

# Boolean Expression Diagrams <sup>\*</sup>

(Submitted to Information and Computation)

Henrik Reif Andersen and Henrik Hulgaard

Department of Information Technology, Building 344

Technical University of Denmark

DK-2800 Lyngby, Denmark

e-mail: {hra,henrik}@it.dtu.dk

## Abstract

This paper presents a new data structure called *Boolean Expression Diagrams* (BEDs) for representing and manipulating Boolean functions. BEDs are a generalization of Binary Decision Diagrams (BDDs) which can represent any Boolean circuit in linear space and still maintain many of the desirable properties of BDDs. Two algorithms are described for transforming a BED into a reduced ordered BDD. One is a generalized version of the BDD *apply*-operator while the other can exploit the structural information of the Boolean expression. This ability is demonstrated by verifying that two different circuit implementations of a 16-bit multiplier implement the same Boolean function. Using BEDs, this verification problem is solved in less than a second, while using standard BDD techniques this problem is infeasible. Generally, BEDs are useful in applications, for example tautology checking, where the end-result as a reduced ordered BDD is small.

## 1 Introduction

Within the last decade *Reduced Ordered Binary Decision Diagrams* (OBDDs<sup>1</sup>) introduced by Bryant [3] have become a successful data structure for representing and manipulating Boolean functions. This success is due to the fact that OBDDs are canonical (making testing of functional properties such as satisfiability and equivalence straightforward) and that they are compact for many Boolean functions occurring in practice. However, the applicability of OBDDs depends heavily on the size of the representation and unfortunately some (important) functions, e.g., the multiplication function, have no sub-exponential representation.

This paper presents an extension of OBDDs, called Boolean Expression Diagrams (BEDs). BEDs can represent any Boolean circuit (see, e.g., [2]) in linear space and still maintain many of the desirable properties of OBDDs. This is obtained by extending the OBDD representation with *operator vertices*:

**Definition 1 (Boolean Expression Diagram)** *A Boolean Expression Diagram (BED) is a directed acyclic graph  $G = (V, E)$  with vertex set  $V$  and edge set  $E$ . The vertex set  $V$  contains three types of vertices: terminal, variable, and operator vertices.*

---

<sup>\*</sup>This work is supported by the Danish Technical Research Council

<sup>1</sup>Throughout this paper, we will assume that all decision and expression diagrams are reduced and will omit the 'R'-prefix.

- A terminal vertex  $v$  has as attribute a value  $value(v) \in \{0, 1\}$ .
- A variable vertex  $v$  has as attributes a variable  $var(v)$ , and two sons  $low(v), high(v) \in V$ .
- An operator vertex  $v$  has as attributes a binary Boolean operator  $op(v)$ , and two sons  $low(v), high(v) \in V$ .

The edge set  $E$  is defined by

$$E = \{(v, low(v)), (v, high(v)) \mid v \in V \text{ and } v \text{ is not a terminal vertex}\}.$$

We use  $\mathbf{0}$  and  $\mathbf{1}$  to denote the two terminal vertices.

Variable vertices correspond to the *if-then-else* operator  $x \rightarrow f_1, f_0$  defined by

$$x \rightarrow f_1, f_0 = (x \wedge f_1) \vee (\neg x \wedge f_0).$$

Operator vertices correspond to their respective Boolean connectives, leading to the following correspondence between BEDs and Boolean functions.

**Definition 2** A vertex  $v$  in a BED denotes a Boolean function  $f^v$  defined recursively as:

- If  $v$  is a terminal vertex, then  $f^v = value(v)$ .
- If  $v$  is a variable vertex, then  $f^v$  is the function

$$f^v = var(v) \rightarrow f^{high(v)}, f^{low(v)}.$$

- If  $v$  is an operator vertex, then  $f^v$  is the function

$$f^v = f^{low(v)} \ op(v) \ f^{high(v)}.$$

Before presenting the formal details of BEDs and the algorithms for manipulating them, we illustrate the use of the data structure to prove a tautology.

## 1.1 A simple example

Consider verifying that conjunction distributes over disjunction, i.e., that the following is a tautology:

$$x_1 \wedge (x_2 \vee x_3) \leftrightarrow (x_1 \wedge x_2) \vee (x_1 \wedge x_3). \quad (1)$$

The BED for this expression is shown in figure 1. The low-edges are drawn using dashed lines. Notice that vertices representing the same Boolean sub-expressions are shared. A key operation on BEDs is the *up-step* which moves a variable vertex up above an operator vertex. Let  $op$  be an arbitrary binary Boolean operator, let  $x$  be a Boolean variable, and let  $f_i$  and  $f'_i$  ( $i = 0, 1$ ) be arbitrary Boolean expressions. It is simple to verify that

$$(x \rightarrow f_1, f_0) \ op \ (x \rightarrow f'_1, f'_0) = x \rightarrow (f_1 \ op \ f'_1), (f_0 \ op \ f'_0). \quad (2)$$

This identity, illustrated in figure 2 (a), is used to move the variable  $x$  above the operator  $op$  and is the basis for the up-step. (Equation (2) also holds if the operator vertex  $op$  is a variable vertex. In that case, the up-step is identical to the level exchange operation typically

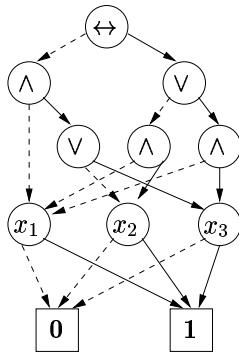


Figure 1: The BED for equation (1).

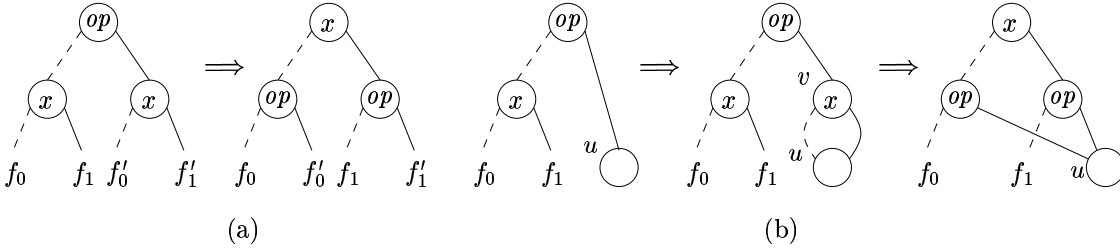


Figure 2: Illustration of the up-step (a) for the case where variable  $x$  exists in both sons of the root and (b) for the case where  $x$  only occurs in the left son.

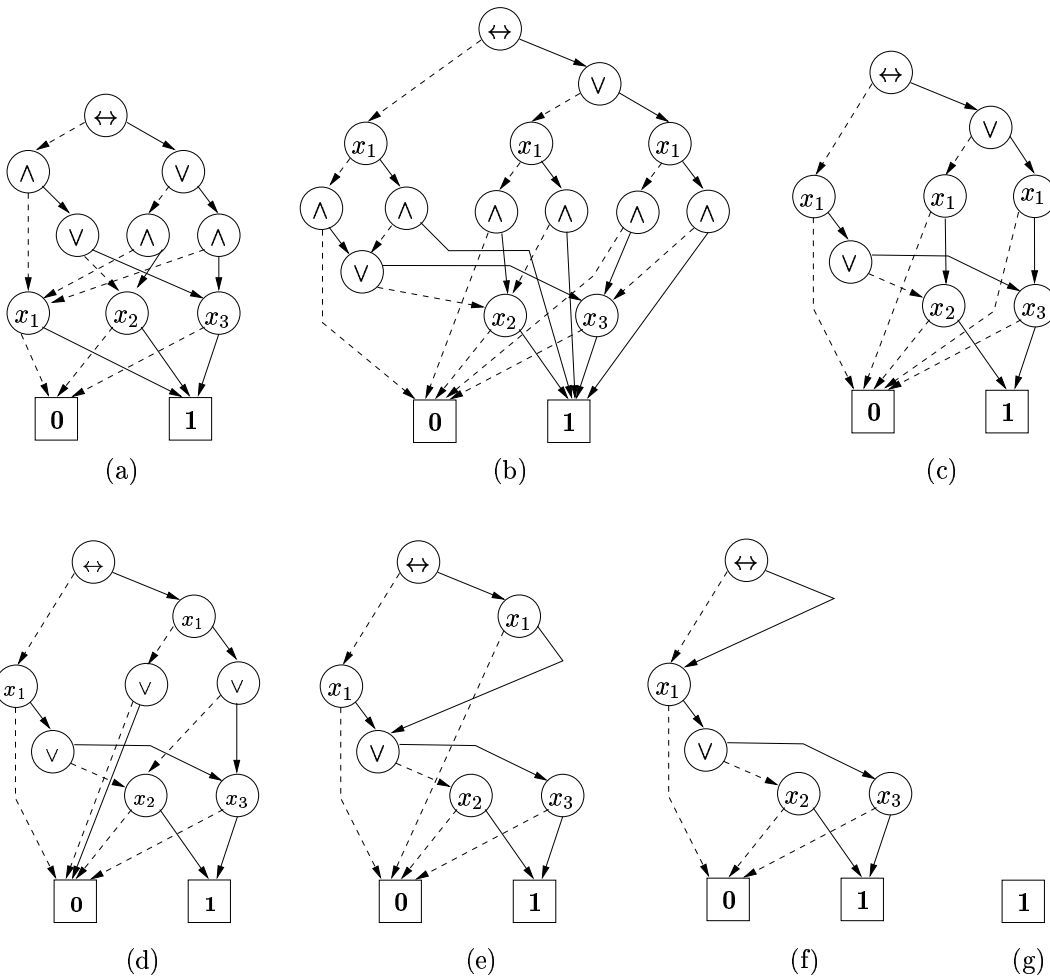
used in OBDDs to dynamically change the variable ordering [18].) In cases where one of the children  $u$  does not contain the variable  $x$ , a new variable vertex  $v$ , with  $\text{var}(v) = x$  and  $\text{low}(v) = \text{high}(v) = u$ , is inserted below the operator vertex before performing the up-step, see figure 2 (b). In fact, this is the only way the size of the BED can increase.

The up-step moves operators closer to the terminal vertices. If some of the expressions  $f_i$  are terminal vertices, the operators are evaluated and the BED simplified. By repeatedly moving variable vertices above operator vertices, all operator vertices are eliminated and the BED is turned into an OBDD.

Consider the example of proving the distributive law (1). Figure 3 shows how the BED from figure 1 is transformed into the tautology **1** by moving  $x_1$  towards the root. This example illustrates that it may not be necessary to move *all* variable vertices to the root in order to obtain an OBDD. Variables  $x_2$  and  $x_3$  could have been replaced with arbitrary large BEDs, and the tautology would have been proven with exactly the same steps.

The example illustrates one way to convert a BED to an OBDD, moving the variables to the top one at a time. This approach is called *up\_one* and its main advantage is that it can exploit structural information in the expression (as was the case in the example). The efficiency of *up\_one* is demonstrated in section 5 where we verify that two different circuit implementations of a 16-bit multiplier are identical.

An alternative way to construct an OBDD is to move all variables up simultaneously. This approach is called *up\_all* and it is a generalization of the OBDD *apply*-operation. We show that the complexity of building an OBDD bottom up using *apply* (the standard way) and building it from a BED using *up\_all* is within a constant factor. Thus, one can construct an OBDD from a BED as efficiently as constructing an OBDD from scratch.



**Figure 3:** Proving the distributive law. (a) BED for the distributive law. (b)  $x_1$  is moved above the three conjunctions using three up-steps. Notice that at this point variable and operator vertices are no longer separated in two distinct layers. (c) Conjunctions with children that are constant vertices are eliminated. (d)  $x_1$  is moved above the disjunction to the right. (e) The disjunction with both children equal to  $\mathbf{0}$  is removed and the two remaining disjunctions are identified. (f) Identifying equivalent variable vertices. At this point the two children of the biimplication operator are identical and (g) the BED is reduced to  $\mathbf{1}$ , proving the tautology.

## 1.2 Related work

Recently, a new way of constructing OBDDs, called MORE, was proposed [12, 13]. MORE is based on the observation that the OBDD for  $f \vee g$  can be constructed by introducing a new variable  $x$  and implicitly existentially quantify  $x$  since  $\exists x. x \rightarrow f, g = f \vee g$ . MORE constructs the OBDD by moving  $x$  towards the terminal vertices using the level exchange operation [9]. The method can be extended to any Boolean connective since disjunction and negation are functionally complete. BEDs can be seen as extending this idea to allow arbitrary operators and allowing these operators to remain in the graph. Furthermore, the algorithms presented here for manipulating BEDs are new. Like MORE, Extended BDDs [15] are also based on the idea of using existential quantification to represent disjunction, although the quantification is annotated on the edges of the graph. Extended BDDs are more succinct than OBDDs, but they are not capable of representing for example mul-

tipliers efficiently.

OBDDs have been extended in a number of other ways, including using other types of decomposition rules, relaxing the variable ordering restrictions, and extending the domains. The Shannon decomposition used in OBDDs can be replaced with either the positive or the negative Davio decomposition, yielding Ordered Functional BDDs [16]. If all three types of decomposition are allowed in one diagram, one obtains Ordered Kronecker Functional Decision Diagrams (OKFDD) [8]. However, none of them are powerful enough to represent all Boolean circuits in polynomial space.

Another modification of the OBDD representation is to relax the variable ordering restriction. Free BDDs [11] (also called read-once branching programs) only require that on any path from the root, a variable is tested at most once. BEDs are exponentially more succinct than Free BDDs since BEDs are as succinct as branching programs which are exponentially more succinct than read-once branching programs [20]. Graph-driven BDDs [19] are closely related to Free BDDs and have similar properties.

Indexed BDDs [14] extends OBDDs with *layers* of variables. The variables in each layer are ordered but different layers may have different orderings. Thus Indexed BDDs are not canonical and are as expressive as branching programs. This makes it possible to represent, e.g., the multiplication function efficiently (using sufficiently many layers). Indexed BDDs is a sub-class of BEDs since BEDs make no assumptions about the ordering or freeness of variables.

Finally, OBDDs have been extended to other domains and/or codomains than Booleans. Examples include \*BMDs [4], MTBDDs [6] and ADDs [1]. These extensions are orthogonal to the OBDD extension presented here and we believe similar extensions are possible for BEDs.

### 1.3 Overview

The paper is organized as follows. Section 2 presents some basic complexity results relating BEDs to Boolean circuits and OBDDs. Section 3 describes the basic representation and construction of BEDs. Section 4 describes algorithms to efficiently manipulate BEDs, including two ways to construct an OBDD from a BED, *up\_one* and *up\_all*. Section 5 presents an application of BEDs, demonstrating efficient tautology check for a multiplication circuit. Finally, section 6 summarizes the contributions of this paper.

## 2 Complexity results

BEDs are closely related to Boolean circuits [2]. Any circuit can be transformed to a BED by replacing each input  $x$  with the BED representing  $x$  (a variable vertex  $v$  with  $var(v) = x$ ,  $low(v) = \mathbf{0}$ , and  $high(v) = \mathbf{1}$ ) and replace each  $k$ -input gate by a tree of  $k - 1$  operator vertices encoding the Boolean function of the gate. This translation is clearly linear in size. Similarly, any BED can be converted to a circuit. Each variable occurring in the BED is an input to the circuit. An operator vertex is replaced with the corresponding gate, and a variable vertex  $v$  is replaced with the sub-circuit  $(\neg x \wedge l) \vee (x \wedge h)$ , where  $x = var(v)$ ,  $l = low(v)$ ,  $h = high(v)$ . This translation is also linear.

Using this relationship we can transfer results on circuits to BEDs. For instance, it follows immediately from the results on CIRCUIT-SAT that determining SATISFIABILITY of a BED is NP-complete and determining TAUTOLOGY is co-NP-complete [10]. As another

consequence, we observe that BEDs are *exponentially more succinct* than OBDDs. An example of this is the multiplier function. Bryant [3] showed that for all variable orderings, the multiplier function requires BDDs of exponential size. However, since there are combinational circuits implementing this function using only a quadratic number of gates [7] (and even less), there exists a BED of this size representing it.

Despite the exponential succinctness over BDDs, it is still the case that most functions require exponentially sized BEDs. Recall that there are  $2^{2^n}$  Boolean functions over  $n$  variables. It follows from a counting argument that a polynomially sized BED can represent almost none of these functions:

**Theorem 3 (Lower bound on size)** *Let  $\#_n(s)$  be the number of different BEDs over  $n$  variables with at most  $s$  vertices. Then for any polynomial  $p(n)$ ,*

$$\frac{\#_n(p(n))}{2^{2^n}} \rightarrow 0 \quad \text{for } n \rightarrow \infty.$$

**Proof:** A straightforward application of Theorem 2.4 in [2, p.763] using the linear transformation to circuits.  $\square$

Fortunately, functions with exponentially sized BEDs do not seem to be of much interest in practice. Even complicated Boolean functions, representing for instance floating-point division, have polynomially sized circuits. This is also witnessed by the fact that it is very difficult to construct explicit examples of functions that provably require exponentially many gates. (The authors have been unable to find any examples in the literature.)

Inspired by OBDDs, we define certain restrictions on the variables of BEDs:

**Definition 4** *A BED is free if on all paths through the graph each variable occurs at most once; it is ordered if on all paths the variables respect a given total order  $<$ .*

We refer to a free BED as FBED and to an ordered BED as OBED. Observe that an (O)BDD is simply an (O)BED without operators. A vertex  $u$  is an *OBDD vertex* if all paths from  $u$  only contains variable vertices and they respect a given total order. From these definitions we get the following inclusions among sub-classes of BEDs:

$$\text{OBDD} \subseteq \text{DAG of OBDDs} \subseteq \text{OBED} \subseteq \text{FBED} \subseteq \text{BED}.$$

The class ‘‘DAG of OBDDs’’ represents BEDs that consist of a layer of operators on top of a layer of OBDDs. Boolean circuits that are transformed into BEDs belong in this class (in this case, the OBDDs are very simple, each consisting of a single variable). Furthermore, this class occurs in the traditional synthesis of OBDDs, where the operators represent *apply-calls*. Since Boolean circuits can be transformed into a ‘‘DAG of OBDDs’’ in linear time (and space) and a (general) BED can be transformed into a Boolean circuit in linear time (and space), the last four classes are equally expressive. This is quite unlike for OBDDs where there is an exponential gap between OBDDs and free BDDs, and between free BDDs and BDDs. Although the different classes of BEDs are equally expressive, the algorithms on BEDs (to be presented in the following) can be optimized if the BEDs are known to belong to some of the more specific classes. In section 4, we describe two algorithms for transforming a (general) BED into an OBDD and some optimizations of these algorithms for the classes OBEDs and FBEDs. The generality of these algorithms make them useful, e.g., for making a free BDD ordered (i.e., transforming it to an OBDD) or for reordering an OBDD.

### 3 Representation of BEDs

BED vertices are constructed using a single operation called *mk*. This operation ensures that the BED is *reduced* and also performs several optimizations of the representation. Contrary to OBDDs, reducedness will not make BEDs canonical (not even when combined with a fixed variable ordering.)

#### 3.1 Reductions of BEDs

We shall forbid the existence of *redundant* vertices, i.e., two vertices representing isomorphic sub-BEDs and vertices that are unnecessary for obvious reasons. For readability, we use  $\alpha(v)$  to denote the “tag” *op(v)* or *var(v)* on non-terminal vertices.

**Definition 5** *A BED is reduced if it contains at most two different terminal vertices and for all non-terminal vertices,  $u$  and  $v$ :*

- (i)  $low(u) = low(v), high(u) = high(v), \text{ and } \alpha(u) = \alpha(v) \Rightarrow u = v,$
- (ii)  $low(u) \neq high(u),$

and for all operator vertices  $v$ :

- (iii)  $low(v)$  and  $high(v)$  are non-terminals.

We shall assume that BEDs are always reduced. The first condition of definition 5 is fulfilled by proper reuse of vertices. This is conveniently taken care of during construction of a BED by testing, whenever a new vertex is to be created, whether another vertex with the same variable/operator, low- and high-sons exists. If this is the case, that vertex is reused otherwise a new vertex is created. Similarly, the second and third conditions are fulfilled by never constructing vertices that violate them. For variable vertices, it is clear that if the low- and high-sons coincide, either one of them can be used instead of creating a new variable vertex. For operator vertices, one should observe that if the two arguments are identical, or one of them is a terminal vertex, all the sixteen Boolean connectives (shown in table 1) reduce to one of the following six:  $K0, K1$  (constant 0/1),  $\pi_1, \pi_2$  (projection onto first or second argument),  $\bar{\pi}_1, \bar{\pi}_2$  (the negation of the first or second argument). In the first two cases, one of the terminal vertices is used. The projections are avoided by using the proper low- or high-son instead. The negations require creation of a negating vertex, i.e., an operator vertex with the operator  $\bar{\pi}_1$ . Such a vertex can easily be constructed so that it fulfills (ii) and (iii) by taking the redundant second argument to be any non-terminal vertex different from the first. We shall assume the presence of a function

$$mk(\alpha, l, h)$$

that performs all the checks above and returns the identity of the resulting vertex, equivalent to a vertex  $u$  with  $\alpha(u) = \alpha, low(u) = l, high(u) = h$ . Using *mk* as the only means for constructing a BED ensures that it is reduced. As shown by Bryant, reducedness ensures canonicity of OBDD vertices:

**Lemma 6 (Canonicity of OBDDs [3])** *If  $u$  and  $v$  are OBDD vertices and  $f^u = f^v$  then  $u = v$ .*

**Table 1:** The sixteen binary Boolean operators and their associated truth-table.

	$op$	$op(x, y)$	Name of Boolean function
		$\begin{array}{cccc} x & 1 & 1 & 0 & 0 \\ y & 1 & 0 & 1 & 0 \end{array}$	
0	$K0$	0000	Constant false
1	$\bar{\vee}$	0001	Negated disjunction
2	$\not\leftarrow$	0010	Negated left-implication
3	$\bar{\pi}_1$	0011	Negation of first argument
4	$\not\rightarrow$	0100	Negated implication
5	$\bar{\pi}_2$	0101	Negation of second argument
6	$\not\leftrightarrow$	0110	Exclusive or
7	$\bar{\wedge}$	0111	Negated conjunction
8	$\wedge$	1000	Conjunction
9	$\leftrightarrow$	1001	Biimplication
10	$\pi_2$	1010	Projection on second argument
11	$\rightarrow$	1011	Implication
12	$\pi_1$	1100	Projection on first argument
13	$\leftarrow$	1101	Left-implication
14	$\vee$	1110	Disjunction
15	$K1$	1111	Constant true

### 3.2 Operator reductions

For operator vertices one can add more checks in order to reuse vertices, thereby reducing the size of the BED. An immediate optimization is to extend  $mk$  to look for operator vertices that differ from the one wanted only by exchanging low and high, by a negation, or by a combination of both.

Going a step further, considering two vertices at a time, we can eliminate all negations below binary operators since for all binary operators  $op$  there exists another operator  $op'$  with  $op'(x, y) = op(\neg x, y)$ .

Finally, taking the identity of vertices into account allows us to exploit equivalences like the *absorption laws*, e.g.,  $x \vee (x \wedge y) = x$ . There are  $16^n$  combinations of  $n$  binary Boolean operators, thus it is feasible to tabulate them all for  $n$  up to three or four. Choosing  $n = 3$  seems like a natural choice since such a reduction table would include equivalences such as the distributive laws.

Another reason for choosing  $n = 3$  is that it allows us to determine *operator 2-cuts*: Consider a BED with the structure shown in figure 4 (a), that is, for some vertex  $u$ , all paths from  $u$  to the terminals go through either vertex  $w_1$  or  $w_2$ . We call the set  $\{w_1, w_2\}$  a 2-cut. 2-cuts can be used to reduce the size of the BED as shown in figure 4 (b).

More formally, consider a BED represented by the graph  $(V, E)$ . A *path* of length  $k$  from a vertex  $u \in V$  to a vertex  $u' \in V$  is a sequence  $\langle v_0, v_1, v_2, \dots, v_k \rangle$  such that  $u = v_0$ ,  $u' = v_k$ , and  $(v_{i-1}, v_i) \in E$  for  $i = 1, 2, \dots, k$ . A path  $p$  from  $u$  to  $u'$  is denoted by  $u \xrightarrow{p} u'$ .

**Definition 7 (2-cut)** *A set  $\{w_1, w_2\} \subseteq V$  is a 2-cut for vertex  $u \in V \setminus \{w_1, w_2\}$  if any path  $p$  from  $u$  to a terminal vertex  $u' \in \{0, 1\}$ , can be decomposed into two parts  $u \xrightarrow{p_1} w \xrightarrow{p_2} u'$  such that  $w \in \{w_1, w_2\}$ . A 2-cut is called an *operator 2-cut* if for all paths  $p$ ,  $p_1$  only contains operator vertices. The cut  $\{low(u), high(u)\}$  is called a *trivial 2-cut* for the vertex  $u$ .*

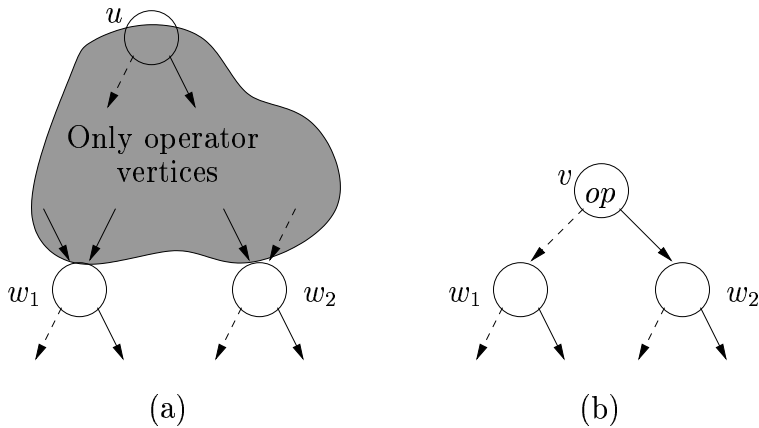


Figure 4: A BED with an operator 2-cut  $\{w_1, w_2\}$ .

If the BED rooted at  $u$  has an operator 2-cut  $\{w_1, w_2\}$ , then there exists a binary Boolean operator  $op$  such that

$$f^u = f^{w_1} op f^{w_2}.$$

That is, all vertices from  $u$  to the 2-cut can be replaced with a single operator vertex  $v$  with  $\alpha(v) = op$  and  $low(v) = w_1$  and  $high(v) = w_2$ . The BED can be constructed such that it contains no non-trivial operator 2-cuts. The following lemma shows that a new operator vertex only will have non-trivial 2-cuts among its children and grandchildren.

**Lemma 8** *Let  $u \in V$  be an operator vertex and let  $l = low(u)$  and  $h = high(u)$  be the low- and high-sons of  $u$ . If  $l$  and  $h$  only have trivial operator 2-cuts, then if  $u$  has any non-trivial operator 2-cut  $\{w_1, w_2\}$ , it is a subset of  $\{l, h, low(l), high(l), low(h), high(h)\}$ .*

**Proof:** By contradiction: assume  $u$  has a non-trivial operator 2-cut  $\{w_1, w_2\}$  which is not a subset of  $\{l, h, low(l), high(l), low(h), high(h)\}$ .

Observe that either  $l$  or  $h$  is not in  $\{w_1, w_2\}$  (otherwise the 2-cut would be trivial). Assume without loss of generality that  $l \notin \{w_1, w_2\}$ . Since  $\{w_1, w_2\}$  is a 2-cut for  $u$ , every path from  $u$  to a terminal vertex contains either  $w_1$  or  $w_2$  (or both). Any path from  $l$  to a terminal vertex also contains either  $w_1$  or  $w_2$  since the path is a postfix of some path from  $u$ . Thus,  $\{w_1, w_2\}$  is an operator 2-cut for  $l$  which can not be trivial since, by assumption,  $\{w_1, w_2\}$  is not a subset of  $\{l, low(l), high(l)\}$ . This is a contradiction.  $\square$

From this lemma it follows that non-trivial cuts only exists among the children and grandchildren of  $u$ . A reduction table constructed with  $n = 3$  makes it easy to find and eliminate all non-trivial 2-cuts when constructing the BED.

### 3.3 Implementation aspects

The data structures for representing BEDs, shown in figure 5, is very similar to those for BDDs. The underlying graph of the BED is stored in a table  $G$  which to each vertex  $v$  associates a tag  $\alpha(v)$  (special tags are used for the terminal vertices),  $low(v)$ , and  $high(v)$ . Furthermore,  $G$  contains the field  $next(v)$  which is used to implement chaining for resolving collisions in a hash table  $H$ . This hash table maps triples of  $(\alpha(v), low(v), high(v))$  to  $v$  and thus implements an inverse of  $G$  used by  $mk$ . The total memory requirements are five words per vertex. Using these data structures, it is not difficult to implement  $mk$  as an expected constant time operation on a uniform RAM model.

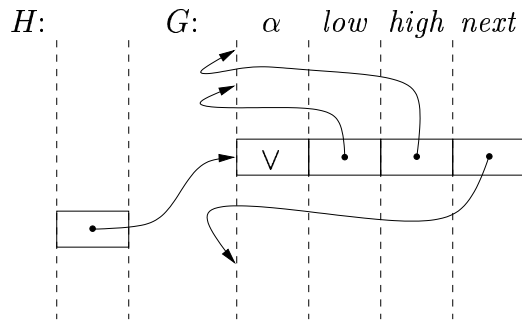


Figure 5: The data structures used to represent a BED.

## 4 Operations on BEDs

The basic operation for constructing OBDDs, called *apply*, takes two OBDDs,  $l$  and  $h$ , and a Boolean connective  $op$  and constructs a new OBDD representing the Boolean expression  $f^l op f^h$ . For BEDs, constructing the representation for the Boolean expression  $f^l op f^h$  is simply a constant time call to  $mk(op, l, h)$ . However, other operations, like checking for tautology or satisfiability, are difficult for BEDs but are constant time operations for OBDDs. Thus, an approach for implementing these operations on BEDs is to convert the BEDs into OBDDs. Since an (O)BDD is simply an (O)BED without operators, a strategy for converting BEDs into OBDDs is to gradually eliminate the operators, keeping all the intermediate BEDs functionally equivalent. We shall show two very different ways of elimination.

### 4.1 Construction of OBDDs with *up\_one*

The first elimination algorithm is based on the algorithm *up\_one* shown in figure 6. *Up\_one* pulls a single variable up to the root by performing a recursive depth-first traversal of the BED and after the recursive calls on the low- and high-sons of a vertex, it makes an up-step. Repeated calls to *up\_one* for each variable moves all variables up past the operators, which makes the operators disappear (by requirement (iii) of reducedness). The function  $mk'(x, l, h)$  is a version of  $mk$ , which further checks for duplicate variables. If any of  $l$  and  $h$  has  $x$  at the root, it is removed:

$$mk'(x, l, h) = \begin{cases} mk(x, low(l), high(h)) & \text{if } \alpha(l) = \alpha(h) = x \\ mk(x, low(l), h) & \text{if } \alpha(l) = x, \alpha(h) \neq x \\ mk(x, l, high(h)) & \text{if } \alpha(l) \neq x, \alpha(h) = x \\ mk(x, l, h) & \text{if } \alpha(l) \neq x, \alpha(h) \neq x \end{cases}$$

The table  $M$  is used to memorize previously computed results and ensures a linear expected runtime.

The introductory example was in fact a use of *up\_one*. As the example shows, in fortunate cases a BED is converted into an OBDD after moving just a few variables up (in the example, one variable was sufficient). In this process, identical sub-BEDs, potentially containing operator vertices, are identified. This is quite unlike traditional OBDD construction where all operators are converted in depth-first order into OBDDs. In particular, an OBDD is constructed for *each sub-expression*. If the result is small and the intermediate OBDDs are large, *up\_one* is an attractive alternative.

```

up_one(x, u) =
1:  if (x, u) in M then return M(x, u)
2:  else if u is a terminal then return u
3:  else
4:    (l, h) ← (up_one(x, low(u)), up_one(x, high(u)))
      /* x can only occur in the roots of l and h */
5:    if α(l) and α(h) are both variable x then
6:      r ← mk'(x, mk(α(u), low(l), low(h)),
                  mk(α(u), high(l), high(h)))
7:    else if α(l) is variable x then
8:      r ← mk'(x, mk(α(u), low(l), h),
                  mk(α(u), high(l), h))
9:    else if α(h) is variable x then
10:     r ← mk'(x, mk(α(u), l, low(h)),
               mk(α(u), l, high(h)))
11:   else
12:     r ← mk(α(u), l, h)
13:   insert ((x, u), r) in M
14:   return r

```

**Figure 6:** The *up\_one*-operation. *Up\_one* takes any BED  $u$  (which could be ordered, free, or completely arbitrary) as argument and returns an equivalent BED with  $x$  occurring at most at the root. The memorization table  $M$  must be initialized to empty prior to the first call.

The following theorem characterizes key properties of *up\_one*. Let  $|u|$  denote the number of vertices in the BED reachable from  $u$ :

$$|u| = |\{v : u \rightsquigarrow v\}|.$$

**Theorem 9 (Up\_one)** *Assume  $u$  is a vertex in a BED and let  $v = \text{up\_one}(x, u)$ . The following properties hold:*

- (i)  $f^v = f^u$ .
- (ii)  $x$  does not occur below  $v$ .
- (iii)  $|v| \leq 2|u| - 1$ .
- (iv) If  $u$  is ordered (free) then  $v$  is also ordered (free).
- (v) *Up\_one* can be implemented with expected running time  $O(n)$  with  $n = |u|$  on a uniform RAM model using hashing.

**Proof:** We prove (i) by induction on the number of vertices below  $u$ . In the proof we ignore the memorization table  $M$ , which speeds up the algorithm but does not influence its correctness provided only correct values are inserted into it. The base case in line 2, when  $u$  is a terminal, clearly satisfy  $f^v = f^u$  since  $v = u$ . For the induction step assume that

$$f^l = f^{\text{low}(u)} \text{ and } f^h = f^{\text{high}(u)}.$$

There are now four cases to consider, corresponding to the four branches. We consider only the first case since the other three are similar (or simpler). We shall use the following

properties of  $mk$  and  $mk'$ :

$$\begin{aligned} f^{mk(op,l,h)} &= f^l \text{ op } f^h \\ f^{mk(x,l,h)} &= x \rightarrow f^h, f^l \\ f^{mk'(x,l,h)} &= x \rightarrow f^h, f^l. \end{aligned}$$

From the assignment of  $r$  in line 6:

$$\begin{aligned} f^r &= x \rightarrow f^{mk(\alpha(u),high(l),high(h))}, f^{mk(\alpha(u),low(l),low(h))} \\ &= x \rightarrow (f^{high(l)} \alpha(u) f^{high(h)}), (f^{low(l)} \alpha(u) f^{low(h)}) \\ &= (x \rightarrow f^{high(l)}, f^{low(l)}) \alpha(u) (x \rightarrow f^{high(h)}, f^{low(h)}) \\ &= f^l \alpha(u) f^h \\ &= f^{low(u)} \alpha(u) f^{high(u)} \\ &= f^u. \end{aligned}$$

The first two steps use the properties of  $mk'$  and  $mk$ . The third step is an up-step. The fourth step uses the semantics of variable vertices. The fifth step uses the induction hypothesis. The final step uses the definition of the semantics of an operator node  $\alpha(u)$ . The proof is similar for a variable node.

Property (ii) is also proven by induction on the number of vertices below  $u$ . The base case is trivial and in the induction step the four cases corresponding to the four branches follow directly by the induction hypothesis (which states that  $x$  does not occur below  $l$  and  $h$ ) and the definition of  $mk'$ .

For property (iii) observe that, due to the table  $M$ , the body of  $up\_one$  is executed at most once for each node below  $u$ . Let  $|u|_x$  be the number of vertices labeled with variable  $x$  below  $u$ :

$$|u|_x = |\{v : u \rightsquigarrow v \text{ and } v \text{ is a variable vertex with } var(v) = x \}|.$$

Let  $n = |u|$  and  $n_x = |u|_x$ . There are  $n - n_x$  vertices with a label different from  $x$ . We shall first count only the new vertices with labels different from  $x$  that are being generated by the  $mk'$  and  $mk$  calls in  $up\_one$ . For each call to  $up\_one$  on vertices  $u$  with  $\alpha(u) \neq x$ , at most two new vertices with label different from  $x$  are generated (in line 6, 8, 10, and 12). This gives a total of at most  $2(n - n_x)$  new vertices. From property (ii) it follows that in the result  $v$  at most one  $x$ -vertex can occur, i.e., the total number of vertices reachable from  $v$  is

$$|v| \leq 2(n - n_x) + 1 = 2n + 1 - 2n_x.$$

For  $n_x \geq 1$ , we have that  $|v| \leq 2n - 1$ . When  $n_x = 0$ , all calls to  $up\_one$  will fall into the last branch (line 12) and  $v = u$ , thus clearly  $|v| \leq 2n - 1$ .

Property (iv) is also proven by an induction on the number of vertices below  $u$ . We prove that if  $u$  is ordered the BED  $v$  is still ordered. The proof for freeness is similar. Assume that  $x_1 < \dots < x_n$  is an ordering for  $u$ . If  $x = x_i$ , the ordering for  $v$  is  $x_i < x_1 < \dots < x_{i-1} < x_{i+1} < \dots < x_n$ . The induction hypothesis for a vertex  $u$  is that  $up\_one(x, u)$  has the new ordering and that  $up\_one(x, u)$  does not contain variables that were not already present in  $u$ . The base case is trivial. For the induction step we can assume from the induction hypothesis that  $l$  and  $h$  both have the new ordering. We consider only the first of the four

branches. The remaining three are similar or simpler. If  $\alpha(u)$  is an operator, then clearly  $r$  has the new ordering. If  $\alpha(u)$  is a variable different from  $x$ , then  $\alpha(u)$  must precede the variables in  $low(u)$  and  $high(u)$  according to the old ordering. Therefore  $\alpha(u)$  also precedes any variables of  $l$  and  $h$  in the new ordering. If  $\alpha(u)$  is the variable  $x$ , from the induction hypothesis,  $x$  also precedes the variables below  $l$  and  $h$ . From property (ii), variable  $x$  can only occur at the root of  $l$  and  $h$  and thus  $mk'$  ensures that  $x$  can only occur at the root of  $r$ .

For property (v) observe again that due to the table  $M$  the body of  $up\_one$  is executed at most once on each node below  $u$ . Insertion and searching in  $M$  can be performed in expected constant time on a uniform RAM model using hashing. Similarly,  $mk$  and  $mk'$  can be performed in expected constant time due to the hash table  $H$ . Thus, the expected running time for each call to  $up\_one$  is constant. This yields an overall running time of  $O(n)$ , where  $n = |u|$ .  $\square$

The analysis of  $up\_one$  above makes no assumptions about how the variables occur in the BED. However, if the BED is known to be at least free (see section 2), several optimizations can be performed. Although these optimization do not improve on the bounds given in the theorem above, they reduce the actual runtime of the algorithm. The first optimization is that  $mk'$  can be replaced with the slightly simpler  $mk$  when the BED is free. The second optimization is based on the observation that if  $u$  is a vertex with variable  $x$ , the recursive calls in line 4 and the following tests can be omitted. This is done by adding the following line

$2\frac{1}{2}$ : **else if**  $\alpha(u)$  is variable  $x$  **then return**  $u$

after line 2.

$Up\_one$  is a very versatile algorithm. In section 4.3 we show how  $up\_one$  is used to implement substitution and existential quantification in BEDs. Here we show how it is used to transform an arbitrary BED into an OBDD. Let  $x_1 < \dots < x_n$  be the variable ordering of the OBDD. The OBDD with root  $v$  is constructed by calling  $up\_one$  for each variable in this ordering:

$$v \leftarrow up\_one(x_1, up\_one(x_2, \dots up\_one(x_n, u) \dots)).$$

These calls pull up the variables in reverse order, that is, first  $x_n$  is pulled to the root, then  $x_{n-1}$  is pulled up and so on. This is clearly inefficient since each variable has to pass through all the variables that have already been pulled up.

A more efficient approach is to pull up the variables in order and instead of pulling a variable all the way to the root each time, it is only pulled up until it reaches a variable which precedes it in the ordering. This is done by modifying  $up\_one$  by adding the lines

$4\frac{1}{3}$ : **if**  $\alpha(u)$  is a variable with  $var(u) < x$  **then**  $r \leftarrow mk(\alpha(u), l, h)$   
 $4\frac{2}{3}$ : **else**

after line 4. Let  $up\_one'$  be this modified version of  $up\_one$ . Then we construct the OBDD by

$$v \leftarrow up\_one'(x_n, up\_one'(x_{n-1}, \dots up\_one'(x_1, u) \dots)).$$

The runtime of this computation is exponential in the worst case, even though  $up\_one$  has linear runtime and it is called only  $n$  times. However, since BEDs are provably exponentially

more succinct than OBDDs, any transformation algorithm will have exponential worst-case behavior.

## 4.2 Construction of OBDDs with `up_all`

The second elimination algorithm, `up_all`, is a generalization of Bryant's `apply`-operator, shown in figure 7. Construction of OBDDs from a Boolean expression using recursive calls

```

apply(op, l, h) =
  if (l, h) in M then return M(l, h)
  else if l and h are terminals then
    r ← op(value(l), value(h))
  else if var(l) = var(h) then
    r ← mk(var(l), apply(op, low(l), low(h)),
           apply(op, high(l), high(h)))
  else if var(l) < var(h) then
    r ← mk(var(l), apply(op, low(l), h),
           apply(op, high(l), h))
  else var(l) > var(h) :
    r ← mk(var(h), apply(op, l, low(h)),
           apply(op, l, high(h)))
  insert ((l, h), r) in M
  return r

```

**Figure 7:** The `apply`-operation. Assumes  $l$  and  $h$  are OBDDs. The imposed total order on the variable vertices is denoted  $<$ . In the code it is assumed that terminal vertices are included at the end of this order when comparing  $var(l)$  and  $var(h)$ . The memorization table  $M$  must be initialized to empty prior to the first call.

of `apply` suggests a bottom up conversion of BEDs into OBDDs. The `up_all` algorithm does that by moving all variables up as a block past the operator vertices. `Up_all` is shown in figure 8. As when building an OBDD using `apply`, `up_all` requires that a total ordering  $<$  of the variables is selected prior to the OBDD construction. Based on the ordering `up_all` converts any BED into an OBDD. Key properties of `up_all` are:

**Theorem 10 (Up\_all)** *Assume  $u$  is a vertex in a BED and  $x_1 < \dots < x_n$  is an ordering of the variables, and let  $v = up\_all(u)$ . The following properties hold:*

- (i)  $f^v = f^u$ .
- (ii)  $v$  is an OBDD.
- (iii) If  $l$  and  $h$  are OBDDs, then  $apply(op, l, h) = up\_all(mk(op, l, h))$ .
- (iv) If  $l$  and  $h$  are OBDDs, the running time of  $up\_all(op, l, h)$  is expected  $O(|l||h|)$  on a uniform RAM model using hashing.

**Proof:** Let  $|u|_{-OBDD}$  be the number of vertices below  $u$  that are not OBDD vertices according to the given ordering:

$$|u|_{-OBDD} = |\{v : u \rightsquigarrow v \text{ and } v \text{ is not an OBDD vertex}\}|.$$

Property (i) and (ii) are proven simultaneously by well-founded induction on the lexicographical ordering  $\prec$  of the measure

$$(|u|_{-OBDD}, |u|).$$

```

up_all(u) =
1:  if  $u$  in  $M$  then return  $M(u)$ 
2:  else if  $u$  is a terminal then return  $u$ 
3:  else
4:     $(l, h) \leftarrow (up\_all(low(u)), up\_all(high(u)))$ 
      /*  $l$  and  $h$  are OBDDs */
5:    if  $l$  and  $h$  are terminal vertices then
6:       $r \leftarrow mk(\alpha(u), l, h)$ 
7:    else if  $\alpha(u)$  is a variable  $x$  with  $x \leq var(l)$  and  $x \leq var(h)$  then
8:       $r \leftarrow mk'(x, l, h)$ 
9:    else if  $var(l) = var(h)$  then
10:      $r \leftarrow mk(var(l), up\_all(mk(\alpha(u), low(l), low(h))),$ 
       $up\_all(mk(\alpha(u), high(l), high(h))))$ 
11:   else if  $var(l) < var(h)$  then
12:      $r \leftarrow mk(var(l), up\_all(mk(\alpha(u), low(l), h)),$ 
       $up\_all(mk(\alpha(u), high(l), h)))$ 
13:   else  $var(l) > var(h)$  :
14:      $r \leftarrow mk(var(h), up\_all(mk(\alpha(u), l, low(h))),$ 
       $up\_all(mk(\alpha(u), l, high(h))))$ 
15:   insert  $(u, r)$  in  $M$ 
16:   return  $r$ 

```

**Figure 8:** The  $up\_all$ -operation. The total order  $<$  is defined as for  $apply$  (see figure 7). The memorization table  $M$  must be initialized to empty prior to the first call.

As in the proof of theorem 9 we ignore the memorization table  $M$ . The induction hypothesis is that  $f^w = f^{up\_all(w)}$  and  $up\_all(w)$  is an OBDD for all vertices  $w \prec u$ . The base case, when  $u$  is a terminal, clearly fulfills property (i) and (ii). For the induction step, we first argue that  $low(u) \prec u$  and  $high(u) \prec u$ . If  $u$  is already an OBDD with the given variable ordering,  $low(u)$  is also an OBDD with the given ordering, and the first part of the measure is zero for both. However, the sizes of both  $low(u)$  and  $high(u)$  are less than  $u$  and thus  $low(u) \prec u$  and  $high(u) \prec u$ . If  $u$  is not an OBDD with the given variable ordering, then  $low(u)$  and  $high(u)$  contains at least one non-OBDD vertex less than  $u$ , showing that  $low(u) \prec u$  and  $high(u) \prec u$ .

This allows us to use the induction hypothesis to conclude that  $f^l = f^{low(u)}$  and  $f^h = f^{high(u)}$  and that  $l$  and  $h$  are OBDDs. If  $l$  and  $h$  are both terminals, the **if**-branch in line 5 is chosen and property (i) and (ii) clearly hold for  $r$ . If  $\alpha(u)$  is a variable  $x$  smaller than the variables in  $l$  and  $h$ , the **if**-branch in line 7 is chosen and  $r$  fulfills (i) and (ii) using the fact that  $mk'$  will remove duplicate occurrences of  $x$ .

For the remaining three **if**-branches, we first argue that the arguments of the recursive calls are before  $u$  in the order  $<$ . We consider only the first of the three branches since the remaining two are similar. The argument of the first recursive call in line 10 is

$$w = mk(\alpha(u), low(l), low(h))$$

thus we must show that  $w \prec u$ . Since  $l$  and  $h$  are OBDDs, the number of non-OBDDs vertices in  $w$  is at most one, i.e.,  $|w|_{\text{-OBDD}} \leq 1$ . If we reach this **if**-branch,  $\alpha(u)$  is either a variable larger than  $l$  and  $h$ , or  $u$  is an operator vertex. Therefore,  $u$  can not be an OBDD-vertex and  $|u|_{\text{-OBDD}} \geq 1$ . Thus, either the measure decreases in its first component, i.e.,

$|w|_{\text{-OBDD}} < |u|_{\text{-OBDD}}$ , or it is the case that  $|u|_{\text{-OBDD}} = |w|_{\text{-OBDD}} = 1$ . In the first case we immediately have that  $w \prec u$ . In the second case,  $u$  contains only one non-OBDD vertex and this must be in the root, i.e.,  $u$ . Therefore  $low(u)$  and  $high(u)$  are already OBDDs with the given variable ordering. The canonicity of OBDDs together with the fact that  $f^l = f^{low(u)}$  implies that  $l = low(u)$  and similarly  $h = high(u)$ . The size of  $w$  is therefore strictly less than that of  $u$  and  $w \prec u$ .

A similar argument allows us to conclude that the measure is also decreasing in the second recursive call. We can then rewrite from the definition of  $r$  and use the induction hypothesis and the semantics of  $mk$  to obtain the following equivalences (assuming  $\alpha(u)$  is an operator, the steps are similar for a variable):

$$\begin{aligned}
f^r &= var(l) \rightarrow f^{mk(\alpha(u), high(l), high(h))}, f^{mk(\alpha(u), low(l), low(h))} \\
&= var(l) \rightarrow (f^{high(l)} \alpha(u) f^{high(h)}), (f^{low(l)} \alpha(u) f^{low(h)}) \\
&= (var(l) \rightarrow f^{high(l)}, f^{low(l)}) \alpha(u) (var(l) \rightarrow f^{high(h)}, f^{low(h)}) \\
&= f^l \alpha(u) f^h \\
&= f^{low(u)} \alpha(u) f^{high(u)} \\
&= f^u.
\end{aligned}$$

This completes the proof of properties (i) and (ii).

Property (iii) is shown by induction on the size of  $|u|$  using the induction hypothesis  $apply(\alpha(u), low(u), high(u)) = up\_all(u)$  where  $\alpha(u)$  is an operator and  $low(u)$  and  $high(u)$  are OBDDs. From (ii) and the canonicity of OBDDs, it follows that  $up\_all(u) = u$  if  $u$  is an OBDD. Thus, the calls to  $up\_all$  in line 4 just returns the arguments, i.e.,  $l = low(u)$  and  $h = high(u)$ . When  $\alpha(u)$  is an operator, the **if**-branch in line 7 is never taken and it is then clear how the four cases of  $apply$  correspond exactly to the remaining four cases of  $up\_all$ .

Property (iv) is shown by bounding the number of elements inserted into  $M$ . By induction one can show that for all pairs  $(w, r)$  inserted in  $M$ , one of the following two cases hold:

- $r = w$  and either  $low(u) \rightsquigarrow w$  or  $high(u) \rightsquigarrow w$ , or
- $w$  is an operator vertex with  $op(w) = \alpha(u)$ ,  $low(u) \rightsquigarrow low(w)$ , and  $high(u) \rightsquigarrow high(w)$ .

Thus, the number of elements in  $M$  is bound by  $|low(u)| + |high(u)| + |low(u)||high(u)|$  and since the body of  $up\_all$  takes constant time, the total runtime is  $O(|low(u)||high(u)|)$ .  $\square$

The worst-case runtime of  $up\_all$  is exponential in  $|u|$ . For the same reason as  $up\_one$ , this is optimal.

As presented above,  $up\_all$  makes no assumptions about how the variables occur in the BED. However,  $up\_all$  can be optimized for several special cases:

**Free BEDs:** The call to  $mk'$  can be replaced by  $mk$  in line 8.

**Ordered BEDs:** The condition of the **if**-statement in line 7 can be simplified to

7': **if**  $\alpha(u)$  is a variable  $x$  **then**

since  $x < var(l)$  and  $x < var(h)$  in an ordered BED.

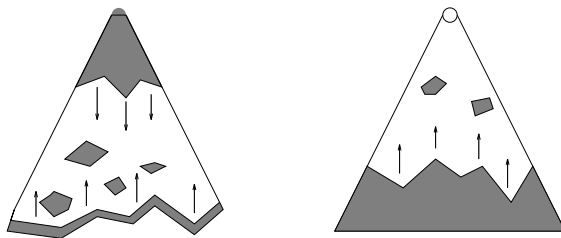
**DAG of OBDDs:** The condition of the **if**-statement in line 2 can be relaxed to

2': **if**  $u$  is a terminal or  $\alpha(u)$  is a variable **then return**  $u$

since if  $\alpha(u)$  is a variable,  $u$  is an OBDD.

**An operator on OBDDs:** If  $u$  is an operator vertex such that  $low(u)$  and  $high(u)$  are OBDDs, line 4 can be replaced with  $(l, h) \leftarrow (low(u), high(u))$  since  $up\_all(w) = w$  when  $w$  is an OBDD. Furthermore, the test in line 7 is always false, thus in this case there is a one-to-one correspondence between  $up\_all$  and  $apply$ . The number of calls to  $mk$  on variables generated by  $up\_all(mk(op, l, h))$  is exactly the same as for  $apply(op, l, h)$ , and the number of calls to  $mk$  on operators is the same as the number of calls to  $apply(op, l, h)$ .

$Up\_one$  and  $up\_all$  are significantly different strategies for building an OBDD. Figure 9 illustrates how the operator vertices are converted into variable vertices by the two algorithms.



**Figure 9:** Converting a BED to an OBDD. To the left, using  $up\_one$  repeatedly, to the right, during an  $up\_all$  call. Grey areas represent variables and white areas represent operators.

### 4.3 Further BED operations

Two commonly used Boolean operations are substitution and existential quantification. Substitution replaces all occurrences of a variable  $x$  with a Boolean function  $f$ . The simplest way to perform substitution on a BED rooted at  $v$  is the following. First perform a call to  $up\_one$ :

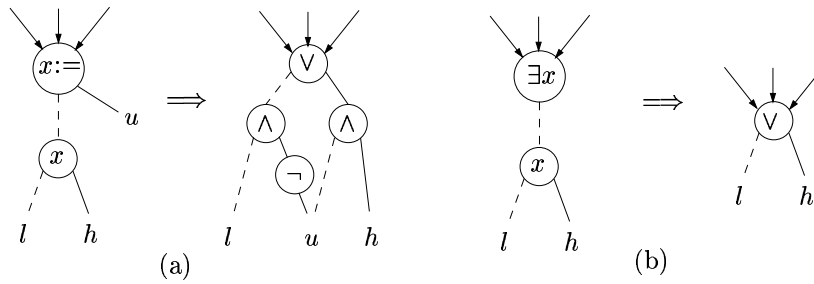
$$w \leftarrow up\_one(x, v).$$

From theorem 9 we know that  $low(w)$  and  $high(w)$  do not contain any occurrences of variable  $x$ . If  $w$  is not the variable  $x$ , then  $x$  is not in the BED rooted at  $w$  and the result is  $w$ . Otherwise,  $var(w) = x$  and the result is the BED for

$$(u \wedge high(w)) \vee (\neg u \wedge low(w))$$

where  $u$  is the root of the BED representing  $f$ . This expression follows immediately from the definition of a variable vertex.

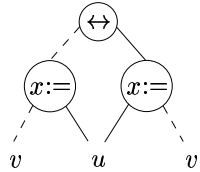
Existential quantification of the variable  $x$  can also be implemented using  $up\_one$ . Again we call  $up\_one$  obtaining  $w$  and if  $w$  does not contain  $x$  the result is  $w$ . Otherwise, the result is  $low(w) \vee high(w)$  since  $\exists x. x \rightarrow f, g = f \vee g$ . For both operations, the complexity is determined by  $up\_one$  which is linear in the size of the BED.



**Figure 10:** (a) Elimination of substitution vertex. (b) Elimination of existential quantification vertex.

An alternative way to implement substitution and existential quantification is to consider them special operator vertices in the BED, see figure 10. The up-step from figure 2 is exactly the same for these new operator vertices, except in the case where the variable below the operator is the variable  $x$ . In those cases, the special operator vertex is replaced with the sub-BED shown in figure 10. These eliminations can easily be performed by adding reduction rules to *mk*. The substitution and existential quantification operators can be eliminated like any other operator in the BED by pulling the variables up past the operators. An operator is eliminated either when it meets a corresponding variable or when it reaches terminal vertices.

One need not immediately eliminate these newly added operator vertices. Keeping them in the BED allows efficient reuse of sub-expressions. Consider the BED in figure 11. If the



**Figure 11:** A BED containing substitution operators. The low-edge points to the expression in which  $x$  is to be substituted with  $f^u$ , pointed to by the high-edge.

vertices  $v$  and  $v'$  are identified at some point in the manipulations, the biimplication is proven immediately, *without* actually performing the substitution.

Other standard BDD operations include restriction  $x := b$ , where  $b$  is a Boolean constant. Clearly, restriction is a special case of substitution with  $f$  equal to either true or false. Notice that all the operations described in this section have linear running times which is better than the corresponding OBDD operations. Other operations, like satisfy-one, can be performed by constructing an OBDD using *up\_all* and perform the operation on the OBDD. Since *up\_all* never has worse running time than *apply*, the total running time for these operations is no worse than if they are performed directly on an OBDD.

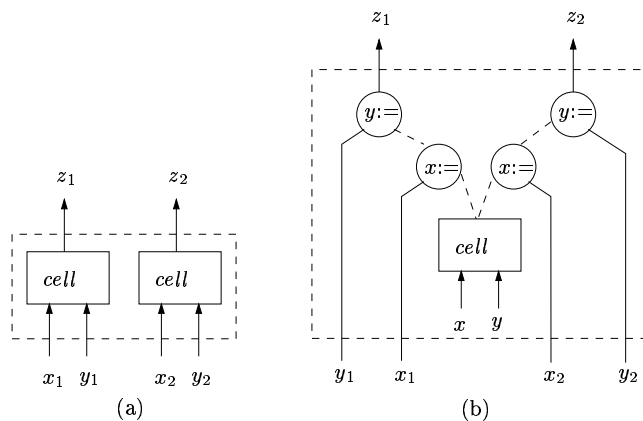
## 5 An application of BEDs

Tautology checking of a BED is an application where the end-result as an OBDD is known to be small (the terminal vertex **1**) if the BED indeed is a tautology. Thus, to demonstrate the efficiency of BEDs, we consider the combinational logic-level verification problem which is to determine whether two given combinational circuits implement the same Boolean function.

We consider two implementations of a 16-bits multiplier (`c6288` and `c6288nr` from the ISCAS-85 benchmarks). Checking whether these two versions implement the same functionality corresponds to a tautology check for each pair of outputs. A BED is built directly from the circuit netlist as explained in section 2. The circuits `c6288` and `c6288nr` contain 2416 and 2399 gates, respectively, and the resulting reduced BED contains 3478 vertices. All 32 outputs are verified using `up_one` in 0.7 CPU seconds on a Sun Ultra-SPARC 1.

This example was chosen since multipliers are notoriously difficult to verify using OBDDs [3]. Due to the exponential growth of the size of the OBDD representation (in the number of operand bits), the straightforward approach of building and comparing the OBDD for the two circuits `c6288` and `c6288nr` is not feasible. The OBDD representation of a 15-bit multiplier uses more than 12 million vertices [17] and this number is approximately 2.7 times larger for each additional bit in the operands. This demonstrates the effectiveness of `up_one`.

Substitution operators can be used to represent a circuit hierarchy. Consider a circuit with two instances of a sub-circuit `cell`, see figure 12 (a). Instead of flattening the hierarchy, the two instances of `cell` are represented in the BED data structure using substitutions, see figure 12 (b). This information can greatly improve the performance when verifying equivalence of large hierarchical circuits. We illustrate this by verifying equivalence between two



**Figure 12:** (a) Two instances of a sub-circuit `cell`. (b) Representing the same circuit using substitutions.

implementations of  $n$ -bit multipliers. A combinational  $n$ -bit multiplier can be constructed using four  $n/2$ -bit multipliers. The four multipliers each compute partial products which are shifted and added to form the result. In this way we build two versions of hierarchically specified  $n$ -bit multipliers based on the 16-bit multipliers `c6288` and `c6288nr`. The BED representing two hierarchically specified 1024-bits multipliers contains 11.3 million vertices, yet the equivalence can be established in only 7 CPU minutes.

## 6 Conclusion

We have presented a new data structure called Boolean Expression Diagrams for representing and manipulating Boolean expressions. BEDs are as succinct as Boolean circuits, yet they have many of the desirable properties of BDDs. Properties like TAUTOLOGY and SATISFIABILITY are determined by transforming the BED representation into a reduced ordered

BDD. This can be done efficiently by using one of the two algorithms *up\_one* or *up\_all*. As shown in theorem 10, the cost of constructing an OBDD from scratch using *apply* and the cost of building a BED and transform it into an OBDD using *up\_all* are within a constant factor. In fact, recent research [13] has shown that *up\_all* is a highly efficient approach to build an OBDD; it uses considerably less memory and no more time than when the OBDD is constructed with *apply*.

*Up\_one* is a new way to construct an OBDD which can exploit structural similarities between sub-expressions. For some applications *up\_one* is highly efficient, for example as demonstrated by proving the identity of two 16-bits multipliers (`c6288` and `c6288nr` from the ISCAS 85 benchmarks) in less than a second. *Up\_one* is also the basis for other operations like existential quantification and substitution, making the running times of these operation linear in the size of the BED.

BEDs are particularly useful when the end-result is expected to have a small OBDD representation, e.g., for tautology checks. Another area that may benefit from using the BED representation is in symbolic model checking. Several researchers have observed that when performing fixed-point iterations using OBDDs, the intermediate results are often much larger than the final result. Clearly, the succinctness of BEDs compared to BDDs can alleviate this problem. This is possible because all operations used in performing the fixed-point computation can be performed directly on the BED without first expanding it to an OBDD. In fact, some of the tricks researchers have used to make OBDDs more efficient are embodied in BEDs. For example, Burch, Clarke, and Long [5] demonstrated that the complexity of BDD-based symbolic verification is drastically reduced by using a *partitioned transition relation* where the transition relation is represented as an implicit conjunction of OBDDs. This corresponds to representing the transition relation as a BED with conjunction vertices at the top level and only lifting the variables up to just under these vertices.

BEDs can be seen as an intermediate form between circuits and the highly structured OBDDs. In this paper we have focussed on ways to obtain an OBDD from the BED description. However, there seems to be a huge unexploited potential for manipulating the BEDs directly, without necessarily converting them to OBDDs. For example, it seems plausible that ideas from logic can be transferred to BEDs; the reduction of operator vertices described in section 3.2 is a first step in this direction.

## References

- [1] R. I. Bahar, E. A. Frohm, C. M. Gaona, G. D. Hachtel, E. Macii, A. Pardo, and F. Somenzi. Algebraic decision diagrams and their applications. In *Proc. International Conf. Computer-Aided Design (ICCAD)*, pages 188–191, 1993.
- [2] R. B. Boppana and M. Sipser. The complexity of finite functions. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume A: Algorithms and Complexity, pages 758–804. Elsevier Science Publisher, 1990.
- [3] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, August 1986.
- [4] R. E. Bryant and Y.-A. Chen. Verification of arithmetic functions with binary moment diagrams. In *Proc. ACM/IEEE Design Automation Conference (DAC)*, pages 535–541, 1995.

- [5] J. R. Burch, E.M. Clarke, and D. E. Long. Representing circuits more efficiently in symbolic model checking. In *Proc. ACM/IEEE Design Automation Conference (DAC)*, pages 403–407, 1991.
- [6] E. M. Clarke, K.L. McMillan, X. Zhao, M. Fujita, and J. Yang. Spectral transforms for large Boolean functions with application to technology mapping. In *Proc. ACM/IEEE Design Automation Conference (DAC)*, pages 54–60, 1993.
- [7] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, 1990.
- [8] R. Drechsler, A. Sarabi, M. Theobald, B. Becker, and M.A. Perkowski. Efficient representation and manipulation of switching functions based on ordered Kronecker functional decision diagrams. In *Proc. ACM/IEEE Design Automation Conference (DAC)*, pages 415–419, 1994.
- [9] M. Fujita, Y. Matsunga, and T. Kakuda. On variable ordering of binary decision diagrams for the application of multi-level synthesis. In *Proc. European Conference on Design Automation (EDAC)*, pages 50–54, 1991.
- [10] M. R. Garey and D. S. Johnson. *Computers and Intractability—A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, 1979.
- [11] J. Gergov and C. Meinel. Efficient Boolean manipulation with OBDD’s can be extended to FBDD’s. *IEEE Transactions on Computers*, 43(10):1197–1209, October 1994.
- [12] A. Hett, R. Drechsler, and B. Becker. MORE: Alternative implementation of BDD-packages by multi-operand synthesis. In *European Design Conference*, 1996.
- [13] A. Hett, R. Drechsler, and B. Becker. Fast and efficient construction of BDDs by reordering based synthesis. In *IEEE European Design & Test Conference*, 1997.
- [14] J. Jain, J. Bitner, M. S. Abadir, J. A. Abraham, and D. S. Fussell. Indexed BDDs: Algorithmic advances and techniques to represent and verify Boolean functions. *IEEE Transactions on Computers*, 46(11):1230–1245, November 1997.
- [15] S.-W. Jeong, B. Plessier, G. Hachtel, and F. Somenzi. Extended BDD’s: Trading off canonicity for structure in verification algorithms. In *Proc. International Conf. Computer-Aided Design (ICCAD)*, pages 464–467, 1991.
- [16] U. Kebschull, E. Schubert, and W. Rosenstiel. Multilevel logic synthesis based on functional decision diagrams. In *Proc. European Conference on Design Automation (EDAC)*, pages 43–47, 1992.
- [17] H. Ochi, K. Yasuoka, and S. Yajima. Breadth-first manipulation of very large binary-decision diagrams. In *Proc. International Conf. Computer-Aided Design (ICCAD)*, pages 48–55, 1993.
- [18] R. Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *Proc. International Conf. Computer-Aided Design (ICCAD)*, pages 42–47, 1993.
- [19] D. Sieling and I. Wegener. Graph driven BDDs – a new data structure for Boolean functions. *Theoretical Computer Science*, 141(1-2):283–310, 1995.
- [20] I. Wegener. On the complexity of branching programs and decision trees for clique functions. *Journal of the ACM*, 35(2):461–471, April 1988.