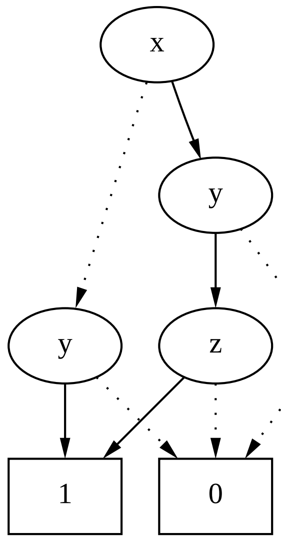
An Introduction to Binary Decision Diagrams

Henrik Reif Andersen



Lecture notes for 49285 Advanced Algorithms E97, October 1997.
(Minor revisions, Apr. 1998)
E-mail: `hra@it.dtu.dk`. Web: `http://www.it.dtu.dk/~hra`

Department of Information Technology, Technical University of Denmark
Building 344, DK-2800 Lyngby, Denmark.

Preface

This note is a short introduction to Binary Decision Diagrams. It provides some background knowledge and describes the core algorithms. More details can be found in Bryant's original paper on Reduced Ordered Binary Decision Diagrams [Bry86] and the survey paper [Bry92]. A recent extension called Boolean Expression Diagrams is described in [AH97].

This note is a revision of an earlier version from fall 1996 (based on versions from 1995 and 1994). The major differences are as follows: Firstly, ROBDDs are now viewed as nodes of one global graph with one fixed ordering to reflect state-of-the-art of efficient BDD packages. The algorithms have been changed (and simplified) to reflect this fact. Secondly, a proof of the canonicity lemma has been added. Thirdly, the sections presenting the algorithms have been completely restructured. Finally, the project proposal has been revised.

Acknowledgements

Thanks to the students on the courses of fall 1994, 1995, and 1996 for helping me debug and improve the notes. Thanks are also due to Hans Rischel, Morten Ulrik Sørensen, Niels Maretti, Jørgen Staunstrup, Kim Skak Larsen, Henrik Hulgaard, and various people on the Internet who found typos and suggested improvements.

Contents

1 Boolean Expressions

The classical calculus for dealing with *truth values* consists of *Boolean variables* $x, y, ...$, the constants *true* 1 and *false* 0, the operators of *conjunction* $\wedge$, *disjunction* $\vee$, *negation* $\neg$, *implication* $\Rightarrow$, and *bi-implication* $\Leftrightarrow$ which together form the Boolean expressions. Sometimes the variables are called *propositional variables* or *propositional letters* and the Boolean expressions are then known as *Propositional Logic*.

Formally, Boolean expressions are generated from the following grammar:

$$t \ ::= \ x \mid 0 \mid 1 \mid \neg t \mid t \wedge t \mid t \vee t \mid t \Rightarrow t \mid t \Leftrightarrow t,$$

where x ranges over a set of Boolean variables. This is called the *abstract syntax* of Boolean expressions. The concrete syntax includes parentheses to solve ambiguities. Moreover, as a common convention it is assumed that the operators bind according to their relative priority. The priorities are, with the highest first: $\neg$, $\wedge$, $\vee$, $\Leftrightarrow$, $\Rightarrow$. Hence, for example

$$\neg x_1 \wedge x_2 \vee x_3 \Rightarrow x_4 = (((\neg x_1) \wedge x_2) \vee x_3) \Rightarrow x_4 \,.$$

A Boolean expression with variables $x_1, \ldots, x_n$ denotes for each assignment of truth values to the variables itself a truth value according to the standard truth tables, see figure 1. *Truth assignments* are written as sequences of assignments of values to variables, e.g., $[0/x_1, 1/x_2, 0/x_3, 1/x_4]$ which assigns 0 to x_1 and x_3, 1 to x_2 and x_4. With this particular truth assignment the above expression has value 1, whereas $[0/x_1, 1/x_2, 0/x_3, 0/x_4]$ yields 0.

	$\neg$
0	1
1	0

$\wedge$	0	1
0	0	0
1	0	1

$\vee$	0	1
0	0	1
1	1	1

$\Rightarrow$	0	1
0	1	1
1	0	1

$\Leftrightarrow$	0	1
0	1	0
1	0	1

Figure 1: Truth tables.

The set of truth values is often denoted $\mathbb{B} = \{0, 1\}$. If we fix an ordering of the variables of a Boolean expression t we can view t as defining a function from $\mathbb{B}^n$ to $\mathbb{B}$ where n is the number of variables. Notice, that the particular ordering chosen for the variables is essential for what function is defined. Consider for example the expression $x \Rightarrow y$. If we choose the ordering $x < y$ then this is the function $f(x, y) = x \Rightarrow y$, true if the first argument implies the second, but if we choose the ordering $y < x$ then it is the function $f(y, x) = x \Rightarrow y$, true if the second argument implies the first. When we later consider compact representations of Boolean expressions, such variable orderings play a crucial role.

Two Boolean expressions t and t' are said to be equal if they yield the same truth value for all truth assignments. A Boolean expression is a *tautology* if it yields true for all truth assignments; it is *satisfiable* if it yields true for at least one truth assignment.

Exercise 1.1 Show how all operators can be encoded using only $\neg$ and $\vee$. Use this to argue that any Boolean expression can be written using only $\vee$, $\wedge$, variables, and $\neg$ applied to variables.

Exercise 1.2 Argue that t and t' are equal if and only if $t \Leftrightarrow t'$ is a tautology. Is it possible to say whether t is satisfiable from the fact that $\neg t$ is a tautology?

2 Normal Forms

A Boolean expression is in *Disjunctive Normal Form (DNF)* if it consists of a disjunction of conjunctions of variables and negations of variables, i.e., if it is of the form

$$(t_1^1 \land t_2^1 \land \cdots \land t_{k_1}^1) \lor \cdots \lor (t_1^l \land t_2^l \land \cdots \land t_{k_l}^l) \tag{1}$$

where each t_i^j is either a variable x_i^j or a negation of a variable $\neg x_i^j$. An example is

$$(x \land \neg y) \lor (\neg x \land y)$$

which is a well-known function of x and y (which one?). A more succinct presentation of (1) is to write it using indexed versions of $\land$ and $\lor$:

$$\bigvee_{j=1}^{l} \left(\bigwedge_{i=1}^{k_j} t_i^j \right) .$$

Similarly, a *Conjunctive Normal Form (CNF)* is an expression that can be written as

$$\bigwedge_{j=1}^{l} \left(\bigvee_{i=1}^{k_j} t_i^j \right)$$

where each t_i^j is either a variable or a negated variable. It is not difficult to prove the following proposition:

Proposition 1 *Any Boolean expression is equal to an expression in CNF and an expression in DNF.*

In general, it is hard to determine whether a Boolean expression is satisfiable. This is made precise by a famous theorem due to Cook [Coo71]:

Theorem 1 (Cook) *Satisfiability of Boolean expressions is NP-complete.*

(For readers unfamiliar with the notion of NP-completeness the following short summary of the pragmatic consequences suffices. Problems that are NP-complete can be solved by algorithms that run in exponential time. No polynomial time algorithms are known to exist for any of the NP-complete problems and it is very unlikely that polynomial time algorithms should indeed exist although nobody has yet been able to prove their non-existence.)

Cook's theorem even holds when restricted to expressions in CNF. For DNFs satisfiability is decidable in polynomial time but for DNFs the tautology check is hard (co-NP complete). Although satisfiability is easy for DNFs and tautology check easy for CNFs, this does not help us since the conversion between CNFs and DNFs is exponential as the following example shows.

Consider the following CNF over the variables $x_0^1, \ldots x_0^n, x_1^1, \ldots, x_1^n$:

$$(x_0^1 \vee x_1^1) \wedge (x_0^2 \vee x_1^2) \wedge \cdots \wedge (x_0^n \vee x_1^n) .$$

The corresponding DNF is a disjunction which has a disjunct for each of the n-digit binary numbers from 000...000 to 111...111 — the i'th digit representing a choice of either x_0^i (for 0) or x_1^i (for 1):

$$\begin{array}{c}
(x_0^1 \wedge x_0^2 \wedge \cdots \wedge x_0^{n-1} \wedge x_0^n) \quad \vee \\
(x_0^1 \wedge x_0^2 \wedge \cdots \wedge x_0^{n-1} \wedge x_1^n) \quad \vee \\
\vdots \\
(x_1^1 \wedge x_1^2 \wedge \cdots \wedge x_1^{n-1} \wedge x_0^n) \quad \vee \\
(x_1^1 \wedge x_1^2 \wedge \cdots \wedge x_1^{n-1} \wedge x_1^n) .
\end{array}$$

Whereas the original expression has size proportional to n the DNF has size proportional to $n2^n$.

The next section introduces a normal form that has more desirable properties than DNFs and CNFs. In particular, there are efficient algorithms for determining the satisfiability and tautology questions.

Exercise 2.1 Describe a polynomial time algorithm for determining whether a DNF is satisfiable.

Exercise 2.2 Describe a polynomial time algorithm for determining whether a CNF is a tautology.

Exercise 2.3 Give a proof of proposition 1.

Exercise 2.4 Explain how Cook's theorem implies that checking in-equivalence between Boolean expressions is NP-hard.

Exercise 2.5 Explain how the question of tautology and satisfiability can be decided if we are given an algorithm for checking equivalence between Boolean expressions.

3 Binary Decision Diagrams

Let $x \to y_0, y_1$ be the *if-then-else* operator defined by

$$x \to y_0, y_1 \;=\; (x \wedge y_0) \vee (\neg x \wedge y_1)$$

hence, $t \to t_0, t_1$ is true if t and t_0 are true or if t is false and t_1 is true. We call t the *test expression*. All operators can easily be expressed using only the if-then-else operator and the constants 0 and 1. Moreover, this can be done in such a way that all tests are performed only on (un-negated) variables and variables occur in no other places. Hence the operator gives rise to a new kind of normal form. For example, $\neg x$ is $(x \to 0, 1)$, $x \Leftrightarrow y$ is $x \to (y \to 1, 0), (y \to 0, 1)$. Since variables must only occur in tests the Boolean expression x is represented as $x \to 1, 0$.

An *If-then-else Normal Form (INF)* is a Boolean expression built entirely from the if-then-else operator and the constants 0 and 1 such that all tests are performed only on variables.

If we by $t[0/x]$ denote the Boolean expression obtained by replacing x with 0 in t then it is not hard to see that the following equivalence holds:

$$t \;=\; x \rightarrow t[1/x], t[0/x]\,. \tag{2}$$

This is known as the *Shannon expansion* of t with respect to x. This simple equation has a lot of useful applications. The first is to generate an INF from any expression t. If t contains no variables it is either equivalent to 0 or 1 which is an INF. Otherwise we form the Shannon expansion of t with respect to one of the variables x in t. Thus since $t[0/x]$ and $t[1/x]$ both contain one less variable than t, we can recursively find INFs for both of these; call them t_0 and t_1. An INF for t is now simply

$$x \rightarrow t_1, t_0.$$

We have proved:

Proposition 2 *Any Boolean expression is equivalent to an expression in INF.*

Example 1 Consider the Boolean expression $t = (x_1 \Leftrightarrow y_1) \wedge (x_2 \Leftrightarrow y_2)$. If we find an INF of t by selecting in order the variables x_1, y_1, x_2, y_2 on which to perform Shannon expansions, we get the expressions

$$\begin{aligned}
t &= x_1 \rightarrow t_1, t_0 \\
t_0 &= y_1 \rightarrow 0, t_{00} \\
t_1 &= y_1 \rightarrow t_{11}, 0 \\
t_{00} &= x_2 \rightarrow t_{001}, t_{000} \\
t_{11} &= x_2 \rightarrow t_{111}, t_{110} \\
t_{000} &= y_2 \rightarrow 0, 1 \\
t_{001} &= y_2 \rightarrow 1, 0 \\
t_{110} &= y_2 \rightarrow 0, 1 \\
t_{111} &= y_2 \rightarrow 1, 0
\end{aligned}$$

Figure 2 shows the expression as a tree. Such a tree is also called a *decision tree.* □

A lot of the expressions are easily seen to be identical, so it is tempting to identify them. For example, instead of t_{110} we can use t_{000} and instead of t_{111} we can use t_{001}. If we substitute t_{000} for t_{110} in the right-hand side of t_{11} and also t_{001} for t_{111}, we in fact see that t_{00} and t_{11} are identical, and in t_1 we can replace t_{11} with t_{00}.

If we in fact identify *all* equal subexpressions we end up with what is known as a *binary decision diagram* (a *BDD*). It is no longer a tree of Boolean expressions but a directed acyclic graph (DAG).

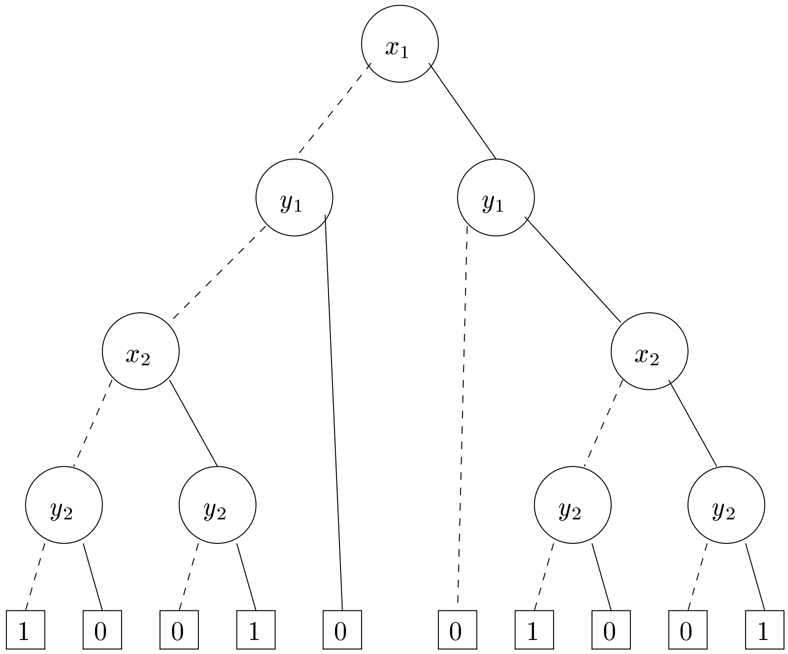


Figure 2: A decision tree for $(x_1 \Leftrightarrow y_1) \wedge (x_2 \Leftrightarrow y_2)$. Dashed lines denote low-branches, solid lines high-branches.

Applying this idea of sharing, t can now be written as:

$$\begin{aligned} t &= x_1 \to t_1, t_0 \\ t_0 &= y_1 \to 0, t_{00} \\ t_1 &= y_1 \to t_{00}, 0 \\ t_{00} &= x_2 \to t_{001}, t_{000} \\ t_{000} &= y_2 \to 0, 1 \\ t_{001} &= y_2 \to 1, 0 \end{aligned}$$

Each subexpression can be viewed as the node of a graph. Such a node is either *terminal* in the case of the constants 0 and 1, or *non-terminal*. A non-terminal node has a low-edge corresponding to the else-part and a high-edge corresponding to the then-part. See figure 3. Notice, that the number of nodes has decreased from 9 in the decision tree to 6 in the BDD. It is not hard to imagine that if each of the terminal nodes were other big decision trees the savings would be dramatic. Since we have chosen to consistently select variables in the same order in the recursive calls during the construction of the INF of t, the variables occur in the same orderings on all paths from the root of the BDD. In this situation the binary decision diagram is said to be *ordered* (an *OBDD*). Figure 3 shows a BDD that is also an OBDD.

Figure 4 shows four OBDDs. Some of the tests (e.g., on x_2 in b) are redundant, since both the low- and high-branch lead to the same node. Such unnecessary tests can be removed: any reference to the redundant node is simply replaced by a reference to

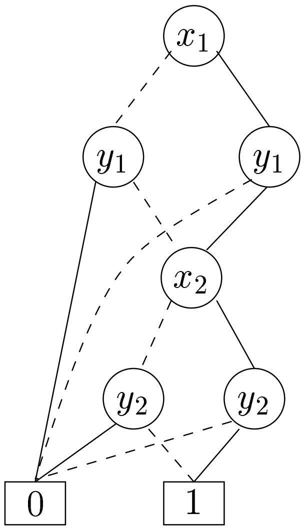


Figure 3: A BDD for $(x_1 \Leftrightarrow y_1) \wedge (x_2 \Leftrightarrow y_2)$ with ordering $x_1 < y_1 < x_2 < y_2$. Low-edges are drawn as dotted lines and high-edges as solid lines.

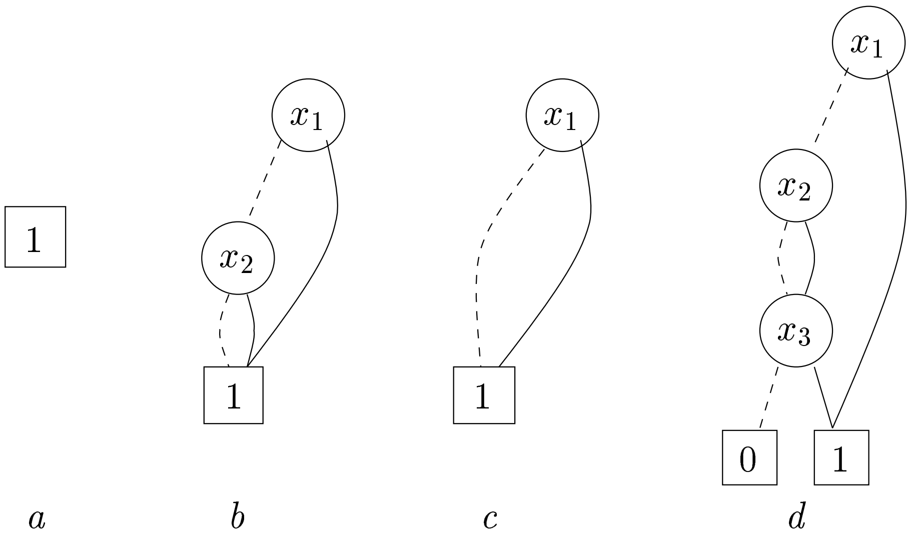


Figure 4: Four OBDDs: a) An OBDD for 1. b) Another OBDD for 1 with two redundant tests. c) Same as *b* with one of the redundant tests removed. d) An OBDD for $x_1 \vee x_3$ with one redundant test.

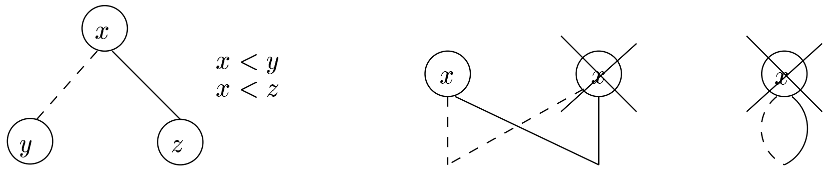


Figure 5: The ordering and reducedness conditions of ROBDDs. Left: Variables must be *ordered*. Middle: Nodes must be *unique*. Right: Only *non-redundant tests* should be present.

its subnode. If all identical nodes are shared and all redundant tests are eliminated, the OBDD is said to be *reduced* (an *ROBDD*). ROBDDs have some very convenient properties centered around the *canonicity lemma* below. (Often when people speak about BDDs they really mean *RO*BDDs.) To summarize:

> A *Binary Decision Diagram (BDD)* is a rooted, directed acyclic graph with
>
> - one or two terminal nodes of out-degree zero labeled 0 or 1, and
> - a set of variable nodes u of out-degree two. The two outgoing edges are given by two functions $low(u)$ and $high(u)$. (In pictures, these are shown as dotted and solid lines, respectively.) A variable $var(u)$ is associated with each variable node.
>
> A BDD is *Ordered* (OBDD) if on all paths through the graph the variables respect a given linear order $x_1 < x_2 < \cdots < x_n$. An (O)BDD is *Reduced* (R(O)BDD) if
>
> - (**uniqueness**) no two distinct nodes u and v have the same variable name and low- and high-successor, i.e.,
>
> $$var(u) = var(v), low(u) = low(v), high(u) = high(v) \text{ implies } u = v,$$
>
> and
> - (**non-redundant tests**) no variable node u has identical low- and high-successor, i.e.,
>
> $$low(u) \neq high(u)\,.$$

The ordering and reducedness conditions are shown in figure 5.

ROBDDs have some interesting properties. They provide compact representations of Boolean expressions, and there are efficient algorithms for performing all kinds of logical operations on ROBDDs. They are all based on the crucial fact that for any function $f : \mathbb{B}^n \to \mathbb{B}$ there is *exactly one ROBDD representing it.* This means, in particular, that there is *exactly one* ROBDD for the constant true (and constant false) function on $\mathbb{B}^n$: the terminal node 1 (and 0 in case of false). Hence, it is possible to *test in constant time whether an ROBDD is constantly true or false.* (Recall that for Boolean expressions this problem is NP-complete.)

To make this claim more precise we must say what we mean for an ROBDD to represent a function. First, it is quite easy to see how the nodes u of an ROBDD inductively defines Boolean expressions t^u: A terminal node is a Boolean constant. A non-terminal node marked with x is an if-then-else expression where the condition is x and the two branches are the Boolean expressions given by the low- or high-son, respectively:

$$\begin{aligned} t^0 &= 0 \\ t^1 &= 1 \\ t^u &= var(u) \to t^{high(u)}, t^{low(u)}, \quad \text{if } u \text{ is a variable node.} \end{aligned}$$

Moreover, if $x_1 < x_2 < \cdots < x_n$ is the variable ordering of the ROBDD, we associate with each node u the function f^u that maps $(b_1, b_2, \ldots, b_n) \in \mathbb{B}^n$ to the truth value of $t^u[b_1/x_1, b_2/x_2, \ldots, b_n/x_n]$. We can now state the key lemma:

Lemma 1 (Canonicity lemma)
For any function $f : \mathbb{B}^n \to \mathbb{B}$ there is exactly one ROBDD u with variable ordering $x_1 < x_2 < \cdots < x_n$ such that $f^u = f(x_1, \ldots, x_n)$.

Proof: The proof is by induction on the number of arguments of f. For $n = 0$ there are only two Boolean functions, the constantly false and constantly true functions. Any ROBDD containing at least one non-terminal node is non-constant. (Why?) Therefore there is exactly one ROBDD for each of these: the terminals 0 and 1.

Assume now that we have proven the lemma for all functions of n arguments. We proceed to show it for all functions of $n+1$ arguments. Let $f : \mathbb{B}^{n+1} \to \mathbb{B}$ be any Boolean function of $n+1$ arguments. Define the two functions f_0 and f_1 of n arguments by fixing the first argument of f to 0 respectively 1:

$$f_b(x_2, \ldots, x_{n+1}) = f(b, x_2, \ldots, x_{n+1}) \text{ for } b \in \mathbb{B}.$$

(Sometimes f_0 and f_1 are called the *negative* and *positive co-factors* of f with respect to x_1.) These functions satisfy the following equation:

$$f(x_1, \ldots, x_n) \;=\; x_1 \to f_1(x_2, \ldots, x_n), f_0(x_2, \ldots, x_n)\,. \tag{3}$$

Since f_0 and f_1 take only n arguments we assume by induction that there are unique ROBDD nodes u_0 and u_1 with $f^{u_0} = f_0$ and $f^{u_1} = f_1$.

There are two cases to consider. If $u_0 = u_1$ then $f^{u_0} = f^{u_1}$ and $f_0 = f^{u_0} = f^{u_1} = f_1 = f$. Hence $u_0 = u_1$ is an ROBDD for f. It is also the only ROBDD for f since due to the ordering, if x_1 is at all present in the ROBDD rooted at u, x_1 would need to be the root node. However, if $f = f^u$ then $f_0 = f^u[0/x_1] = f^{low(u)}$ and $f_1 = f^u[1/x_1] = f^{high(u)}$. Since $f_0 = f^{u_0} = f^{u_1} = f_1$ by assumption, the low- and high-son of u would be the same, making the ROBDD violate the reducedness condition of non-redundant tests.

If $u_0 \neq u_1$ then $f^{u_0} \neq f^{u_1}$ by the induction hypothesis (using the names $x_2, \ldots, x_{n+1}$ in place of $x_1, \ldots, x_n$). We take u to be the node with $var(u) = x_1$, $low(u) = u_0$, and $high(u) = u_1$, i.e., $f^u = x_1 \to f^{u_1}, f^{u_0}$ which is reduced. By assumption $f^{u_1} = f_1$ and $f^{u_0} = f_0$ therefore using (3) we get $f^u = f$. Suppose that v is some other node with $f^v = f$. Clearly, f^v must depend on x_1, i.e., $f^v[0/x_1] \neq f^v[1/x_1]$ (otherwise also

$f_0 = f^v[0/x_1] = f^v[1/x_1] = f_1$, a contradiction). Due to the ordering this means that $var(v) = x_1 = var(u)$. Moreover, from $f^v = f$ it follows that $f^{low(v)} = f_0 = f^{u_0}$ and $f^{high(v)} = f_1 = f^{u_1}$, which by the induction hypothesis implies that $low(v) = u_0 = low(u)$ and $high(v) = u_1 = high(u)$. From the reducedness property of uniqueness it follows that $u = v$. $\square$

An immediate consequence is the following. Since the terminal 1 is an ROBDD for all variable orderings it is the only ROBDD that is constantly true. So in order to check whether an ROBDD is constantly true it suffices to check whether it is the terminal 1 which is definitely a constant time operation. Similarly, ROBDDs that are constantly false must be identical to the terminal 0. In fact, to determine whether two Boolean functions are the same, it suffices to construct their ROBDDs (in the same graph) and check whether the resulting nodes are the same!

The ordering of variables chosen when constructing an ROBDD has a great impact on the size of the ROBDD. If we consider again the expression $(x_1 \Leftrightarrow y_1) \wedge (x_2 \Leftrightarrow y_2)$ and construct an ROBDD using the ordering $x_1 < x_2 < y_1 < y_2$ the ROBDD consists of 9 nodes (figure 6) and not 6 nodes as for the ordering $x_1 < y_1 < x_2 < y_2$ (figure 3).

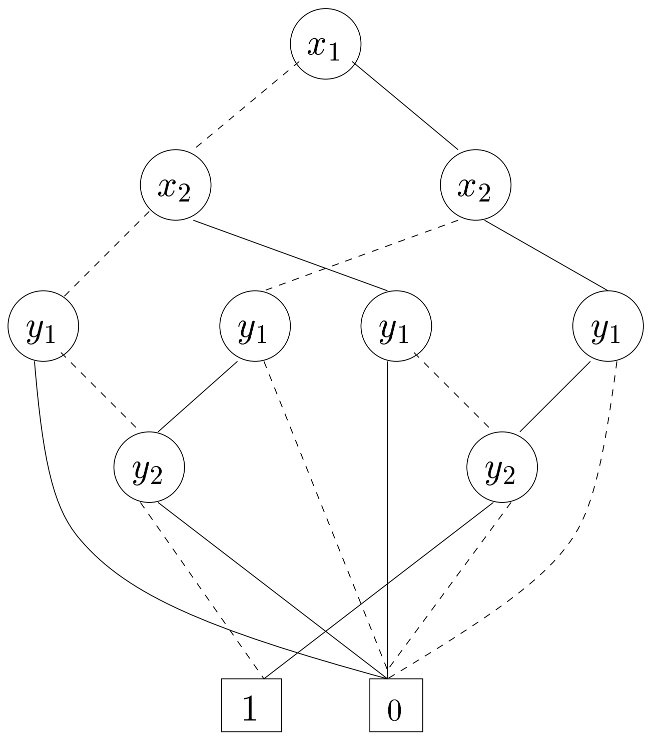


Figure 6: The ROBDD for $(x_1 \Leftrightarrow y_1) \wedge (x_2 \Leftrightarrow y_2)$ with variable ordering $x_1 < x_2 < y_1 < y_2$.

Exercise 3.1 Show how to express all operators from the if-then-else operator and the constants 0 and 1.

Exercise 3.2 Draw the ROBDDs for $(x_1 \Leftrightarrow y_1) \wedge (x_2 \Leftrightarrow y_2) \wedge (x_3 \Leftrightarrow y_3)$ with orderings $x_1 < x_2 < x_3 < y_1 < y_2 < y_3$ and $x_1 < y_1 < x_2 < y_2 < x_3 < y_3$.

Exercise 3.3 Draw the ROBDDs for $(x_1 \Leftrightarrow y_1) \vee (x_2 \Leftrightarrow y_2)$ with orderings $x_1 < x_2 < y_1 < y_2$ and $x_1 < y_1 < x_2 < y_2$. How does it compare with the example in figures 3 and 6? Based on the examples you have seen so far, what variable ordering would you recommend for constructing a small ROBDD for $(x_1 \Leftrightarrow y_1) \wedge (x_2 \Leftrightarrow y_2) \wedge (x_3 \Leftrightarrow y_3) \wedge \cdots \wedge (x_k \Leftrightarrow y_k)$?

Exercise 3.4 Give an example of a sequence of ROBDDs $u_n, 0 \leq n$ which induces exponentially bigger decision trees. I.e., if u_n has size $\Theta(n)$ then the decision tree should have size $\Theta(2^n)$.

Exercise 3.5 Construct an ROBDD of maximum size over six variables.

4 Constructing and Manipulating ROBDDs

In the previous section we saw how to construct an OBDD from a Boolean expression by a simple recursive procedure. The question arises now how do we construct a *reduced* OBDD? One way is to first construct an OBDD and then proceed by reducing it. Another more appealing approach, which we follow here, is to reduce the OBDD during construction.

To describe how this is done we will need an explicit representation of ROBDDs. Nodes will be represented as numbers $0, 1, 2, \ldots$ with 0 and 1 reserved for the terminal nodes. The variables in the ordering $x_1 < x_2 < \cdots < x_n$ are represented by their indices $1, 2, \ldots, n$. The ROBDD is stored in a table $T : u \mapsto (i, l, h)$ which maps a node u to its three attributes $var(u) = i$, $low(u) = l$, and $high(u) = h$. Figure 7 shows the representation of the ROBDD from figure 3 (with the variable names changed to $x_1 < x_2 < x_3 < x_4$).

4.1 Mk

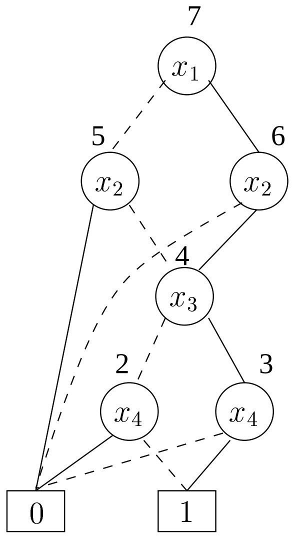
In order to ensure that the OBDD being constructed is reduced, it is necessary to determine from a triple (i, l, h) whether there exists a node u with $var(u) = i$, $low(u) = l$, and $high(u) = h$. For this purpose we assume the presence of a table $H : (i, l, h) \mapsto u$ mapping triples (i, l, h) of variable indices i, and nodes l, h to nodes u. The table H is the "inverse" of the table T, i.e., for variable nodes u, $T(u) = (i, l, h)$, if and only if, $H(i, l, h) = u$. The operations needed on the two tables are:

$T : u \mapsto (i, l, h)$

$init(T)$	initialize T to contain only 0 and 1
$u \leftarrow add(T, i, l, h)$	allocate a new node u with attributes (i, l, h)
$var(u), low(u), high(u)$	lookup the attributes of u in T

$H : (i, l, h) \mapsto u$

$init(H)$	initialize H to be empty
$b \leftarrow member(H, i, l, h)$	check if (i, l, h) is in H
$u \leftarrow lookup(H, i, l, h)$	find $H(i, l, h)$
$insert(H, i, l, h, u)$	make (i, l, h) map to u in H



$T : u \mapsto (i, l, h)$

u	*var*	*low*	*high*
0	5		
1	5		
2	4	1	0
3	4	0	1
4	3	2	3
5	2	4	0
6	2	0	4
7	1	5	6

Figure 7: Representing an ROBDD with ordering $x_1 < x_2 < x_3 < x_4$. The numbers inside the vertices are the identities used in the representation. The numbers 0 and 1 are reserved for the terminal nodes. The numbers to the right of the ROBDD shows the index of the variables in the ordering. The constants are assigned an index which is the number of variables in the ordering plus one (here $4+1 = 5$). This makes some subsequent algorithms easier to present. The low- and high-fields are unused for the terminal nodes.

Mk$[T, H](i, l, h)$
if $l = h$ **then return** l
else if *member*(H, i, l, h) **then**
  return *lookup*(H, i, l, h)
else $u \leftarrow$ *add*(T, i, l, h)
  insert(H, i, l, h, u)
  return u

Figure 8: The function mk$[T, H](i, l, h)$.

BUILD$[T, H](t)$

1: **function** BUILD'(t, i) =
2: **if** $i > n$ **then**
3: **if** t is false **then return** 0 **else return** 1
4: **else** $v_0 \leftarrow$ BUILD'$(t[0/x_i], i+1)$
5: $v_1 \leftarrow$ BUILD'$(t[1/x_i], i+1)$
6: **return** MK(i, v_0, v_1)
7: **end** BUILD'
8:
9: **return** BUILD'$(t, 1)$

Figure 9: Algorithm for building an ROBDD from a Boolean expression t using the ordering $x_1 < x_2 < \cdots < x_n$. In a call BUILD'(t, i), i is the lowest index that any variable of t can have. Thus when the test $i > n$ succeeds, t contains no variables and must be either constantly false or true.

We shall assume that all these operations can be performed in *constant time*, $O(1)$. Section 5 will show how such a low complexity can be achieved.

The function MK$[T, H](i, l, h)$ (see figure 8) searches the table H for a node with variable index i and low-, high-branches l, h and returns a matching node if one exists. Otherwise it creates a new node u, inserts it into H and returns the identity of it. The running time of MK is $O(1)$ due to the assumptions on the basic operations on T and H. The OBDD is ensured to be reduced if nodes are only created through the use of MK. In describing MK and subsequent algorithms, we make use of the notation $[T, H]$ to indicate that MK depends on the global data structures T and H, but we leave out the arguments when invoking it as part of other algorithms.

4.2 Build

The construction of an ROBDD from a given Boolean expression t proceeds as in the construction of an if-then-else normal form (INF) in section 2. An ordering of the variables $x_1 < \cdots < x_n$ is fixed. Using the Shannon expansion $t = x_1 \rightarrow t[1/x_1], t[0/x_1]$, a node for t is constructed by a call to MK, after the nodes for $t[0/x_1]$ and $t[1/x_1]$ have been constructed by recursion. The algorithm is shown in figure 9. The call BUILD'(t, i) constructs an ROBDD for a Boolean expression t with variables in $\{x_i, x_{i+1}, \ldots, x_n\}$. It does so by first recursively constructing ROBDDs v_0 and v_1 for $t[0/x_i]$ and $t[1/x_i]$ in lines 4 and 5, and then proceeding to find the identity of the node for t in line 6. Notice that if v_0 and v_1 are identical, or if there already is a node with the same i, v_0 and v_1, no new node is created.

An example of using BUILD to compute an ROBDD is shown in figure 10. The running time of BUILD is bad. It is easy to see that for a variable ordering with n variables there will always be generated on the order of 2^n calls .

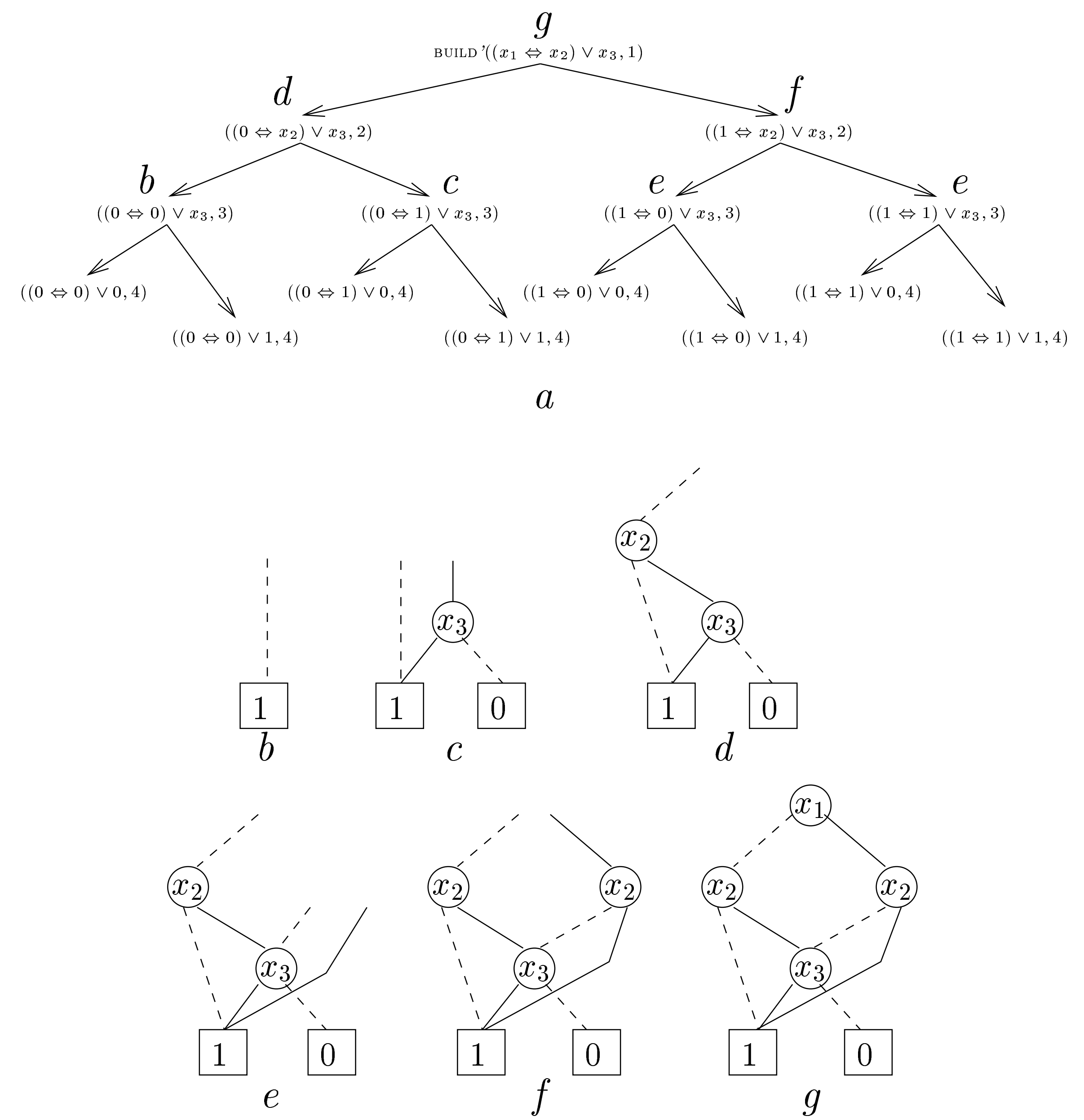


Figure 10: Using BUILD on the expression $(x_1 \Leftrightarrow x_2) \vee x_3$. (a) The tree of calls to BUILD. (b) The ROBDD after the call BUILD'$((0 \Leftrightarrow 0) \vee x_3, 3)$. (c) After the call BUILD'$((0 \Leftrightarrow 1) \vee x_3, 3)$. (d) After the call BUILD'$((0 \Leftrightarrow x_2) \vee x_3, 2)$. (e) After the calls BUILD'$((1 \Leftrightarrow 0) \vee x_3, 3)$ and BUILD'$((1 \Leftrightarrow 1) \vee x_3, 3)$. (f) After the call BUILD'$((1 \Leftrightarrow x_2) \vee x_3, 2)$. (g) The final result.

4.3 Apply

APPLY$[T, H](op, u_1, u_2)$
init(G)

function APP$(u_1, u_2) =$
  if $G(u_1, u_2) \neq$ *empty* **then return** $G(u_1, u_2)$
  else if $u_1 \in \{0, 1\}$ **and** $u_2 \in \{0, 1\}$ **then** $u \leftarrow op(u_1, u_2)$
  else if $var(u_1) = var(u_2)$ **then**
    $u \leftarrow$ MK$(var(u_1),$ APP$(low(u_1), low(u_2)),$ APP$(high(u_1), high(u_2)))$
  else if $var(u_1) < var(u_2)$ **then**
    $u \leftarrow$ MK$(var(u_1),$ APP$(low(u_1), u_2),$ APP$(high(u_1), u_2))$
  else $(*\ var(u_1) > var(u_2)\ *)$
    $u \leftarrow$ MK$(var(u_2),$ APP$(u_1, low(u_2)),$ APP$(u_1, high(u_2)))$
  $G(u_1, u_2) \leftarrow u$
  return u
end APP

return APP(u_1, u_2)

Figure 11: The algorithm APPLY$[T, H](op, u_1, u_2)$.

All the binary Boolean operators on ROBDDs are implemented by the same general algorithm APPLY(op, u_1, u_2) that for two ROBDDs computes the ROBDD for the Boolean expression t^{u_1} *op* t^{u_2}. The construction of APPLY is based on the Shannon expansion (2):

$$t \;=\; x \to t[1/x], t[0/x]\,.$$

Observe that for all Boolean operators *op* the following holds:

$$(x \to t_1, t_2)\ op\ (x \to t_1', t_2') \;=\; x \to t_1\ op\ t_1',\ t_2\ op\ t_2' \tag{4}$$

If we start from the root of the two ROBDDs we can construct the ROBDD of the result by recursively constructing the low- and the high-branches and then form the new root from these. Again, to ensure that the result is reduced, we create the node through a call to MK. Moreover, to avoid an exponential blow-up of recursive calls, *dynamic programming* is used. The algorithm is shown in figure 11.

Dynamic programming is implemented using a table of results G. Each entry (i, j) is either *empty* or contains the earlier computed result of APP(i, j). The algorithm distinguishes between four different cases, the first of them handles the situation where both arguments are terminal nodes, the remaining three handle the situations where at least one argument is a variable node.

If both u_1 and u_2 are terminal, a new *terminal node* is computed having the value of *op* applied to the two truth values. (Recall, that terminal node 0 is represented by a node with identity 0 and similarly for 1.)

If at least one of u_1 and u_2 are non-terminal, we proceed according to the variable index. If the nodes have the same index, the two low-branches are paired and APP recursively computed on them. Similarly for the high-branches. This corresponds exactly to the case shown in equation (4). If they have different indices, we proceed by pairing the node with lowest index with the low- and high-branches of the other. This corresponds to the equation

$$(x_i \to t_1, t_2)\ op\ t \quad = \quad x_i \to t_1\ op\ t, t_2\ op\ t \tag{5}$$

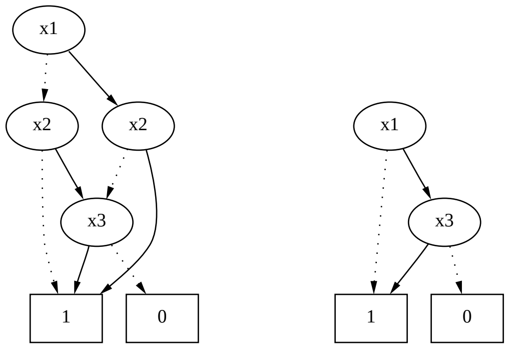
which holds for all t. Since we have taken the index of the terminals to be one larger than the index of the non-terminals, the last two cases, $var(u_1) < var(u_2)$ and $var(u_1) > var(u_2)$, take account of the situations where one of the nodes is a terminal.

Figure 12 shows an example of applying the algorithm on two small ROBDDs. Notice how pairs of nodes from the two ROBDDs are combined and computed.

To analyze the complexity of APPLY we let $|u|$ denote the number of nodes that can be reached from u in the ROBDD. Assume that G can be implemented with constant lookup and insertion times. (See section 5 for details on how to achieve this.) Due to the dynamic programming at most $|u_1|\,|u_2|$ calls to APPLY are generated. Each call takes constant time. The total running time is therefore $O(|u_1|\,|u_2|)$.

4.4 Restrict

The next operation we consider is the *restriction* of a ROBDD u. That is, given a truth assignment, for example $[0/x_3, 1/x_5, 1/x_6]$, we want to compute the ROBDD for t^u under this restriction, i.e., find the ROBDD for $t^u[0/x_3, 1/x_5, 1/x_6]$. As an example consider the ROBDD of figure 10(g) (repeated below to the left) representing the Boolean expression $(x_1 \Leftrightarrow x_2) \vee x_3$. Restricting it with respect to the truth assignment $[0/x_2]$ yields an ROBDD for $(\neg x_1 \vee x_3)$. It is constructed by replacing each occurrence of a node with label x_2 by its left branch yielding the ROBDD at the right:



The algorithm again uses MK to ensure that the resulting OBDD is reduced. Figure 13 shows the algorithm in the case where only singleton truth assignments ($[b/x_j]$, $b \in \{0, 1\}$) are allowed. Intuitively, in computing RESTRICT(u, j, b) we search for all nodes with $var = j$ and replace them by their low- or high-son depending on b. Since this might force nodes above the point of replacemen to become equal, it is followed by a reduction (through the calls to MK). Due to the two recursive calls in line 3, the algorithm has an exponential running time, see exercise 4.7 for an improvement that reduces this to linear time.

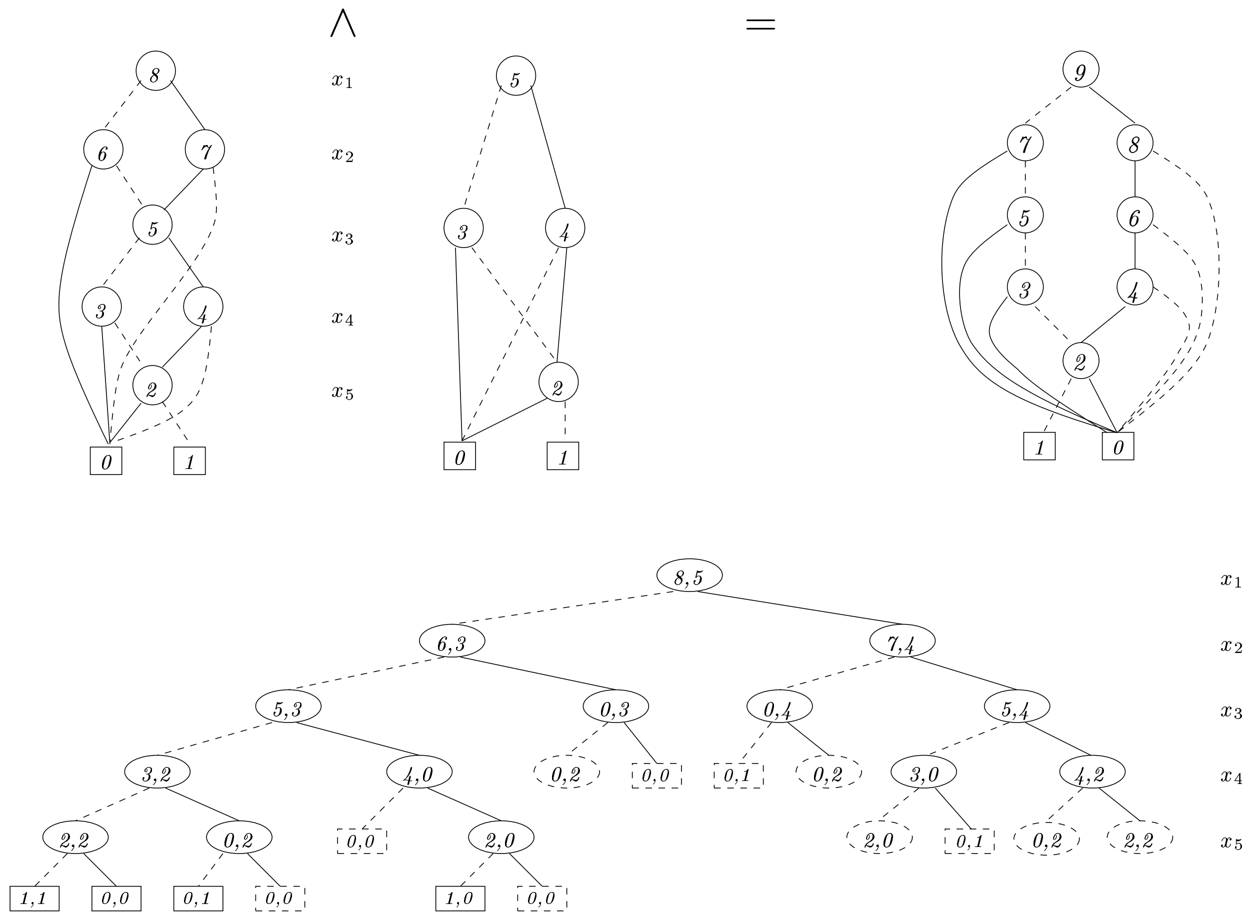


Figure 12: An example of applying the algorithm APPLY for computing the conjunction of the two ROBDDs shown at the top left. The result is shown to the right. Below the tree of arguments to the recursive calls of APP. Dashed nodes indicate that the value of the node has previously been computed and is not recomputed due to the use of dynamic programming. The solid ellipses show calls that finishes by a call to MK with the variable index indicated by the variables to the right of the tree.

```
RESTRICT[T, H](u, j, b) =
function res(u) =
    if var(u) > j then return u
    else if var(u) < j then return MK(var(u), res(low(u)), res(high(u)))
    else (* var(u) = j *) if b = 0 then return res(low(u))
    else (* var(u) = j, b = 1 *) return res(high(u))
end res
return res(u)
```

Figure 13: The algorithm RESTRICT$[T, H](u, j, b)$ which computes an ROBDD for $t^u[j/b]$.

4.5 SatCount, AnySat, AllSat

In this section we consider operations to examine the set of satisfying truth assignments of a node u. A truth assignment ρ satisfies a node u if $t^u[\rho]$ can be evaluated to 1 using the truth tables of the Boolean operators. Formally, the satisfying truth assignments is the set $sat(u)$:

$$sat(u) = \{\rho \in \mathbb{B}^{\{x_1,\ldots,x_n\}} \mid t^u[\rho] \text{ is true }\},$$

where $\mathbb{B}^{\{x_1,\ldots,x_n\}}$ denotes the set of all truth assignments for variables $\{x_1, \ldots, x_n\}$, i.e., functions from $\{x_1, \ldots, x_n\}$ to the truth values $\mathbb{B} = \{0, 1\}$. The first algorithm, SATCOUNT, computes the size of $sat(u)$, see figure 14. The algorithm exploits the following fact. If u is a node with variable index $var(u)$ then two sets of truth assignments can make f^u true. The first set has $varu$ equal to 0, the other has $varu$ equal to 1. For the first set, the number is found by finding the number of truth assignments $count(low(u))$ making $low(u)$ true. All variables between $var(u)$ and $var(low(u))$ in the ordering can be chosen arbitrarily, therefore in the case of $varu$ being 0, a total of $2^{var(low(u))-var(u)-1} * count(low(u))$ satisfying truth assignments exists. To be efficient, dynamic programming should be applied in SATCOUNT (see exercise 4.10).

The next algorithm ANYSAT in figure 15 finds a satisfying truth assignment. Some irrelevant variables present in the ordering might not appear in the result and they can be assigned any value whatsoever. ANYSAT simply finds a path leading to 1 by a depth-first traversal, prefering somewhat arbitrarily low-edges over high-edges. It is particularly simple due to the observation that *if a node is not the terminal 0, it has at least one path leading to 1.* The running time is clearly linear in the result.

ALLSAT in figure 16 finds all satisfying truth-assignments leaving out irrelevant variables from the ordering. ALLSAT(u) finds all paths from a node u to the terminal 1. The running time is linear in the size of the result multiplied with the time to add the single assignments $[x_{var(u)} \mapsto 0]$ and $[x_{var(u)} \mapsto 1]$ in front of a list of up to n elements. However, the result can be exponentially large in $|u|$, so the running time is the poor $O(2^{|u|}n)$.

```
SatCount[T](u)
function count(u)
    if u = 0 then res ← 0
    else if u = 1 then res ← 1
    else res ← 2^(var(low(u))−var(u)−1) * count(low(u))
             + 2^(var(high(u))−var(u)−1) * count(high(u))
    return res
end count

return 2^(var(u)−1) * count(u)
```

Figure 14: An algorithm for determining the number of valid truth assignments. Recall, that the "variable index" *var* of 0 and 1 in the ROBDD representation is $n+1$ when the ordering contains n variables (numbered 1 through n). This means that $var(0)$ and $var(1)$ always gives $n+1$.

```
AnySat(u)
    if u = 0 then Error
    else if u = 1 then return []
    else if low(u) = 0 then return [x_var(u) ↦ 1, AnySat(high(u))]
    else return [x_var(u) ↦ 0, AnySat(low(u))]
```

Figure 15: An algorithm for returning a satisfying truth-assignment. The variables are assumed to be $x_1, \ldots, x_n$ ordered in this way.

```
AllSat(u)
    if u = 0 then return ⟨ ⟩
    else if u = 1 then return ⟨ [ ] ⟩
    else return
        ⟨add [x_var(u) ↦ 0] in front of all
          truth-assignments in AllSat(low(u)),
          add [x_var(u) ↦ 1] in front of all
          truth-assignments in AllSat(high(u))⟩
```

Figure 16: An algorithm which returns all satisfying truth-assignments. The variables are assumed to be $x_1, \ldots x_n$ ordered in this way. We use $\langle \cdots \rangle$ to denote sequences of truth assignments. In particular, $\langle\,\rangle$ is the empty sequence of truth assignments, and $\langle\,[\,]\,\rangle$ is the sequence consisting of the single empty truth assignment.

```
SIMPLIFY(d, u)
function sim(d, u)
    if d = 0 then return 0
    else if u ≤ 1 then return u
    else if d = 1 then
        return MK(var(u), sim(d, low(u)), sim(d, high(u)))
    else if var(d) = var(u) then
        if low(d) = 0 then return sim(high(d), high(u))
        else if high(d) = 0 then return sim(low(d), low(u))
        else return MK(var(u),
                       sim(low(d), low(u)),
                       sim(high(d), high(u)))
    else if var(d) < var(u) then
        return MK(var(d), sim(low(d), u), sim(high(d), u))
    else
        return MK(var(u), sim(d, low(u)), sim(d, high(u)))
end sim

return sim(d, u)
```

Figure 17: An algorithm (due to Coudert et al [CBM89]) for simplifying an ROBDD b that we only care about on the domain d. Dynamic programming should be applied to improve efficiency (exercise 4.12)

MK(i, u_0, u_1)	$O(1)$	
BUILD(t)	$O(2^n)$	
APPLY(op, u_1, u_2)	$O(\|u_1\|\,\|u_2\|)$	
RESTRICT(u, j, b)	$O(\|u\|)$	See note
SATCOUNT(u)	$O(\|u\|)$	See note
ANYSAT(u)	$O(\|p\|)$	$p = AnySat(u)$, $\|p\| = O(\|u\|)$
ALLSAT(u)	$O(\|r\| * n)$	$r = AllSat(u)$, $\|r\| = O(2^{\|u\|})$
SIMPLIFY(d, u)	$O(\|d\|\|u\|)$	See note

Note: These running times only holds if dynamic programming is used (exercises 4.7, 4.10, and 4.12).

Table 1: Worst-case running times for the ROBDD operations. The running times are the expected running times since they are all based on a hash-table with expected constant time search and insertion operations.

4.6 Simplify

The final algorithm called SIMPLIFY is shown in figure 17. The algorithm is used to simplify an ROBDD by trying to remove nodes. The simplification is based on a domain d of interest. The ROBDD u is supposed to be of interest only on truth assignments that also satisfy d. (This occurs when using ROBDDs for *formal verification*. Section 7 shows how to do formal verification with ROBDDs, but contains no example of using SIMPLIFY.)

To be precise, given d and u, SIMPLIFY finds another ROBDD u', typically smaller than u, such that $t^d \wedge t^u = t^d \wedge t^{u'}$. It does so by trying to identify sons, and thereby making some nodes redundant. A more detailed analysis is left to the reader.

The running time of the algorithms of the previous sections is summarized in table 1.

4.7 Existential Quantification and Substitution

When applying ROBDDs often *existential quantification* and *composition* is used. Existential quantification is the Boolean operation $\exists x.t$. The meaning of an existential quantification of a Boolean variable is given by the following equation:

$$\exists x.t \;=\; t[0/x] \vee t[1/x]\,. \tag{6}$$

On ROBDDs existential quantification can therefore be implemented using two calls to RESTRICT and a single call to APPLY.

Composition is the ROBDD operation performing the equivalent of substitution on Boolean expression. Often the notation $t[t'/x]$ is used to describe the result of substituting all free occurrences of x in t by t'. (An occurrence of a variable is free if it is not within the scope of a quantifier.)[1] To perform this substitution on ROBDDs we observe the

[1] Since ROBDDs contain no quantifiers we shall not be concerned with the problems of free variables of t' being bound by quantifiers of t.

following equation, which holds if t contains no quantifiers:

$$t[t'/x] \;=\; t[t' \to 1, 0/x] \;=\; t' \to t[1/x], t[0/x]. \tag{7}$$

Since $(t' \to t[1/x], t[0/x]) = (t' \wedge t[1/x]) \vee (\neg t' \wedge t[0/x])$ we can compute this with two applications of RESTRICT and three applications of APPLY (with the operators $\wedge$, $(\neg\_) \wedge \_$, $\vee$). However, by essentially generalizing APPLY to operators *op* with three arguments we can do better (see exercise 4.13).

Exercises

Exercise 4.1 Construct the ROBDD for $\neg x_1 \wedge (x_2 \Leftrightarrow \neg x_3)$ with ordering $x_1 < x_2 < x_3$ using the algorithm BUILD in figure 9.

Exercise 4.2 Show the representation of the ROBDD of figure 6 in the style of figure 7.

Exercise 4.3 Suggest an improvement BUILDCONJ(t) of BUILD which generates only a linear number of calls for Boolean expressions t that are conjunctions of variables and negations of variables.

Exercise 4.4 Construct the ROBDDs for x and $x \Rightarrow y$ using whatever ordering you want. Compute the disjunction of the two ROBDDs using APPLY.

Exercise 4.5 Construct the ROBDDs for $\neg(x_1 \wedge x_3)$ and $x_2 \wedge x_3$ using BUILD with the ordering $x_1 < x_2 < x_3$. Use APPLY to find the ROBDD for $\neg(x_1 \wedge x_3) \vee (x_2 \wedge x_3)$.

Exercise 4.6 Is there any essential difference in running time between finding RESTRICT$(b, 1, 0)$ and RESTRICT$(b, n, 0)$ when the variable ordering is $x_1 < x_2 < \cdots < x_n$?

Exercise 4.7 Use dynamic programming to improve the running time of RESTRICT.

Exercise 4.8 Generalise RESTRICT to arbitrary truth assignments $[x_{i_1} = b_{i_1}, x_{i_2} = b_{i_2}, \ldots, x_{i_n} = b_{i_n}]$. It might be convenient to assume that $x_{i_1} < x_{i_2} < \cdots < x_{i_n}$.

Exercise 4.9 Suggest a substantially better way of building ROBDDs for (large) Boolean expressions than BUILD.

Exercise 4.10 Change SATCOUNT such that dynamic programming is used. How does this change the running time?

Exercise 4.11 Explain why dynamic programming does not help in improving the running time of ALLSAT.

Exercise 4.12 Improve the efficiency of SIMPLIFY with dynamic programming.

Exercise 4.13 Write the algorithm COMPOSE(u_1, x, u_2) for computing the ROBDD of $u_1[u_2/x]$ efficiently along the lines of APPLY. First generalize APPLY to operators *op* with three arguments (as for example the if-then-else operator), utilizing once again the Shannon expansion. Then use equation 7 to write the algorithm.

5 Implementing the ROBDD operations

There are many choices that have to be taken in implementing the ROBDD operations. There is no obvious best way of doing it. This section gives hints for some reasonable solutions.

First, the node table T is an array as shown in figure 7. The only problem is that the size of the array is not known until the full BDD has been constructed. Either a fixed upper bound could be assumed, or other tricks must be applied (for example *dynamic arrays* [CLR90, sec. 18.4]). The table H could be implemented as a hash-table using for instance the hash function

$$h(i, v0, v1) = pair(i, pair(v0, v1)) \bmod m$$

where *pair* is a pairing function that maps pairs of natural numbers to natural numbers and m is a prime. One choice for the pairing function is

$$pair(i, j) = \frac{(i + j)(i + j + 1)}{2} + i$$

which is a bijection, and therefore "perfect": it produces no collisions. As usual with hash-tables we have to decide on the size as a prime m. However, since the size of H grows dynamically it can be hard to find a good choice for m. One solution would be to take m very large, for example $m = 15485863$ (which is the 1000000'th prime number), and then take as the hashing function

$$h'(i, v0, v1) = h(i, v0, v1) \bmod 2^k$$

using a table of size 2^k. Starting from some reasonable small value of k we could increase the table when it contains 2^k elements by adding one to k, construct a new table and rehash all elements into this new table. (Again, see for example [CLR90, sec. 18.4] for details.) For such a dynamic hash-table the amortized, expected cost of each operation is still $O(1)$.

The table G used in Apply could be implemented as a two-dimensional array. However, it turns out to be very sparsely used – especially if we succeed in getting small ROBDDs – and it is better to use a hash-table for it. The hashing function used could be $g(v0, v1) = pair(v0, v1) \bmod m$ and as for H a dynamic hash-table could be used.

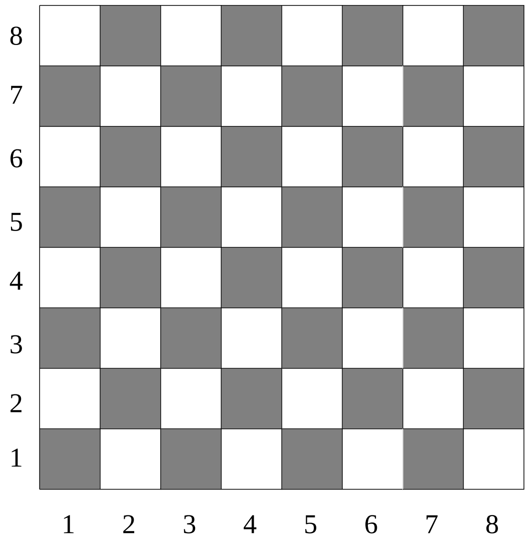
6 Examples of problem solving with ROBDDs

This section will describe various examples of problems that can be solved with an ROBDD-package. The examples are *not* chosen to illustrate when ROBDDs are the best choice, but simply chosen to illustrate the scope of potential applications.

6.1 The 8 Queens problem

A classical chess-board problem is the *8 queens problem*: Is it possible to place 8 queens on a chess board so that no queen can be captured by another queen? To be a bit more general we could ask the question for arbitrary N: Is it possible to place N queens safely on a $N \times N$ chess board?

To solve the problem using ROBDDs we must encode it using Boolean variables. We do this by introducing a variable for each position on the board. We name the variables as $x_{ij}, 1 \leq i, j \leq N$ where i is the row and j is the column. A variable will be 1 if a queen is placed on the corresponding position.



The capturing rules for queens require that no other queen can be positioned on the same row, column, or any of the diagonals. This we can express as Boolean expressions: For all i, j,

$$
\begin{aligned}
x_{ij} &\Rightarrow \bigwedge_{1 \leq l \leq N, l \neq j} \neg x_{il} \\
x_{ij} &\Rightarrow \bigwedge_{1 \leq k \leq N, k \neq i} \neg x_{kj} \\
x_{ij} &\Rightarrow \bigwedge_{1 \leq k \leq N, 1 \leq j+k-i \leq N, k \neq i} \neg x_{k,j+k-i} \\
x_{ij} &\Rightarrow \bigwedge_{1 \leq k \leq N, 1 \leq j+i-k \leq N, k \neq i} \neg x_{k,j+i-k}
\end{aligned}
$$

Moreover, there must be a queen in each row: For all i,

$$x_{i1} \vee x_{i2} \vee \cdots \vee x_{iN}$$

Taking the conjunction of all the above requirements, we get a predicate $Sol_N(\vec{x})$ true at exactly the configurations that are solutions to the N queens problem.

Exercise 6.1 (8 Queens Problem) Write a program that can find an ROBDD for $Sol_N(\vec{x})$ when given N as input. Make a table of the number of solutions to the N queens problem for $N = 1, 2, 3, 4, 5, 6, 7, 8, \ldots$ When there is a solution, give one.

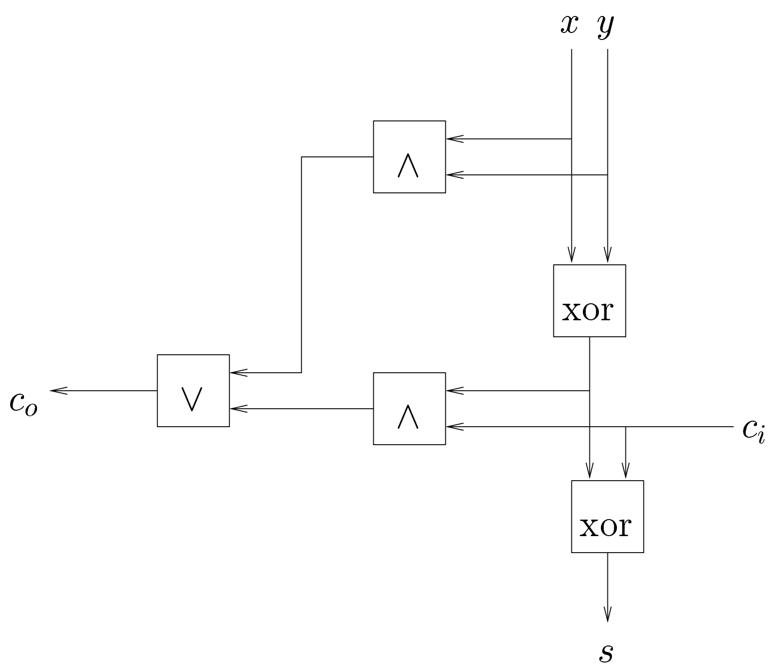


Figure 18: A full-adder

6.2 Correctness of Combinational Circuits

A *full-adder* takes as arguments two bits x and y and an incoming carry bit c_i. It produces as output a sum bit s and an outgoing carry bit c_o. The requirement is that $2 * c_o + s = x + y + c_i$, in other words c_o is the most significant bit of the sum of x, y, and c_i, and s the least significant bit. The requirement can be written down as a table for c_o and a table for s in terms of values of x, y, and c_i. From such a table it is easy to write down a DNF for c_o and s.

At the normal level of abstraction a combinational circuit is nothing else than a Boolean expression. It can be represented as an ROBDD, using Build to construct the trivial ROBDDs for the inputs and using a call to Apply for each gate.

Exercise 6.2 Find DNFs for c_o and s. Verify that the circuit in figure 18 implements a one bit full-adder using the ROBDD-package and the DNFs.

6.3 Equivalence of Combinational Circuits

As above we can construct an ROBDD from a combinational circuit and use the ROBDDs to show properties. For instance, the equivalence with other circuits.

Exercise 6.3 Verify that the two circuits in figure 19 are *not* equivalent using ROBDDs. Find an input that returns different outputs in the two circuits.

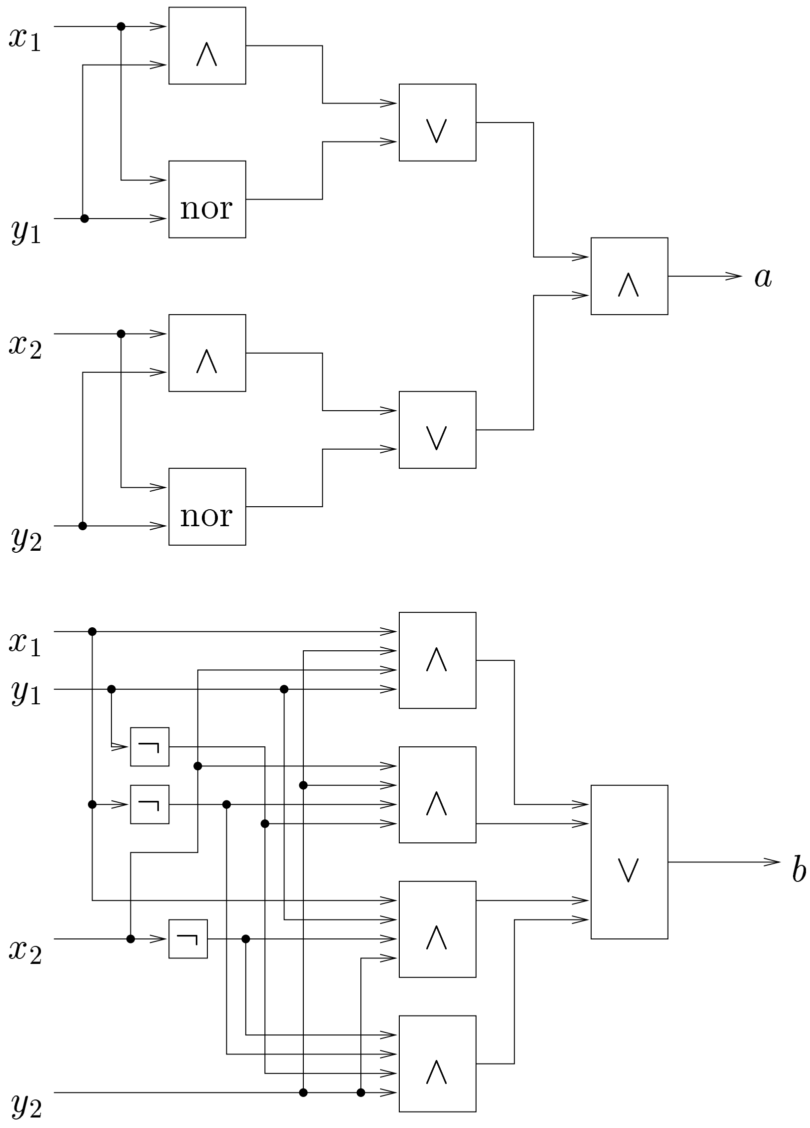


Figure 19: Two circuits used in exercise 6.3

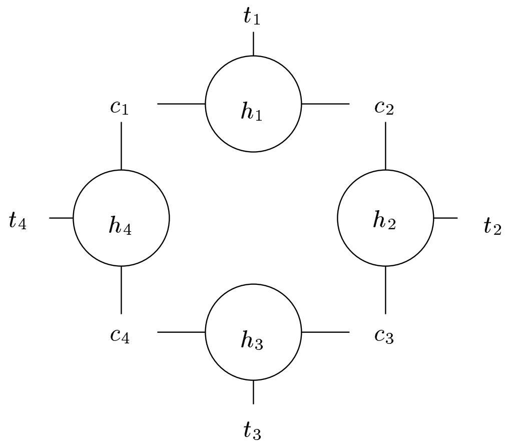


Figure 20: Milner's Scheduler with 4 cyclers. The token is passed clockwise from c_1 to c_2 to c_3 to c_4 and back to c_1

7 Verification with ROBDDs

One of the major uses of ROBDDs is in *formal verification*. In formal verification a model of a system M is given together with some properties P supposed to hold for the system. The task is to determine whether indeed M satisfy P. The approach we take, in which we shall use an algorithm to answer the satisfaction problem, is often called *model checking*.

We shall look at a concrete example called *Milner's Scheduler* (taken from Milner's book [Mil89]). The model consists of N *cyclers*, connected in a ring, that co-operates on starting and detecting termination of N tasks that are not further described. The scheduler must make sure that the N tasks are always started in order but they are allowed to terminate in any order. This is one of the properties that has to be shown to hold for the model. The cyclers try to fulfill this by passing a token: the holder of the token is the only process allowed to start its task.

All cyclers are similar except that one of them has the token in the initial state. The cyclers cyc_i, $1 \leq i \leq N$ are described in a state-based fashion as small transition systems over the Boolean variables t_i, h_i, and c_i. The variable t_i is 1 when task i is running and 0 when it is terminated; h_i is 1 when cycler i has a token, 0 otherwise; c_i is 1 when cycler $i-1$ has put down the token and cycler i not yet picked it up. Hence a cycler starts a task by changing t_i from 0 to 1, and detects its termination when t_i is again changed back to 0; and it picks up the token by changing c_i from 1 to 0 and puts it down by changing c_{i+1} from 0 to 1. The behaviour of cycler i is described by two transitions:

$$\begin{array}{ll} \textbf{if } c_i = 1 \ \wedge \ t_i = 0 & \textbf{then } t_i, c_i, h_i \mathrel{\textbf{:=}} 1, 0, 1 \\ \textbf{if } h_i = 1 & \textbf{then } c_{(i \bmod N)+1}, h_i \mathrel{\textbf{:=}} 1, 0 \end{array}$$

The meaning of a transition "**if** *condition* **then** *assignment*" is that, if the *condition* is true in some state, then the system can evolve to a new state performing the (parallel) *assignment*. Hence, if the system is in a state where c_i is 1 and t_i is 0 then we can simultaneously set t_i to 1, c_i to 0 and h_i to 1.

The transitions are encoded by a single predicate over the value of the variables before the transitions (the *pre-state*) and the values after the transition (the *post-state*). The variables in the pre-state are the $t_i, h_i, c_i, 1 \leq i \leq N$ which we shall collectively refer to as $\vec{x}$ and in the post-state $t'_i, h'_i, c'_i, 1 \leq i \leq N$, which we shall refer to as $\vec{x}'$. Each transition is an atomic action that excludes any other action. Therefore in the encoding we shall often have to say that a lot of variables are unchanged. Assume that S is a subset of the unprimed variables $\vec{x}$. We shall use a predicate $\mathit{unchanged}_S$ over $\vec{x}, \vec{x}'$ which ensures that all variables in S are unchanged. It is defined as follows:

$$\mathit{unchanged}_S \;=_{\text{def}}\; \bigwedge_{x \in S} x = x' \,.$$

It is slightly more convenient to use the predicate $\mathit{assigned}_{S'} = \mathit{unchanged}_{\vec{x} \setminus S'}$ which express that every variable *not* in S' is unchanged. We can now define P_i, the transitions of cycler i over the variables $\vec{x}, \vec{x'}$ as follows:

$$\begin{array}{rcl} P_i \;=_{\text{def}} & & (c_i \wedge \neg t_i \wedge t'_i \wedge \neg c'_i \wedge h'_i \qquad \wedge\, \mathit{assigned}_{\{c_i, t_i, h_i\}}) \\ & \vee & (h_i \wedge \qquad c'_{(i \bmod N)+1} \wedge \neg h'_i \wedge \mathit{assigned}_{\{c_{(i \bmod N)+1}, h_i\}}) \end{array}$$

The signalling of termination of task i, by changing t_i from 1 to 0 performed by the environment is modeled by N transitions $E_i, 1 \leq i \leq N$:

$$E_i \;=_{\text{def}}\; t_i \wedge \neg t'_i \wedge \mathit{assigned}_{\{t_i\}},$$

expressing the transitions **if** $t_i = 1$ **then** $t_i := 0$. Now, at any given state the system can perform one of the transitions from one of the P_i's or the E_i's, i.e., all possible transitions are given by the predicate T:

$$T \;=_{\text{def}}\; P_1 \vee \cdots \vee P_n \vee E_1 \vee \cdots \vee E_n \,.$$

In the initial state we assume that all tasks are stopped, no cycler has a token and only place 1 (c_1) has a token. Hence the initial state can be characterized by the predicate I over the unprimed variables $\vec{x}$ given by:

$$I \;=_{\text{def}}\; \neg\vec{t} \wedge \neg\vec{h} \wedge c_1 \wedge \neg c_2 \wedge \cdots \wedge \neg c_N \,.$$

(Here $\neg$ applied to a vector $\vec{t}$ means the conjunction of $\neg$ applied to each coordinate t_i.) The predicates describing Milner's Scheduler are summarized in figure 21.

Within this setup we could start asking a lot of questions. For example,

1. Can we find a predicate R over the unprimed variables characterizing exactly the states that can be reached from I? R is called the set of *reachable states*.

2. How many reachable states are there?

3. Is it the case that in all reachable states only one token is present?

4. Is task t_i always only started after t_{i-1}?

$$
\begin{array}{lll}
\mathit{unchanged}_S & =_{\text{def}} & \bigwedge_{x\in S} x = x' \\
\mathit{assigned}_{S'} & =_{\text{def}} & \mathit{unchanged}_{\vec{x}\setminus S'} \\
P_i & =_{\text{def}} & \quad (c_i \wedge \neg t_i \wedge t'_i \wedge \neg c'_i \wedge h'_i \wedge \quad \mathit{assigned}_{c_i,t_i,h_i}) \\
 & & \vee\ (h_i \wedge \quad c'_{i \bmod N+1} \wedge \neg h'_i \wedge \mathit{assigned}_{c_{i \bmod N+1},h_i}) \\
E_i & =_{\text{def}} & t_i \wedge \neg t'_i \wedge \mathit{assigned}_{t_i} \\
T & =_{\text{def}} & \bigvee\limits_{1\le i\le N} P_i \vee E_i \\
I & =_{\text{def}} & \neg\vec{t} \wedge \neg\vec{h} \wedge c_1 \wedge \neg c_2 \wedge \cdots \wedge \neg c_N
\end{array}
$$

Figure 21: Milner's Scheduler as described by the transition predicate T and the initial-state predicate I.

5. Does Milner's Scheduler possess a deadlock? I.e., is there a reachable state in which no transitions can be taken?

To answer these questions we first have to compute R. Intuitively, R must be the set of states that either satisfy I (are initial) or within a finite number of T transitions can be reached from I. This suggest an iterative algorithm for computing R as an increasing chain of approximations $R^0, R^1, \ldots, R^k, \ldots$ Step k of the algorithm find states that with less than k transitions can be reached from I. Hence, we take $R^0 = 0$ the constantly false predicate and compute R^{k+1} as the disjunction of I and the set of states which from one transition of T can be reached from R^k. Figure 22 illustrates the computation of R.

How do we compute this with ROBDDs? We start with the ROBDD $R = \boxed{0}$. At any point in the computation the next approximation is computed by the disjunction of I and T composed with the previous approximation R'. We are done when the current and the previous approximations coincide:

```
REACHABLE-STATES(I, T, x⃗, x⃗′)
    R ← [0]
    repeat
        R′ ← R
        R ← I ∨ (∃x⃗. T ∧ R)[x⃗/x⃗′]
    until R′ = R
    return R
```

7.1 Knights tour

Using the same encoding of a chess board as in section 6.1, letting $x_{ij} = 1$ denote the presence of a Knight at position (i,j) we can solve other problems. We can encode moves of a Knight as transitions. For each position, 8 moves are possible if they stay on the board. A Knight at (i,j) can be moved to any one of $(i\pm1, j\pm2), (i\pm2, j\pm1)$ assuming they are vacant and within the board boundary. For all i,j and k,l with $1 \le k,l \le N$ and $(k,l) \in \{(i\pm1, j\pm2), (i\pm2, j\pm1)\}$:

$$M_{ij,kl} =_{\text{def}} x_{ij} \wedge \neg x_{k,l} \wedge \neg x'_{ij} \wedge x'_{kl} \wedge \bigwedge_{(i',j')\notin\{(i,j),(k,l)\}} x_{i'j'} = x'_{i'j'}\ .$$

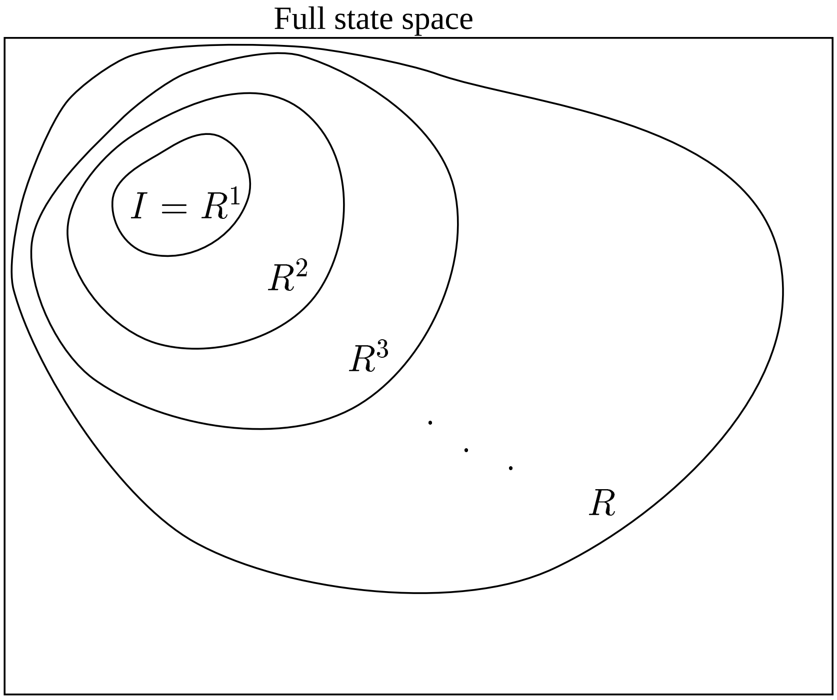


Figure 22: Sketch of computation of the reachable states

Hence, the transitions are given as the predicate $T(\vec{x}, \vec{x}')$:

$$T(\vec{x}, \vec{x}') =_{\text{def}} \bigvee_{1 \leq i,j,k,l \leq N, (k,l) \in \{(i \pm 1, j \pm 2), (i \pm 2, j \pm 1)\}} M_{ij,kl}$$

Exercise 7.1 (Knight's tour) Write a program to solve the following problem using the ROBDD-package: Is it possible for a Knight, positioned at the lower left corner to visit all positions on an $N \times N$ board? (*Hint:* Compute iteratively all the positions that can be reached by the Knight.) Try it for various N.

Exercise 7.2 Why does the algorithm REACHABLE-STATES always terminate?

Exercise 7.3 In this exercise we shall work with Milner's Scheduler for $N = 4$. It is by far be the most convenient to solve the exercise by using an implementation of an ROBDD package.

- **a)** Find the reachable states as an ROBDD R.
- **b)** Find the number of reachable states.
- **c)** Show that in all reachable states at most one token is present on any of the placeholders $c_1, \ldots, c_N$ by formulating a suitable property P and prove that $R \Rightarrow P$.

d) Show that in all reachable states Milner's Scheduler can always perform a transition, i.e., it does not possess a *deadlock*.

Exercise 7.4 Complete the above exercise by showing that the tasks are always started in sequence $1, 2, \ldots, N, 1, 2 \ldots$

Exercise 7.5 Write a program that given an N as input computes the reachable states of Milner's Scheduler with N cyclers. The program should write out the number of reachable states (using SATCOUNT). Run the program for $N = 2, 4, 6, 8, 10, \ldots$ Measure the running times and draw a graph showing the measurements as a function of N. What is the asymptotic running time of your program?

8 Project: An ROBDD Package

This project implements a small package of ROBDD-operations. The full package should contain the following operations:

Init(n)
Initialize the package. Use n variables numbered 1 through n.

Print(u)
Print a representation of the ROBDD on the standard output. Useful for debugging.

Mk(i, l, h)
Return the number u of a node with $var(u) = i, low(u) = l, high(u) = h$. This could be an existing node, or a newly created node. The reducedness of the ROBDD should not be violated.

Build(t)
Construct an ROBDD from a Boolean expression. You could restrict yourself to the expressions x or $\neg x$ or finite conjunctions of these. (Why?)

Apply(op, u_1, u_2)
Construct the ROBDD resulting from applying op on u_1 and u_2.

Restrict(u, j, b)
Restrict the ROBDD u according to the truth assignment $[b/x_j]$.

SatCount(u)
Return the number of elements in the set $sat(u)$. (Use a type that can contain very large numbers such as floating point numbers.)

AnySat(u)
Return a satisfying truth assignment for u

Sub-project 1

Implement the tables T and H with their operations listed in section 4. On top of these implement the operations INIT(n), PRINT(u), and MK(i, l, h).

Sub-project 2

Continue implementation of the package by adding the operations $\textsc{Build}(t)$ and $\textsc{Apply}(op, u_1, u_2)$.

Sub-project 3

Finish your implementation of the package by adding $\textsc{Restrict}(u, j, b)$, $\textsc{SatCount}(u)$, and $\textsc{AnySat}(u)$.

References

[AH97] Henrik Reif Andersen and Henrik Hulgaard. Boolean expression diagrams. In *Proceedings, Twelfth Annual IEEE Symposium on Logic in Computer Science*, pages 88–98, Warsaw, Poland, June 29–July 2 1997. IEEE Computer Society.

[Bry86] Randal E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, 8(C-35):677–691, 1986.

[Bry92] Randal E. Bryant. Symbolic Boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, September 1992.

[CBM89] Olivier Coudert, Christian Berthet, and Jean Christophe Madre. Verification of synchronous sequential machines based on symbolic execution. In J. Sifakis, editor, *Automatic Verification Methods for Finite State Systems. Proceedings*, volume 407 of *LNCS*, pages 365–373. Springer-Verlag, 1989.

[CLR90] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms.* McGraw-Hill, 1990.

[Coo71] S.A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on the Theory of Computing*, pages 151–158, New York, 1971. Association for Computing Machinery.

[Mil89] Robin Milner. *Communication and Concurrency.* Prentice Hall, 1989.