phi_f \circ f)$.

Corollary 4.2.10. Let $f : \mathrm{Eval}(\mathcal{Z}) \to \mathbb{R}$ be a non-constant function with $\mathcal{Z} = \{z\}$, i.e. $f|_{z=0}$ and $f|_{z=1}$ are constant functions with $f|_{z=0} \neq f|_{z=1}$. Then, f can uniquely be represented by a NADD.

Proof. The 0-Scalar property ensures that every constant function with value $c \in \mathbb{R}$ has a unique NADD representation:

$$c = 0 \cdot f_{\boxed{0}} + c = f_{\langle 0,c,\boxed{0}\rangle}.$$

Let $c_0 = f|_{z=0}$ and $c_1 = f|_{z=1}$ the represented values of the cofactors of f. f can be expressed as:

$$f = \lambda \cdot z + \tau$$

with $\lambda = c_1 - c_0$ and $\tau = c_0$. The uniqueness of the constant function representation indicates that $\lambda_0(v) = \lambda_1(v) = 0$ for a z-node $v \in V_I$ that represents $\phi_f \circ f$ or $1 - (\phi_f \circ f)$. The τ_0-Arrangement property defines which function is represented by v.

- $c_0 < c_1 \Leftrightarrow \lambda > 0 \Rightarrow f_v = \phi_f \circ f$
- $c_0 > c_1 \Leftrightarrow \lambda < 0 \Rightarrow f_v = 1 - (\phi_f \circ f)$

Lemma 4.2.9 shows the uniqueness of the triple $\langle \lambda, \tau, v\rangle$ with $f = f_{\langle \lambda,\tau,v\rangle}$. □

Theorem 4.2.11. [**NADDs are unique**]**:** The NADD representation of a function $f : \mathrm{Eval}(\mathcal{Z}) \to \mathbb{R}$ with given variable ordering π is unique[4].

Proof. We may assume without loss of generality that all variables in $\mathcal{Z}$ are essential for f. By induction on $n = |\mathcal{Z}|$ (number of variables) we show that for any function $f : \mathrm{Eval}(\mathcal{Z}) \to \mathbb{R}$ where $a = \min f(\eta)$, $b = \max f(\eta)$ it holds:

There exists a unique triple $\langle \lambda, \tau, v\rangle$ such that $f = \lambda f_v + \tau$ where v is a NADD node and $\lambda, \tau \in \mathbb{R}$ and $\lambda = 0$ iff f is constant.

Basis of induction:

- $n = 0$: Then f is constant, i.e. $a = b$. In a NADD only the zero drain is allowed. This, together with the 0-Scalar property induces that the only chance to obtain a NADD for f is to use the expansion:

$$f = \underbrace{0}_{=\lambda} \cdot f_{\boxed{0}} + \underbrace{a}_{=\tau}.$$

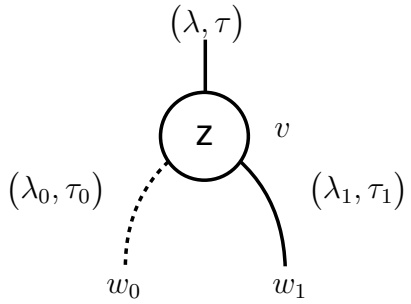
- $n = 1$: Follows by Corollary 4.2.10.

[4]Uniqueness is understood with respect to isomorphism of the sub NADDs with root v.

Step of induction $n-1 \Rightarrow n$ $(n > 1)$:
We may assume without loss of generality that z is the first essential variable of the given variable ordering π. By Lemma 4.2.9 f can be represented by:

$$f = \lambda \cdot g + \tau$$

with $\lambda \neq 0$, $\min g = 0$ and $\max g = 1$ in exactly two ways that can be distinguished by $\lambda > 0$ or $\lambda < 0$. Every z-node v has the following form:



As a consequence of Lemma 4.2.9 each normalized node v can only represent either g or $1 - g$.

We also use the decomposition into the cofactors $f|_{z=0}$ and $f|_{z=1}$ of f.

$$\begin{aligned} \lambda \cdot g + \tau = f & \\ &= z \cdot f|_{z=1} + (1 - z) \cdot f|_{z=0} \\ &= z \cdot (f|_{z=1} - f|_{z=0}) + f|_{z=0} \end{aligned}$$

This equation leads to:

$$g = \frac{z \cdot (f|_{z=1} - f|_{z=0}) + f|_{z=0} - \tau}{\lambda}.$$

The cofactors of g have the following form:

$$g|_{z=0} = \frac{f|_{z=0} - \tau}{\lambda} \text{ and } g|_{z=1} = \frac{f|_{z=1} - \tau}{\lambda}.$$

According to Lemma 4.2.9 every non-constant cofactor of f can be expressed as:

$$f|_{z=i} = \lambda_i \cdot g_i + \tau_i$$

with $i = 0, 1$. If $f|_{z=i}$ is a non-constant function then this also holds for

$$g|_{z=i} = \frac{\lambda_i \cdot g_i + \tau_i - \tau}{\lambda}.$$

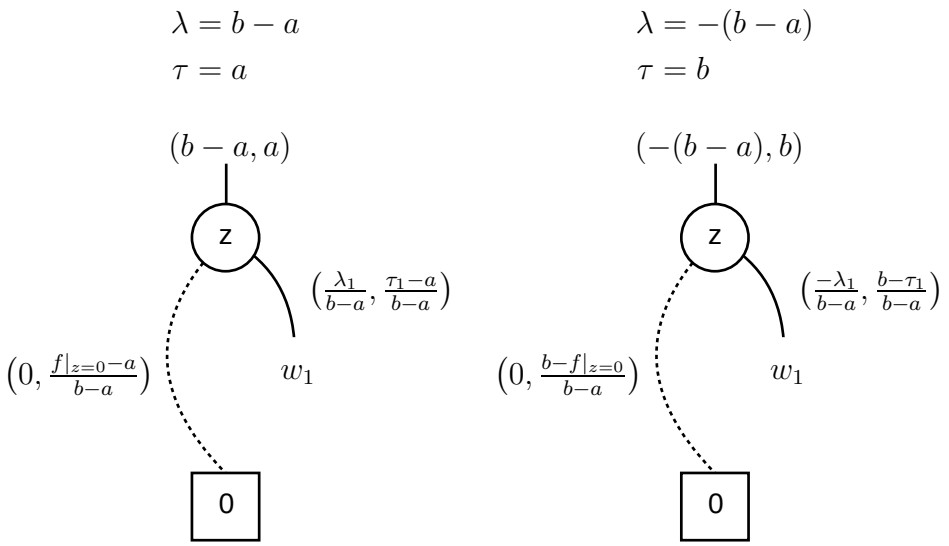
But this equation does not always hold. The case where both cofactors are constant functions was already proven by the induction basis ($n = 1$). This way, we only have to show the three remaining situations.

- Case 1: $f|_{z=0}$ is constant, i.e. $f|_{z=1}$ has $n-1$ essential variables.
  The induction basis ($n = 0$) and the induction hypothesis yield that $g|_{z=0}$ and $g|_{z=1}$ can uniquely be represented by

$$\left\langle 0, \frac{f|_{z=0} - \tau}{\lambda}, \boxed{0} \right\rangle \text{ and } \left\langle \frac{\lambda_1}{\lambda}, \frac{\tau_1 - \tau}{\lambda}, w_1 \right\rangle.$$

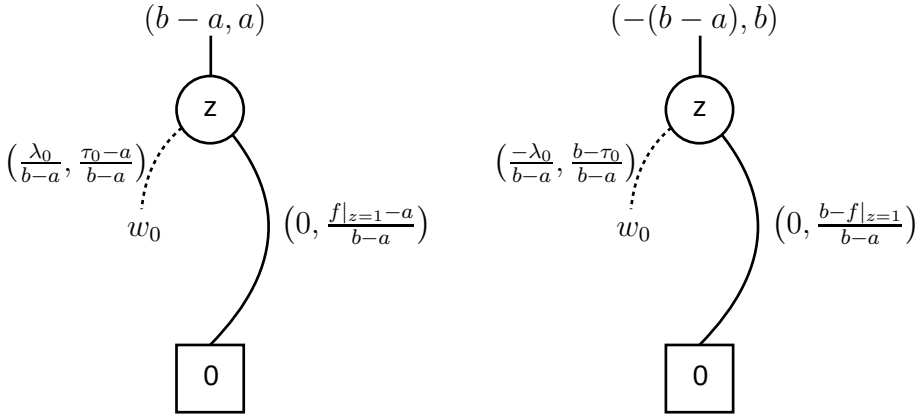
  The induction hypothesis implies that exactly one normalized (λ, τ)-ADD representation of g_1 or $1 - g_1$ does not violate the conditions for NADDs. Without loss of generality we assure that $f_{w_1} = g_1$.

  As mentioned above, we have to differentiate two cases:



  The λ_1-Positivity is violated by either $\frac{\lambda_1}{b-a}$ or $\frac{-\lambda_1}{b-a}$. Thus, a NADD node $v \in V_I$ can only represent one of the two functions g or $1 - g$, but not both.

- Case 2: $f|_{z=1}$ is constant, i.e. $f|_{z=0}$ has $n-1$ essential variables.
  This case can be proven analogously.

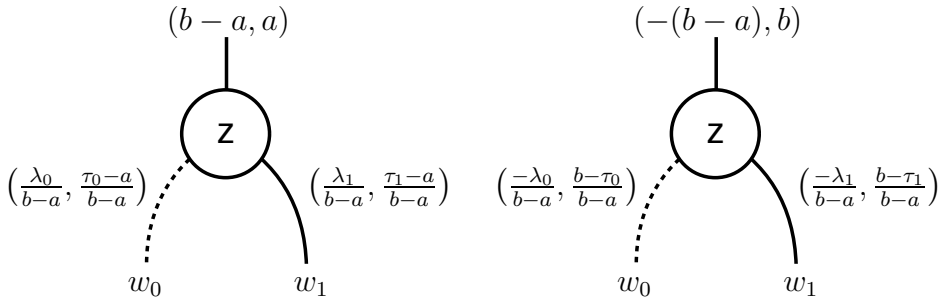


The λ_0-Positivity induces if $g = \phi_f \circ f$ or $1 - g = 1 - (\phi_f \circ f)$ violates the NADD conditions.

- Case 3: $f|_{z=0}$ and $f|_{z=1}$ are non-constant functions, i.e. $f|_{z=0}$ and $f|_{z=1}$ have at most $n-1$ essential variables.
  The induction hypothesis induces that $g|_{z=0}$ and $g|_{z=1}$ can uniquely be represented by

$$\left\langle \frac{\lambda_0}{\lambda}, \frac{\tau_0 - \tau}{\lambda}, w_0 \right\rangle \text{ and } \left\langle \frac{\lambda_1}{\lambda}, \frac{\tau_1 - \tau}{\lambda}, w_1 \right\rangle.$$

  As in case 1, the induction hypothesis induces that only one function can be represented by w_0 and w_1. Furthermore, we assume without loss of generality that $f_{w_0} = g_0$ and $f_{w_1} = g_1$.



  The λ_0-Positivity is violated by either $\frac{\lambda_0}{b-a}$ or $\frac{-\lambda_0}{b-a}$. Thus, a NADD node $v \in V_I$ can only represent one of the two functions g or $1 - g$, but not both.

This proves the induction hypothesis. □

4.3. Algorithms

The reduction rules for NADDs are more complicated than for all other shown OBDD variants. So the **find_or_add** function has to be reformulated. The concept of all algorithms on OBDDs was to build the resulting function from the bottom to the top. As seen in Section 2.3.3 situations which violate the given restrictions have to be eliminated. One restriction for SOBDDs with negative edges was that only the zero successor could have a negative edge. The situation with NADDs is similar to this. Algorithm 10 shows the **find_or_add** function for NADDs. The λ_0-Positivity, λ_1-Positivity and the τ_0-Arrangement are directly checked in this algorithm. The 0-Scalar property cannot be guaranteed in the **find_or_add** function. It must be checked in every algorithm

Algorithm 10 find_or_add($z, \langle \lambda_0, \tau_0, w_0 \rangle, \langle \lambda_1, \tau_1, w_1 \rangle$)

Input: Variable $z \in \mathcal{Z}$ and tuples $\langle \lambda_0, \tau_0, w_0 \rangle, \langle \lambda_1, \tau_1, w_1 \rangle$ of a π-NADD
Output: Tuple $\langle \lambda, \tau, v \rangle$

if $\langle \lambda_0, \tau_0, w_0 \rangle = \langle \lambda_1, \tau_1, w_1 \rangle$ **then**
  return $\langle \lambda_0, \tau_0, w_0 \rangle$
else if $(\lambda_0 < 0)$ or $(\lambda_0 = 0 \wedge \lambda_1 < 0)$ or $(\lambda_0 = 0 \wedge \lambda_1 = 0 \wedge \tau_1 < \tau_0)$ **then**
  $a \leftarrow \min\{-(\lambda_0 + \tau_0), -\tau_0, -(\lambda_1 + \tau_1), -\tau_1\}$
  $b \leftarrow \max\{-(\lambda_0 + \tau_0), -\tau_0, -(\lambda_1 + \tau_1), -\tau_1\}$

  $t_0 \leftarrow \left\langle \frac{-\lambda_0}{b-a}, \frac{-\tau_0 - a}{b-a}, w_0 \right\rangle$
  $t_1 \leftarrow \left\langle \frac{-\lambda_1}{b-a}, \frac{-\tau_1 - a}{b-a}, w_1 \right\rangle$
  if $\exists v \in V_I$ with $var(v) = z$, $\langle \lambda_0(v), \tau_0(v), succ_0(v) \rangle = t_0$ and $\langle \lambda_1(v), \tau_1(v), succ_1(v) \rangle = t_1$ **then**
    return $\langle a - b, -a, v \rangle$
  else
    $v \leftarrow$ new z-node v with $\langle \lambda_0(v), \tau_0(v), succ_0(v) \rangle = t_0$ and $\langle \lambda_1(v), \tau_1(v), succ_1(v) \rangle = t_1$
    return $\langle a - b, -a, v \rangle$
  end if
else
  $a \leftarrow \min\{(\lambda_0 + \tau_0), \tau_0, (\lambda_1 + \tau_1), \tau_1\}$
  $b \leftarrow \max\{(\lambda_0 + \tau_0), \tau_0, (\lambda_1 + \tau_1), \tau_1\}$

  $t_0 \leftarrow \left\langle \frac{\lambda_0}{b-a}, \frac{\tau_0 - a}{b-a}, w_0 \right\rangle$
  $t_1 \leftarrow \left\langle \frac{\lambda_1}{b-a}, \frac{\tau_1 - a}{b-a}, w_1 \right\rangle$
  if $\exists v \in V_I$ with $var(v) = z$, $\langle \lambda_0(v), \tau_0(v), succ_0(v) \rangle = t_0$ and $\langle \lambda_1(v), \tau_1(v), succ_1(v) \rangle = t_1$ **then**
    return $\langle b - a, a, v \rangle$
  else
    $v \leftarrow$ new z-node v with $\langle \lambda_0(v), \tau_0(v), succ_0(v) \rangle = t_0$ and $\langle \lambda_1(v), \tau_1(v), succ_1(v) \rangle = t_1$
    return $\langle b - a, a, v \rangle$
  end if
end if

on NADDs. All calculations used in Algorithm 10 are results from Theorem 4.2.11.

Algorithm 11 explains the multiplication on NADDs. As described above the 0-Scalar property is checked here to guarantee a unique NADD representation. This algorithm is a good example why this property must be checked in every algorithm. In this case the only chance to get a constant function (if one function is non-constant) is the multiplication by 0. In the addition the only possible case would be if f and $-f$ are added. The check for such 0-Scalar possibilities is the main difference in writing algorithms for NADDs compared to ADDs.

The terminal case if one multiplier is constant is a great performance enhancement. No more traversation has to be performed if a result can be calculated with scalar multiplication and translation.

4.4. Algebraic Computation

NADDs were developed to provide a suitable data structure for algebraic computations even with a large domain. The sizes of ADDs are greatly influenced by the domain. The distances of the represented values of a function influences a NADD. The upper bound for the size of a NADD size is the ADD size which can easily be seen in the transformation from an ADD into a (λ, τ)-ADD. In many cases the size of a NADD is much smaller than the ADD size of a function. The relationship between SOBDDs and SOBDDs with negative edges is similar to the relationship between ADDs and NADDs.

4.4.1. Basic Algebraic Computation

In this section the ability for algebraic computations with NADDs will be analyzed.

4.4.1.1. Benchmark

Table 4.1 shows the results of the benchmark functions represented by NADDs. The used variable ordering is $\pi = (x_0, \ldots, x_k, y_0, \ldots, y_k)$. Unlike the situation for ADDs, this is the optimal ordering for addition and multiplication for NADDs. The representation of the function $f_{\overline{x}}$ is linear in terms of size. Every new variable splits the represented domain to two parts with equal distances. For that kind of function the normalization is optimal. For this benchmark function all variable orderings are optimal. Functions with equal distances in their divided subtrees can be represented with one node on this level. Why

Algorithm 11 MULTIPLICATION($\langle \lambda_0, \tau_0, v_0 \rangle, \langle \lambda_1, \tau_1, v_1 \rangle$)

Input: Tuples $\langle \lambda_0, \tau_0, v_0 \rangle, \langle \lambda_1, \tau_1, v_1 \rangle$ of a π-NADD
Output: Tuple $\langle \lambda, \tau, v \rangle$ with $\lambda \cdot f_v + \tau = (\lambda_0 \cdot f_{v_0} + \tau_0) \cdot (\lambda_1 \cdot f_{v_1} + \tau_1)$

if $v_0 \in V_T$ **then**
  if $\tau_0 = 0$ **then**
    return $\left\langle 0, 0, \boxed{0} \right\rangle$
  else
    return $\langle \tau_0 \cdot \lambda_1, \tau_0 \cdot \tau_1, v_1 \rangle$
  end if
else if $v_1 \in V_T$ **then**
  if $\tau_1 = 0$ **then**
    return $\left\langle 0, 0, \boxed{0} \right\rangle$
  else
    return $\langle \tau_1 \cdot \lambda_0, \tau_0 \cdot \tau_1, v_2 \rangle$
  end if
else
  $z \leftarrow \min\{var(v_0), var(v_1)\}$

  $\langle \lambda_{0,0}, \tau_{0,0}, v_{0,0} \rangle \leftarrow \langle \lambda_0 \cdot \lambda_0(v_0|_{z=0}), \lambda_0 \cdot \tau_0(v_0|_{z=0}) + \tau_0, v_0|_{z=0} \rangle$
  $\langle \lambda_{1,0}, \tau_{1,0}, v_{1,0} \rangle \leftarrow \langle \lambda_1 \cdot \lambda_0(v_1|_{z=0}), \lambda_1 \cdot \tau_0(v_1|_{z=0}) + \tau_1, v_1|_{z=0} \rangle$
  $\langle \lambda_{w_0}, \tau_{w_0}, w_0 \rangle \leftarrow$MULTIPLICATION$(\langle \lambda_{0,0}, \tau_{0,0}, v_{0,0} \rangle, \langle \lambda_{1,0}, \tau_{1,0}, v_{1,0} \rangle)$

  $\langle \lambda_{0,1}, \tau_{0,1}, v_{0,1} \rangle \leftarrow \langle \lambda_0 \cdot \lambda_1(v_0|_{z=1}), \lambda_0 \cdot \tau_1(v_0|_{z=1}) + \tau_0, v_0|_{z=1} \rangle$
  $\langle \lambda_{1,1}, \tau_{1,1}, v_{1,1} \rangle \leftarrow \langle \lambda_1 \cdot \lambda_1(v_1|_{z=1}), \lambda_1 \cdot \tau_1(v_1|_{z=1}) + \tau_1, v_1|_{z=1} \rangle$
  $\langle \lambda_{w_1}, \tau_{w_1}, w_1 \rangle \leftarrow$MULTIPLICATION$(\langle \lambda_{0,1}, \tau_{0,1}, v_{0,1} \rangle, \langle \lambda_{1,1}, \tau_{1,1}, v_{1,1} \rangle)$

  return **find_or_add**$(z, \langle \lambda_{w_0}, \tau_{w_0}, w_0 \rangle, \langle \lambda_{w_1}, \tau_{w_1}, w_1 \rangle)$
end if

k	NADD Nodes		
	$\lvert f_{\overline{x}} \rvert$	$\lvert f_{\overline{x}} + f_{\overline{y}} \rvert$	$\lvert f_{\overline{x}} \cdot f_{\overline{y}} \rvert$
0	2	3	3
1	3	5	6
2	4	7	11
3	5	9	20
4	6	11	37
5	7	13	70
6	8	15	135
7	9	17	264
8	10	19	521
9	11	21	1034
10	12	23	2059

Table 4.1: Size of the benchmark functions represented by a NADD

the chosen variable ordering is optimal for $f_{\overline{x}} + f_{\overline{y}}$ can easily be seen. The principle for a NADD is that the values are calculated on the way to the drain. Scalar multiplication and translation leave the structure of a NADD function untouched. Whenever the first function reaches the drain no more traversing and calculating is needed in the second function. The number of nodes for the functions $f_{\overline{x}} + f_{\overline{y}}$ is exactly:

$$2 \cdot (k+1) + 1.$$

The multiplication function $f_{\overline{x}} \cdot f_{\overline{y}}$ has still an exponential sized NADD but with a much smaller increasing rate than the ADD. Recall that any ADD representing the multiplication has $\Theta(4^k)$ nodes. The size of this function represented by a NADD can be calculated recursively:

$$|f_{(x_0,\ldots,x_k)} \cdot f_{(y_0,\ldots,y_k)}| = 2 \cdot |f_{(x_0,\ldots,x_{k-1})} \cdot f_{(y_0,\ldots,y_{k-1})}| - k + 1.$$

Thus,

$$|f_{\overline{x}} \cdot f_{\overline{y}}| = 2^{k+1} + k + 1.$$

A detailed comparison between all benchmarks will be given in Chapter 5.

4.4.2. Matrix Representation

The matrix representation with NADDs is similar to the representation with ADDs.

k	NADD Nodes	
	$\lvert\mathbb{1}_k\rvert$	$\lvert H_k\rvert$
2	3	3
4	6	8
8	9	20
16	12	46
32	15	100
64	18	210
128	21	432
256	24	878
512	27	1770
1024	30	3556

Table 4.2: Size of the benchmark matrices represented by a NADD

4.4.2.1. Benchmark

The results for the benchmark matrices are shown in Table 4.2. The identity matrix $\mathbb{1}_k$ can be seen as a Boolean function and thus the normalization has no effect. In this case a NADD has the same size as a SOBDD with negative edges representing the same function.

The Hilbert matrix H_k cannot be represented in a compact way because of the structure of the values.

But the ORANI678 matrix has a more compact NADD representation with 90385 NADD nodes to store the 90158 entries.

4.5. Implementation

The implementation of NADDs for this thesis can be found in [JJS-BDD]. There it can be seen that the realization of NADDs differs a bit from the theoretical point of view. In practice real values cannot be represented exactly. Computers store an approximation of the value. Thus, the equality of two tuples cannot be calculated by the comparison of the exact values. Like in many other applications this problem can be solved by defining tolerance values.

For that reason two NADD-tuples $\langle \lambda_0, \tau_0, w_0 \rangle$ and $\langle \lambda_1, \tau_1, w_1 \rangle$ are called ε-equal if for given $\lambda_\varepsilon > 0$ and $\tau_\varepsilon > 0$:

$$(|\lambda_0 - \lambda_1| \leq \lambda_\varepsilon) \wedge (|\tau_0 - \tau_1| \leq \tau_\varepsilon) \wedge (w_0 = w_1).$$

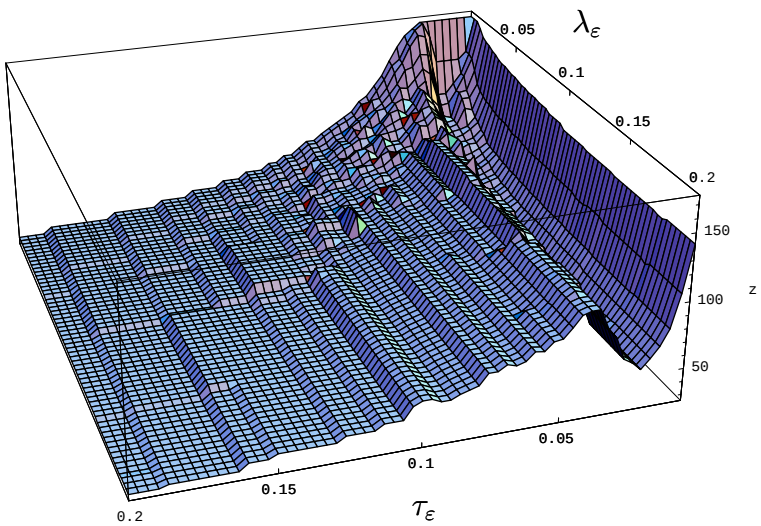


Figure 4.5: Number of nodes for the 32×32 Hilbert matrix with different tolerance values

4.5.1. Influence of λ_ε and τ_ε

λ_ε and τ_ε have influence on the **find_or_add** function. This means that the number of nodes can vary for different tolerance values.

Figure 4.5 shows the number of nodes for the Hilbert matrix of size 32×32 with different tolerance values. Different τ_ε and λ_ε values are inscribed on the x respectively y-axis. With increasing tolerance values the number of nodes and with that the accuracy of the represented function is decreasing. The error for that function can be measured by the maximum absolute error for all matrix entries. Figure 4.6 shows the influence of different tolerance values on the maximum absolute error. It can be seen that the scaling factor has more influence on the result than the translation. In practice it is reasonable to choose small tolerance values to get the best accuracy. If the values are chosen too small the number of nodes increases without increasing accuracy. This situation causes the loss of uniqueness for too small tolerances.

4.5.2. Storing the node parameters

A NADD node uses much more memory than a normal SOBDD or ADD node. A compact representation of nodes is important to implement NADDs. It is not necessary to store all four values of a node. Every node in a NADD is normalized, i.e.

$$\min\{(\lambda_0 + \tau_0), \tau_0, (\lambda_1 + \tau_1), \tau_1\} = 0$$

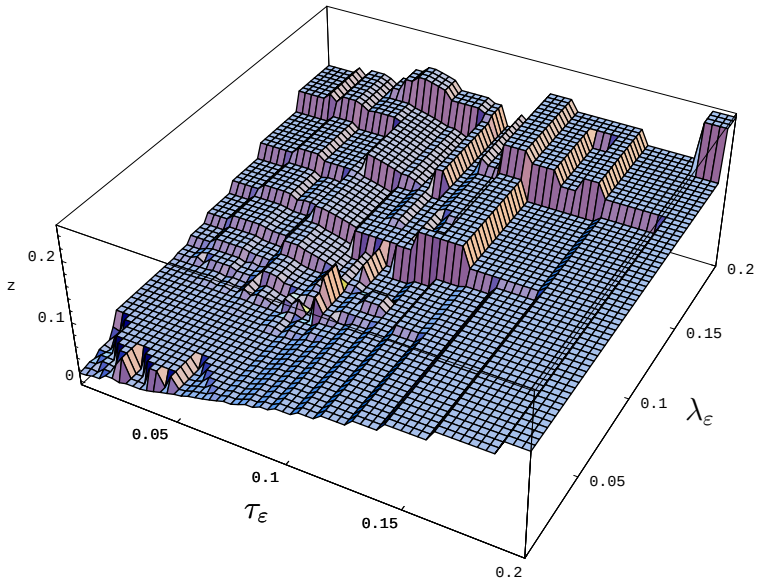


Figure 4.6: The maximum absolute error for the 32×32 Hilbert matrix with different tolerance values

and

$$\max\{(\lambda_0 + \tau_0), \tau_0, (\lambda_1 + \tau_1), \tau_1\} = 1.$$

The information which parameter equals the minimum respectively maximum can be stored in a more compact way than the values. The minimum respectively maximum value is known so that the remaining parameters can be calculated.

5 Experimental Results

This chapter compares the different introduced OBDD variants in relation to size and run-time. The comparison is splitted into algebraic computation and matrix representation.

5.1. Algebraic Computation

The results of the benchmark functions for algebraic computations build the basis for the comparison. ADDs and NADDs represent the function itself whereas SOBDDs store the binary encoding of that function. But both, ADDs and NADDs, could also represent the function by its binary encoding. Thus, there are two comparisons. One between ADDs and NADDs and one between the direct function representation and the binary encoding representation of a function.

k	$\lvert f_{\overline{x}} \rvert$		
	SOBDD	ADD	NADD
0	2	3	2
1	3	7	3
2	4	15	4
3	5	31	5
4	6	63	6
5	7	127	7
6	8	255	8
7	9	511	9
8	10	1023	10
9	11	2047	11
10	12	4095	12

Table 5.1: $f_{\overline{x}}$ represented by all introduced OBDD variants

Table 5.1 shows the sizes of all introduced OBDD variants that represent the function $f_{\overline{x}}$. $f_{\overline{x}}$ has $k+1$ essential variables and thus, the SOBDD and NADD

k	$\lvert f_{\overline{x}} + f_{\overline{y}} \rvert$		
	SOBDD	ADD	NADD
0	4	6	3
1	9	18	5
2	14	44	7
3	19	98	9
4	24	208	11
5	29	430	13
6	34	876	15
7	39	1770	17
8	44	3560	19
9	49	7142	21
10	54	14308	23

Table 5.2: $f_{\overline{x}} + f_{\overline{y}}$ represented by all OBDD variants

representations are optimal. As ADDs have to store the different values in different nodes it cannot be expected that there is a compact ADD representation for any algebraic computation benchmark function.

But not only the size of the OBDD variants is important for a practical application. The time to build a function is also a measure for the quality of a data structure.

The SOBDD representation for $f_{\overline{x}}$ can be constructed in $\Theta(k)$, while ADDs and NADDs have to calculate the function. But the times to construct these functions are too small[1] to compare them.

Now we turn to the addition $(f_{\overline{x}} + f_{\overline{y}})$ represented by the OBDD variants. The size of these OBDD representations are shown in Table 5.2.

The SOBDD and NADD size is linear in the number of variables. The NADD representation is optimal because $2(k+1)+1$ nodes are needed to represent the $2(k+1)$ essential variables. As mentioned before it was expected that ADDs are not capable to represent the function $f_{\overline{x}} + f_{\overline{y}}$.

The times to build $f_{\overline{x}} + f_{\overline{y}}$ can be seen in Table 5.3. The construction with ADDs is much slower than with the other variants which is due to the number of nodes. NADDs calculate much more than SOBDDs and ADDs while traversing a graph but the additional effort is compensated by the additional terminal cases.

The last remaining algebraic benchmark function emphasizes the differences between the introduced OBDD variants most plainly. Table 5.4 shows the

[1] $\sim$ 10 milliseconds

k	$f_{\overline{x}} + f_{\overline{y}}$		
	SOBDD	ADD	NADD
4	0.04	0.03	0.03
5	0.05	0.05	0.03
6	0.07	0.13	0.03
7	0.08	0.42	0.03
8	0.11	1.66	0.03
9	0.13	6.47	0.04
10	0.17	29.22	0.04

Table 5.3: Times to build $f_{\overline{x}} + f_{\overline{y}}$ measured in seconds

k	$\lvert f_{\overline{x}} \cdot f_{\overline{y}} \rvert$		
	SOBDD	ADD	NADD
0	3	6	3
1	12	24	6
2	45	93	11
3	153	352	20
4	475	1377	37
5	1511	5358	70
6	4674	21078	135
7	14558	83203	264
8	45054	329908	521
9	139404	1308670	1034
10	429911	5199280	2059

Table 5.4: $f_{\overline{x}} \cdot f_{\overline{y}}$ represented by OBDD variants

representation of $f_{\overline{x}} \cdot f_{\overline{y}}$ with the OBDD variants. ADDs and SOBDDs grow exponentially in $2k$ variables. The structure of $f_{\overline{y}}$ does not change while building $f_{\overline{x}} \cdot f_{\overline{y}}$ with NADDs, so that NADDs grow exponentially in k variables.

k	$f_{\overline{x}} \cdot f_{\overline{y}}$		
	SOBDD	ADD	NADD
4	< 1	< 1	< 1
5	1	< 1	< 1
6	5	< 1	< 1
7	21	2	< 1
8	87	7	< 1
9	498	33	< 1
10	2202	144	< 1

Table 5.5: Times to build $f_{\overline{x}} \cdot f_{\overline{y}}$ measured in seconds

The times to build this function can be seen in Table 5.5. The invariance of scalar multiplication reduces the construction time for NADDs immensely. The representation of $f_{\overline{x}} \cdot f_{\overline{y}}$ with SOBDDs is much smaller than with ADDs, but more time is needed to build the SOBDD representation. At first sight this might be surprising, but with a closer look it can be seen that the i-th bit needs $i - 1$ lower bits to be calculated. This explains the smaller size but much higher computation time.

The mentioned numerical errors for NADDs do not appear for these benchmark functions. The reason for this is that the functions were build in a way such that the used calculations do not affect the already calculated sub NADDs. The represented domain is also responsible for this result.

5.2. Matrix Representation

The matrix representation is the main purpose for ADDs. NADDs were mainly introduced for algebraic computation. But the ADD size of a function is an upper bound for NADDs, so that NADDs should also be suitable for matrix representation.

For matrix representation the variable ordering is interleaved. With this ordering the advantages of NADDs[2] cannot be used to full capacity. The main drawback of matrix representation with NADDs is that numerical errors can influence all values of a matrix.

Table 5.6 shows the sizes for $\mathbb{1}_k$ and H_k represented by ADDs and NADDs.

[2] terminal cases and compact representation

k	$\lvert\mathbb{1}_k\rvert$		$\lvert H_k\rvert$	
	ADD	NADD	ADD	NADD
2	5	3	6	3
4	8	6	18	8
8	11	9	44	20
16	14	12	98	46
32	17	15	208	100
64	20	18	430	210
128	23	21	876	432
256	26	24	1770	878
512	29	27	3560	1770
1024	32	30	7142	3556

Table 5.6: Sizes of the benchmark matrices

The identity matrix has a compact OBDD representation. The different ADD and NADD sizes can be explained by negative edges. As shown in Lemma 4.2.9 and Theorem 4.2.11 only f or $1 - f$ can be represented by a NADD node. Thus, the NADD representation of Boolean functions is equivalent to SOBDDs with negative edges. The structure of the Hilbert matrix H_k is not well suited for OBDD representation. A matrix with block structure can have a compact OBDD representation. This can be seen through the cofactors of a matrix. Equivalent cofactors can be represented by the same OBDD node. NADDs provide a more compact matrix representation of H_k but the numerical errors mentioned cannot be ignored[3].

The last benchmark matrix is an example for sparse matrices as they often appear in practice. The ADD representation with 472187 nodes is much worse than the NADD representation with 90385 nodes. Numerical errors still appear in this example but are not of great influence.

The time to build a matrix representation with ADDs or NADDs is almost equal.

[3]see Figure 4.6

6 Conclusion and Perspective

In this thesis a new OBDD variant for discrete function representation was introduced. The concept of negative edges on SOBDDs has been expanded to obtain a more compact representation. The main idea was to normalize the cofactors of the function, so that various represented functions could share the same subgraphs. With some restrictions NADDs were shown to be canonical.

Some observations from various OBDD variants can still be used while others do no longer hold. The size of a NADD still depends on the given variable ordering, but not on the size of the represented domain. The nature of how the values are spreaded over the domain is relevant for the NADD size. This could be seen in the case of algebraic computation with defined benchmark functions.

The great advantage of NADDs shows when they are used for algebraic computation. For this application they serve much better than SOBDDs and ADDs. A very useful observation is that an optimal variable ordering for all tested operations can be given. The special properties of NADDs[1] reduce the number of necessary calculations and thus, lower the maximum absolute size of numerical errors. The computation time to build a NADD representing a given function is much smaller than for SOBDDs and ADDs. When NADDs are used for the representation of switching functions they have the same topology than SOBDDs with negative edges and thus, the same size.

The matrix representation with NADDs has some drawbacks. An entry of a matrix usually depends on the variables for the rows and the columns, so that the function that represents the matrix cannot be split into two functions with disjoint variables. For this reason, the interleaving of the variables has been emphasized to be a good variable ordering. But with this ordering the advantages of NADDs cannot be used. Much more calculation has to be done which increases the maximum absolute size of possible numerical errors and the runtime to build the matrix representation. The numerical problem with NADDs should be investigated more deeply to find methods to reduce these errors. One possible attempt could be the use of a normalization domain which depends on the given problem. Another idea could be to find variable

[1] the invariance of scalar multiplication and translation

orderings that reduce the number of calculations and thus, the sizes of the errors. Reordering methods on NADDs have to be defined to obtain this. With the introduction of these new methods it should be examined for which further applications NADDs could be used efficiently.

The representation of sparse matrices seems to have a more compact NADD representation than ADDs. It has been exposed that a compact NADD size results in manageable numerical errors. Further comparisons of other OBDD variants, between NADDs and for example EVBDDs and BMDs, should be performed to get more information about the quality of NADDs.

List of Figures

1.1. Binary Decision Tree 2

2.1. Decision Tree and a corresponding Binary Decision Diagram . . 5
2.2. BDD violating the variable ordering condition 6
2.3. BDD violating the Read Once condition 6
2.4. Reduction rules applied from bottom to top level 9
2.5. Reduction rules applied from top to bottom level 9
2.6. Conjunction of two ROBDD functions creates redundant nodes . 12
2.7. Before and after the application of the negating function 15
2.8. The same function represented by ROBDDs with different variable orderings 16
2.9. Construction of an OBDD representing a symmetric function . . 18
2.10. Switching the drains influences all functions 20
2.11. Transformation rules for negative edges 22

3.1. ADD function for $f_{\overline{x}}$ with $k = 2$ 35
3.2. Structure and city plot of ORANI678 37

4.1. Bottom-up definition of a function represented by a (λ, τ)-ADD 41
4.2. The zero drain does not ensure uniqueness 44
4.3. Modifying the parameters without changing the function 46
4.4. Different reduced normalized (λ, τ)-ADDs representing the same function 47
4.5. Number of nodes for the 32×32 Hilbert matrix with different tolerance values 61
4.6. The maximum absolute error for the 32×32 Hilbert matrix with different tolerance values 62

List of Tables

1.1. Truth Table 1

2.1. Complexity of operators on ROBDDs 15
2.2. Complexity for different function classes 17
2.3. Complexity of operators on SOBDDs 21
2.4. Complexity of operators on different OBDD variants 26
2.5. Size of the benchmark functions represented by a SOBDD . . . 29

3.1. Size of the benchmark functions represented by an ADD 34
3.2. Size of the benchmark matrices represented by an ADD 37

4.1. Size of the benchmark functions represented by a NADD 59
4.2. Size of the benchmark matrices represented by a NADD 60

5.1. $f_{\overline{x}}$ represented by all introduced OBDD variants 63
5.2. $f_{\overline{x}} + f_{\overline{y}}$ represented by all OBDD variants 64
5.3. Times to build $f_{\overline{x}} + f_{\overline{y}}$ measured in seconds 65
5.4. $f_{\overline{x}} \cdot f_{\overline{y}}$ represented by OBDD variants 65
5.5. Times to build $f_{\overline{x}} \cdot f_{\overline{y}}$ measured in seconds 66
5.6. Sizes of the benchmark matrices 67

Bibliography

[Akers 1978] S. B. Akers, *Binary decision diagrams*, IEEE Transactions on Computer Design C-27, pp 509-516, 1978

[Bahar 1993] R. I. Bahar, E. A. Frohm, C. M. Goana, E. Maciii, A. Pardo, F. Somenzi, *Algebraic Decision Diagrams and their Applications*, Proceedings on the International Conference on Computer Aided Design, 1993

[Baier 2002] C. Baier, *Binäre Entscheidungsgraphen*, Skript zur Vorlesung, University of Bonn, 2002

[Bollig 1996] B. Bollig, I. Wegner, *Improving the Variable Ordering of OBDDs is NP-Complete*, IEEE Transactions on Computers, Volume 45, 1996

[Bryant 1986] R. E. Bryant, *Graph-Based Algorithms for Boolean Function Manipulation*, IEEE Transactions on Computers - Volume C-35, 1986

[Bryant 1991] R. E. Bryant, *On the complexity of VLSI implementations and graph representations of boolean functions with applications to integer multiplication*, IEEE Transactions on Computers - Volume 40, 1991

[Bryant 1992] R. E. Bryant, *Symbolic Boolean Manipulation with Ordered Binary Decision Diagrams*, ACM Computing Surveys - Volume 24, 1992

[Clarke 1996] E. Clarke, M. Fujita, X. Zhao, *Multi-Terminal Binary Decision Diagrams and Hybrid Decision Diagrams*, Representations of Discrete Functions, Kluwer Academic Publishers, pp 93-108, 1996

[Cook 1971] S. A. Cook, *The Complexity of Theorem-Proving Procedures*, Proceedings of the Third Annual ACM Symposium on the Theory of Computing, 1971

[Corman 1990] T. H. Corman, C. E. Leiserson, R. L. Rivest, *Introduction to Algorithms*, MIT Press, pp 837, 1990

[Friedman 1990] S. J. Friedman, K. J. Supowit, *Finding the Optimal Variable Ordering for Binary Decision Diagrams*, IEEE Transactions on Computers, Volume 39, 1990

[JJS-BDD] http://www.jjs-bdd.de

[Lai 1992] Y. T. Lai, S. Sastry, *Edge-valued binary decision diagrams for multi-level hierarchical verification*, In Proceedings of the 29th Conference on Design Automation, pp 608-613, IEEE Computer Society Press, 1992

[Lee 1959] C. Y. Lee, *Representation of switching circuits by binary-decision programs*, Bell System Technical Journal 38, pp 985-999, 1959

[Matrix Market] http://math.nist.gov/MatrixMarket/

[Minato et al 1990] S. Minato, N. Ishiura, S. Yajima, *Shared binary decision diagrams with attributed edges for efficient Boolean function manipulation*, Proceedings of the 27th ACM/IEEE Design Automation Conference, ACM, New York, pp 52-57, 1990

[Tafertshofer 1997] P. Tafertshofer, M. Pedram, *Factored Edge-Valued Binary Decision Diagrams*, Formal Methods in System Design: An International Journal, Volume 10, Kluwer Academic Publishers, pp 243-270, 1997

[Tani 1993] S. Tani, K. Kamaguchi, *The Complexity of the Optimal Variable Ordering Problem of Shared Binary Decision Diagrams*, Proceedings of the 4th International Symposium on Algorithms and Computation, 1993

Index

(λ, τ)-ADD, 40
    function, 41
    normalized, 44
0-Scalar, 55, 57
H_k, 36
$\mathbb{1}_k$, 36
$f_{\overline{x}}$, 28
$f_{\overline{x}} + f_{\overline{y}}$, 28
$f_{\overline{x}} \cdot f_{\overline{y}}$, 28

ADD, 31
    function, 32
assignment, 3
attributed edges, 20, 39, 43

cofactor, 4

elimination rule, 9
Eval($\mathcal{Z}$), 3
evaluation, 3

find_or_add, 10, 24, 32, 55
find_or_add_drain, 32

isomorphism rule, 9

$\mathbb{K}$-function, 3

NADD, 44
    0-Scalar, 49
    λ_0-Positivity, 49
    λ_1-Positivity, 49
    τ_0-Arrangement, 49
    reduced, 47
negative edges, 21
node
    non-terminal, 4
    successor, 4
    terminal node, 6
normalization function, 46

OBDD, 6
    (λ, τ)-ADD, 40
    ADD, 31
    NADD, 44
    ROBDD, 8
    SOBDD, 19
one drain, 21, 43
ORANI678, 36

Read Once condition, 6
reduced, 8
reduction rules, 8, 24
ROBDD, 8

SAT, 14
scalar factor λ, 39
shannon expansion, 7
SOBDD, 19
switching functions, 4
symmetric, 17

test for equality, 13, 14, 20
translation τ, 39

uniqueness, 8
universal, 5

variable ordering, 4

zero drain, 43

Declaration

Eidesstattliche Erklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet sowie Zitate kenntlich gemacht habe.

Affidavit

I herewith declare, in lieu of oath, that I have prepared this paper on my own, using only the materials (devices) mentioned. Ideas taken, directly or indirectly, from other sources, are identified as such.

Bonn, 4th November 2004

Jörn Ossowski