A Generalized Reduction of Ordered Binary Decision Diagram

Joan Thibault

May 31, 2017

Abstract

Reduced Ordered Binary Decision Diagram ((RO)BDD) [2,12] are the state-of-the-art representation for Boolean functions. They are used in various fields such as logic synthesis, artificial intelligence or combinatorics. However, BDDs suffer from to main issues: (1) their representation is memory expansive and (2) their manipulation is memory intensive as it induces many random memory accesses.

Various variations of ROBDD exist such as Zero-suppressed Decision Diagram (ZDD) [11], Multi-valued Decision Diagram (MDD) [8, 9] and variations of the reduction rules such as "output inverter" [1], "input negation" [10], "shifting variables" [10], "dual edges" [7] or "copy node" [6].

In this report, we introduce a new generalization of the standard reduction rules that we call the "extraction of useless variables" or "extract U" for short. Basically, it detects useless variables (i.e. variables which have no influence on the result of a given function) and extracts them from the local variable order. This generalization allows to add/remove useless variables in linear time (in the number of variables), reduces the number of nodes and tends to reduce the overall memory cost. However, several drawbacks arise: no in-place sifting (permutation of adjacent variables), bigger nodes of variable size and it slightly complexifies manipulations.

We implemented both the "extract U" and the "output negation" variants in an OCaml program and tested it against several benchmarks [3, 5, 13]. We observe an average of 25% less nodes and 32% less memory when representing circuits, and 3% less nodes and memory when representing solutions from generated CNF formulas.

Contents

1 Introduction

Nowadays, it exists many critical systems which rely on digital circuits: in transportation (e.g. cars, train, plains) , communication (e.g. satellites), computation (e.g. data centers, super-computers), exploration (e.g. space rocket, rovers). One way of minimizing risks in digital parts of these systems is to provide a formal proof that they respect their specification. On the other hand, we want to minimize costs and energy consumption while maximizing performances of these digital circuits. In order to efficiently optimize digital circuits we rely on complex programs. However, these programs are rarely proven themselves, thus, circuits optimized using them might not be equivalent to the initial design, therefore, might not respect the specification. The obvious solution would be to prove optimizing programs, however two majors issues arise : these programs are complex (thus, proving them would be expansive) and might be proprietary (thus, one cannot check that the proof is correct). A simpler alternative is to design a program which check that two digital circuits are equivalent. With this alternative, the only piece of software which needs to be proven is the "equivalence checker".

In order to prove that two digital circuits are equivalent, their is two main algorithmic solutions: Firstly, the DPLL (Davis–Putnam–Logemann–Loveland) algorithm. This backtracking procedure is usually implemented with various heuristics such as *unit propagation*, early conflict detection or *conflict driven clause learning*. Secondly, the compilation of both circuits into Reduced Ordered Binary Decision Diagrams (ROBDDs). A ROBDD is a canonical structure which represent a function, thus, once compiled, the identity test can be performed in constant time. However, the compilation might take an exponential time in the number of variable. In this report we will focus on ROBDDs.

ROBDD have various other applications such as : Bounded Model Checking, Planning, Software Verification, Automatic Test Pattern Generation, Combinational Equivalence Checking, Combinatorial Interaction Testing, etc.

However, BDDs are memory expansive as their size tends to grow exponentially with the number of variables. Various variants have been invented in order to capture some semantic properties of the function and reduce the memory consumption. For example, Zero suppressed binary Decision Diagram (ZDD) are better suited for representing sparse functions. In this report we will use the "output inverter" variant [1], which extends the reduction rules in order to guarantee canonicity under negation. Thus, in addition to reduce the size of the structure, allows to negate a function in constant time (reducing the set of useful binary operators to XOR and AND). Other extensions of the reduction rules exist such as: "input negation" [10] (each edge can complement the first locally first input), "shifting variables" [10] (each edge store the number of useless variables before the next significant variables) or "dual edge" [7] (we define the dual of a function f by $\bar{f} = X \longrightarrow \bar{f}(\bar{X})$, therefore the reduction works similarly to the "output inversion").

In addition to use the "output inverter" variant, we introduce a new variant which allows to extract useless variables (a.k.a non-support variables). A useless variables, is a variable which does not change the results such as x_1 in $f(x_0, x_1, x_2) = x_0 \wedge x_2$ or x_0 in $g(x_0) = x_0 \wedge \neg x_0$. We call this new variant "extraction useless variables" or "U-extract" for short.

This report will be organized as follows. In Section 1, we formally introduce

Boolean functions and useless variable. In Section 2, we introduce ROBDDs. In Section 3, we introduce the "U-extract" variant and prove that it maintains the canonicity. In Section 4, we expose an estimations of improvements on three different benchmarks [3, 5, 13] using our implementation in OCaml.

2 Notations

Reduced Ordered Binary Decision Diagrams represent Boolean functions. In this section we introduce notations necessary to their manipulation.

We denote the set of Booleans $\mathbb{B} = \{0, 1\}$. The set of Boolean vector of size $n \in \mathbb{N}$ is denoted $\mathbb{B}^n$. The set of Boolean functions of arity $n \in \mathbb{N}$ is denoted $\mathbb{F}_n = \mathbb{B}^n \longrightarrow \mathbb{B}$.

We denote conjunction by $\wedge$, disjunction by $\vee$, negation by $\neg$. We denote $\longrightarrow_S$, the Shannon operator (n.b. $x \longrightarrow_S y, z = (\neg x \wedge y) \vee (x \wedge z)$).

Restriction

Let $f \in \mathbb{F}_{n+1}$ be a Boolean function of arity $n+1$, i $(0 \leq i < n+1)$ be an integer and $b \in \mathbb{B}$ be a Boolean. We denote $f[i \leftarrow b]$ the Boolean function of arity n defined by $f[i \leftarrow b](x_1, \ldots, x_n) = f(x_0, \ldots, x_{i-1}, b, x_i, \ldots, x_n)$. $f[i \leftarrow 0]$ (respectively $f[i \leftarrow 1]$) is called the i-th negative (respectively positive) restriction.

For each Boolean function f of arity $n{+}1$, we denote $f[i \leftarrow b] = (x_1, ..., x_n) \rightarrow f(x_1, .., x_{i-1}, b, x_i, ..., x_n)$ the function of arity n called the i-th positive restriction of f if $b = 1$, the negative one otherwise.

Let f be a function of arity $n+1$, we denote $f_0 = f[0 \leftarrow 0]$ (respectively $f_1 = f[0 \leftarrow 1]$) the function of arity n.

Construction

Let $f, g \in F_n$ be Boolean functions of arity n and i $(0 \leq i < n+1)$ be an integer. We denote $f \star_i g$ the Boolean function of arity $n+1$ defined by $(f \star_i g)(x_0, \ldots, x_{i-1}, y, x_i, \ldots, x_n) = y \longrightarrow_S f(x_0, \ldots, x_n), g(x_0, \ldots, x_n)$.

We denote $\star = \star_0$

Nb: $(f \star_i g)[i \leftarrow 0] = f$ and $(f \star_i g)[i \leftarrow 1] = g$ $((f \star g)_0 = f$ and $(f \star g)_1 = g)$.

Expansion Theorem

Let f be a function of arity n, then $\forall i, 0 \leq i < n \Rightarrow f = f[i \leftarrow 0] \star_i f[i \leftarrow 1]$ (in particular $f = f_0 \star f_1$)

Restriction Distributivity

- $(\neg f)[i \leftarrow b] = \neg f[i \leftarrow b]$
- $(f \wedge g)[i \leftarrow b] = f[i \leftarrow b] \wedge g[i \leftarrow b]$
- $(f \vee g)[i \leftarrow b] = f[i \leftarrow b] \vee g[i \leftarrow b]$

Useless Variables

Let f be a function of arity n, and i be an integer $(0 \leq i < n)$. The i-th variable of f is said useless iff $f[i \leftarrow 0] = f[i \leftarrow 1]$

3 Reduced Ordered Binary Decision Diagram (ROBDD) and Canonicity

Definition of ROBDD

A Reduced Ordered Binary Decision Diagram is a directed acyclic graph $(V \cup T, \Psi \cup E)$ representing a vector of Boolean functions $F = (f_1, ..., f_k)$. Nodes are partitioned into two sets : the set of internal nodes V and the set of terminal nodes T. Every internal node $v \in V$ has one field var, which represent the index of a variable and two outgoing edges respectively denoted *then* and *else*. When using the "output inverter" variant, there is only one terminal called 0, which represent the functions which always return 0. Arcs are partitioned into two sets : the set of root arcs Ψ and the set of internal arcs E. There is exactly k root arcs, a root arc is denoted Ψ_i with $0 \leq i < k$, informally, Φ_i is the root of the ROBDD representing f_i. Every arc has an inversion field $neg \in \mathbb{B}$ and a destination node denoted *node*.

We denote $\phi(node)$ the semantic of the node *node* and $\psi(arc)$ the semantic of the arc *arc* as follow:

- $\phi(0 \in T) = 0$
- $\psi(arc \in \Psi \cup E) = arc.neg \oplus \phi(arc.node)$
- $\phi(node \in V) = node.var \longrightarrow_S \psi(node.then), \psi(node.else)$

We set that $\forall iarc_i \in \Psi, \psi(arc_i) = f_i$.

We define the equivalence relation $=$ by : $e_1 = e_2$ if $e_1.node = e_2.node$ and $v_1 = v_2$ iff $v_1.var = v_2.var \wedge v_1.then = v_2.then \wedge v_1.else = v_2.else$ (nb. $1 = 1$ and $0 = 0$).

Definition of ROBDD

A BDD is said `ordered` if (1) $\forall v \in V$, $v.then.node \in V \Rightarrow v.var > v.then.node.var$ and $v.else.node \in V \Rightarrow v.var > v.else.node.var$.

A BDD is said `reduced` if (2) $\forall v \in V v.then \neq v.else$ and (3) every node has an in-degree strictly positive.

Theorem : ROBDD are canonical

Let consider a ROBDD G representing $F = (f_1, ..., f_n)$ over a set of n variables $x_n < x_{n-1} < ... < x_1$. Then, for every nodes $v_1, v_2 \in G$, $\phi(v_1) = \phi(v_2) \Leftrightarrow v_1 = v_2$.

Proof

The proof is by induction on n the number of variables appearing in the representation.

Initialization

if $n = 0$ then there is no internal nodes, then F is a vector of constant functions whose representation is clearly unique (4 cases : $(0 \in F) \times (1 \in F)$). The constant function 0 (respectively 1) is represented by an arc to the terminal node 0 (respectively 1)

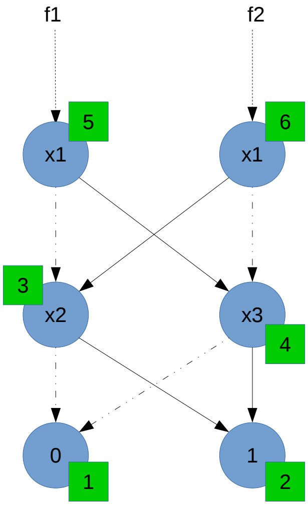

Figure 1: For each internal node, the outgoing full (respectively dashed) arrows represent the `then` (respectively `else`) arc which is followed when the variable is true (respectively false). Each numbered green square represent the identifier affected to each node during the construction process

Induction

We assume the theorem true for any $0 \leq k < n$.

Let f be a component of F. We can write $f = x_n \longrightarrow_S f[x_n = 1], f[x_n = 0]$, both $f[x_n = 1]$ and $f[x_n = 0]$ are uniquely determined by f and have unique representation. Suppose that f does not depend on x_n then $f = f[x_n = 1] = [x_n = 0]$ which is unique (and identical to the representation of $f[x_n = 1]$). Indeed the condition (2) in the definition of a ROBDD, prevent the representation of f from containing a node with a *var* attribute x_n. Suppose now that f does depend on x_n, then using the induction hypothesis its representation is unique. The condition (3) simply guarantees that the representation of F consists exclusively of the representations of its components, which are unique.

4 Basic Manipulation of Binary Decision Diagram (BDD)

4.1 Effective construction

In practice one does not build the decision tree and then reduces it. Rather, BDDs are created starting from the BDDs for the constants and the variables by application of the usual Boolean connectives and are kept reduced at all times. At the same time several functions are represented by one multi-rooted diagram. Indeed, each node of a BDD has a function associated with it. If we have several functions, they will have subfunctions in common. For instance, if we have $f(x_0,x_1,x_2,x_3) = x_1 \longrightarrow_S x_2, x_3$ and $f(x_0,x_1,x_2,x_3) = \neg x_1 \longrightarrow_S x_2, x_3$, we represent them like in figure. As a special case two equivalent functions are represented by the same BDD (not just two identical BDDs). This approach makes equivalence check a constant-time operation. Its implementation is usually based on a dictionary of all BDDs nodes in existence in an application. This dictionary is called the *unique table*. Operations that build BDDs start from the bottom (the constant nodes) and proceed up to the function nodes. Whenever an operation needs to add to a BDD that it is building, it knows already the two nodes (say f_1 and f_0 represented by some identifier (usually implemented as pointers)) that going to be the new node's children and the decision variable v so it just has to check if the node (v, f_1, f_0) already exist and if so return its identifier and if not generate a new identifier and return it. Doing so, the equivalence check is reduced to pointer comparison.

Operator `CONS`

Pseudo-code for dynamic construction of unique nodes.

```
CONS(var, then, else){
  if(then.node = else.node){
    return then
  }else if (mynode = node(var, then, else) in unique table){
    return mynode's identifier
  }else{
    id = new identifier
    store (mynode, id) in unique table
    return arc(node = id)
  }
}
```

4.2 Operators

The usual way of generating new BDDs is to combine existing BDDs using operators such as conjunction AND, disjunction OR, symmetric difference (XOR). As starting point one takes the simple BDDs for the function $f_i = x_i$, for all the variables in the functions of interest. We are therefore interested in an algorithm that given BDDs for f and g, will build the BDDs for $f\ op\ g$, where op is a binary operator (a Boolean function of two arguments). The basic idea comes from the `expansion theorem`, since :

$$f\ op\ g = f[i \leftarrow 1] \star_i f[i \leftarrow 0]\ op\ g[i \leftarrow 1] \star_i g[i \leftarrow 0]$$

then

$$f\ op\ g = (f[i \leftarrow 1]\ op\ g[i \leftarrow 1]) \star_i (f[i \leftarrow 0]\ op\ g[i \leftarrow 0])$$

Therefore, computing $f\ op\ g$ can be performed inductively on the tree structure.

Moreover, if we keep track of previous computation (this process is called memoization), we got a linear algorithm in the product of f and g representation's size.

In order to be complete we have to implement NOT and AND (and XOR)

Example : operator AND
The terminal cases for this operator are:

- $AND(f, 0) = AND(0, f) = 0$
- $AND(f, 1) = AND(1, f) = f$
- $AND(f, f) = f$

These conditions can be computed in constant time, then we use the `memoization table` to check if the result has already been computed, and if not we apply the `expansion theorem` and solve the problem inductively.

```
AND(f, g){
  if f > g{
    return AND(g, f)
  }else if(terminal case){
    return terminal((f, g))
  }else if (memoization table has entry (f, g) {
    return memoization((f, g))
  }else{
    let x be the top variable of {f, g}
    f1, f0 = cofactor(x, f)
    //if f does not depends on x, then f1=f0=f
    g1, g0 = cofactor(x, g)
    //idem for g
    //but either f or g has to depend on x
    r1 = AND(f1, g1)
    r0 = AND(f0, g0)
    r = CONS(x, r1, r0)
    insert {key = (f, g); value = r} in memoization
    return r
    }
}
```

NB: computing $NOT(f)$ can be done in linear time and space in the number of node in the representation of f.

Operator XOR Very similar to AND with some differences in terminal case:

- $XOR(0, f) = XOR(f, 0) = f$
- $XOR(1, f) = XOR(f, 1) = NOT(f)$
- $XOR(f, f) = 0$

5 Complemented Arcs Compression

5.1 Motivation

We may notice that the respective representations of f and $\neg f$ are very similar. The only difference being the values of the leaves that interchanged. This suggest the possibility of actually using the same sub-graph to represent both f and $\neg f$.

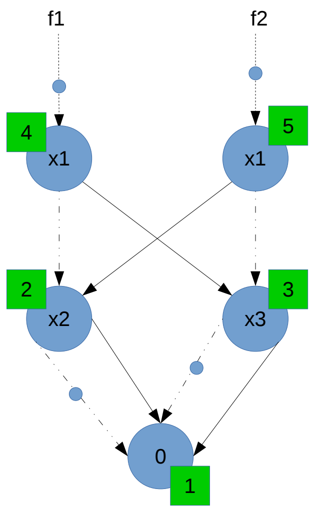

Figure 2: Same BDD as in Figure 1, when introducing complemented arcs (represented by an with a small circle) and maintaining the reduction rule that then arcss (represented by full arrows) can't be complemented arcs.

Suppose we have a BDD representing f, then in order to represent $\neg f$ we just have to remember that the function we have in the multi-rooted graph is complement of f. This can be performed by attaching a new attribute *comp* (standing for complemented) on the arc pointing to the top node of f. This attribute can have two meaning: $x \to x$ or $x \to \neg x$. Arcs with *comp* set to $x \to x$ are called regular arcs. Arcs with *comp* set to $x \to \neg x$ are called complemented arcs.

The use of Complemented arcs slightly complexify (e.g. we have to add a forth reduction rule) the manipulation of BDDs, but has three advantages:

- reduce memory usage (optimally by a factor 2).
- negation can be performed in constant time.
- checking $g = \neg f$ when having g and f can be performed in constant time.

5.2 Change in representation

We add a *comp* (standing for complemented) field on arcs with value in Boolean. We add a new reduction rule (4) : $\forall v \in V, v.then.comp = 1$. We slightly change the definition of semantics:

- $\psi(e) =$ `if` $e.comp = 1$ `then` $\phi(e.node)$ `else` $\neg\phi(e.node)$

We remove the terminal node 1. The constant function 0 is represented by an arc with its *comp* field set to *true* and its *node* field set to terminal node 0. The constant function 1 is represented by an arc with its *comp* field set to $false$ and its *node* field set to terminal node 0.

5.3 Canonicity

The proof is not very different from the previous one on regular BDDs but still has minor changes:

- In the initialization paragraph we to distinguish fewer cases as their is only one terminal node 0.
- In the recurrence step: we need to distinguish the case in which the outgoing arc of the function node for $f[x_n = 1]$ is regular from the case in which it is complemented. Suppose the arc is regular. Then the representation of f must consist of a regular arc pointing to a node labeled x_n with children representing $f[x_n = 1]$ and $f[x_n = 0]$. Suppose the arc is complemented. Then the representation of f must consist of a complement arc pointing to a node labeled x_n with children representing $\neg f[x_n = 1]$ and $\neg f[x_n = 1]$. Hence, in both cases it is unique by the induction hypothesis and the first condition of the theorem.

5.4 More complex manipulations

5.4.1 Effective reduction

Operator `CONS`

Pseudo-code for dynamic construction of unique nodes.

```
CONS(var, then, else){
  if(then.node = else.node & then.comp = else.comp){
    return then
  }else{
    then' = arc(comp = true, node = then.node)
    else' = arc(comp = (then.comp = else.comp), node = else.node)
    id = if (mynode = node(var, then', else') in unique table){
      return mynode's identifier
    }else{
      return new identifier
    }
        return arc(comp = then.comp, node = id)
  }
}
```

5.4.2 Operator computation

Operator NOT Computing NOT is reduced to turn an regular (respectively complemented) arc into a complemented (respectively regular) one. Which can be performed in constant time.

Operator AND We can add a new terminal case:

- $AND(f, \neg f) = AND(\neg f, f) = 0$

Operator XOR Computing NOT in constant time improve the practical run time of XOR but does not change its complexity class. Moreover, using the property : $XOR(\neg x, y) = \neg XOR(x, y)$, we can save some constants.

6 Introduction of Reduced Decision Tree (RDT) and Canonical Shared Forest (CSF)

6.1 Motivation

Adding attribute on arcs allowed us to (slightly) reduce computation time and space. In order to go further in this direction we have to introduce more formalism. In order to simplify reasoning we will split the current structure in two:

- the Canonical Shared Forest (CSF), which job is to ensure structural canonicity (i.e. attribute an identifier to every sub-trees and ensure that any identical sub-trees have the same identifier).
- the Reduced Decision Tree (RDT), which job is to ensure semantic canonicity (i.e. ensure that the semantic of operation is correct and that a function has exactly one representation)

In the following sections we describe the role of both these structures and how they work together.

6.2 From the `unique table` to the Canonical Shared Forest (CSF)

The Canonical Shared Forest (CSF) encapsulates the `unique table` and the `CONS` operator in one structure. Its job is to hide from the RDT the whole memory management. In its basic implementation interactions are reduced to two methods:

- `push` (previously called `CONS`) that takes a node and returns a unique identifier.
- `pull` that take an identifier and return the associated node.

A more complex implementation could use some commands in order to specify when a node is no longer required in order to save memory. In order to save time and extend memory we can think about using distributed memory units, efficiently using of the memory hierarchy or allow concurrent/parallel accesses. However such implementation details fall out of the scope of this lecture.

NB: Implementing the CSF can lead to theoretical issues (especially when dealing with distributed, concurrent or parallel versions) but they are not specific to our problem.

6.3 From Shannon's Decision Tree to Reduced Decision Tree (RDT)

The Reduced Decision Tree (RDT) is the theoretical (and computational) core of the compression system. The basic is to represent sets of somehow related functions by a common graph instead of distinct ones.

6.3.1 Change in representation

Definition : Syntax
We define a Compressed BDD by a graph $G = (V \cup T, \Psi \cup E)$ where:

- there are two sets of vertices:
  - V, the set of internal nodes. Internal nodes have an out-degree of two : its sons are called *then* and *else*.
  - T, the set of terminal nodes. Terminal nodes have an out-degree of zero.
- there are two set of arcs:
  - E the set of internal arcs which have a source node called *source*.
  - Ψ the set of root arcs which has no source node.

  arcs in both sets have a destination called *node* and carry some data called $coreduce$.

Definition : Semantic
We denote $\phi(node)$ the semantic of the node *node* and $\psi(arc)$ the semantic of the arc *arc* as follow:

- $\forall node \in T$, $\phi(node) = f_{node}$ (usually $\phi(0) = () \to 0 \in \mathbb{B}^0 \to \mathbb{B}$)
- $\phi(node) = \psi(node.then) \star_i \psi(node.else)$
- $\psi(arc) = \rho(arc.coreduce)(\phi(arc.node))$ for some function ρ such that $\rho(arc.coreduce) : (\mathbb{B}^k \to \mathbb{B}) \to (\mathbb{B}^n \to \mathbb{B})$ with $k \leq n$. k (respectively n) is called the in-arity (respective out-arity) of *arc.coreduce*.

Reduction Rules A CBDD is said syntactically reduced if it satisfies (1) some local property P on nodes (and their two outgoing arcs) (2) there is no identical (up to isomorphism) sub-graph.

A CBDD is said semantically reduced if $\forall v_1, v_2 \in V \cup T, \phi(v_1) = \phi(v_2) \Rightarrow v_1 = v_2$.

Canonicity We say that the local property P ensures pseudo-canonicity if any syntactically reduced CBDD is semantically reduced.

We say that the local property P ensure full-canonicity (or the canonicity) if for all functions f, every representations of f are equal up to graph isomorphism.

6.3.2 Reduction in constructor

`RDT.CONS`'s goal is to ensure that P is verified when building a new internal node. We may notice that it has three distinct behaviors:

1. the result can be build only with constant nodes and the two available nodes.
2. the result needs a node that has already been build.
3. the result needs a new node.

In both cases 2 and 3, an access to CSF is needed.

```
RDT.CONS(arc1, arc0){
  match RDT-consensus(then, else, cmp) with
    | Solved result ⟶ result
    | Partial (coreduce, reduce) ⟶
      arc(coreduce, CSF.PUSH(reduce))
}
```

`RDT.SPLIT` performs the reverse operation of `RDT.CONS`. We may notice that it has two distinct behaviors:

- the result can be computed without accessing to CSF
- the result is computed accessing CSF.

```
RDT.SPLIT(arc)
  if(terminal case){
    return result
  }else{
    reduce = CSF.PULL(arc.reduce-ident)
    arc1 = compose(arc.coreduce, reduce.then)
    arc0 = compose(arc.coreduce, reduce.else)
    return (arc1, arc0)
  }
}
```

6.3.3 Reduction in operators

In order to compute AND, there are cases:

- its a terminal case, so we can solve it immediately
- its a memoized case, so we can return the result from memory
- We can split the problem into two sub-problems (which can be solved inductively).

We slightly "rotate" these choices in order to abstract the memoization process

```
AND-SOLVE(reduced-problem){
  arcX, arcY = and-unpack(reduced-problem)
  arcX1, arcX0 = RDT.SPLIT(arcX)
  arcY1, arcY0 = RDT.SPLIT(arcY)
  arc1 = match and-solver(arcX1, arcY1) with
    | Solved result ⟶ result
    | Partial (coreduce, subproblem) ⟶
        compose(coreduce, AND-SOLVE(subproblem))
  arc0 = match and-solver(arcX0, arcY0) with
    | Solved result ⟶ result
    | Partial (coreduce, subproblem) ⟶
        compose(coreduce, AND-SOLVE(subproblem))
  return RDT.CONS(arc1, arc0)
}
```

`AND-SOLVE` is a recursive memoized algorithm (i.e. when AND-SOLVE(problem) returns value, we store an entry into a dictionary and when calling AND-SOLVE, we first look into the dictionary to know if the result is already available and if so return it).

```
RDT.AND(arcX, arcY){
  match and-solver(arcX, arcY) with
    | Solved result ⟶ result
    | Partial (coreduce, subproblem) ⟶
        compose(coreduce, AND-SOLVE(subproblem))
}
```

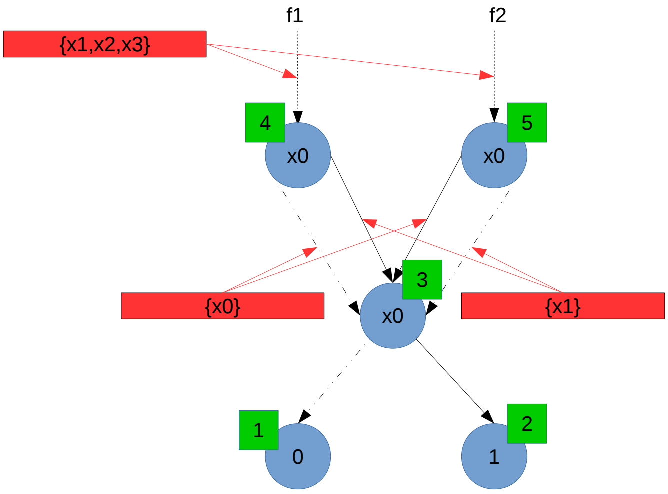

Figure 3: Same BDD as in Figure 1, when detecting and removing useless variables. Each red rectangle contain the support set of each arc's function.

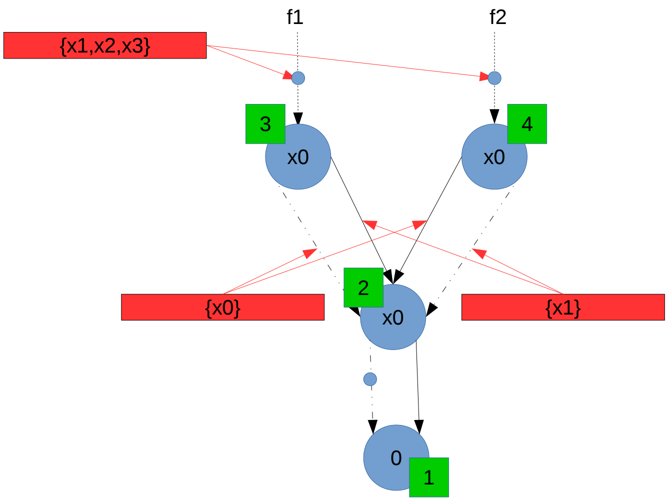

Figure 4: Same BDD as in Figure 2 and 3, with both additional reduction rules : complemented arcs and extraction of useless variables.

7 Pattern extraction : useless variables

7.1 Motivation

We may notice that the respective representations of $f_1 = f(x_{i_1}, ..., x_{i_n})$ $(i_1 < ... < i_n)$ and $f_2 = f(x_{j_1}, ..., x_{j_n})$ $(j_1 < ... < j_n)$ are very similar. The only difference being the values of the *var* field of each nodes within the representation of f. This suggest the possibility of actually using the same sub-graph to represent f, f_1 and f_2.

Suppose we have a BDD representing f, then in order to represent f_1 we just have to remember which variable are in the support of f (the relative order of variables in the support does not change). In this attribute, we will store the support set in the *coreduce* field of arcs.

Removing useless variables slightly complexify (e.g. we have to add an other reduction rule) the manipulation of BDDs, but has three advantages:

- reduce memory usage (in some cases there is an exponential gain, but on most cases its less significant).
- copying a function (if relative order of variables is unchanged) can be performed in constant time.
- checking $g = f$ modulo their useless variables -when having g and f- can be performed in constant time (not very useful).

7.2 Proof of Canonicity

We define the following reductions rules:

- $\forall v \in V, v.then.coreduce \cup v.else.coreduce = \{0, ..., n\}$
- $\forall v \in V \cup T, \phi(v)$ has no useless variable

For every function f, we denote $\tilde{f}$ the restriction of f to its support (maintaining the relative order of the remaining variables).

Existence Let $f \in \mathbb{B}^n \to \mathbb{B}$ be a Boolean Function. We recursively build a valid representation $R(f)$ of f which verify the reduction rules:

- $R((x_1, ..., x_n) \to b) = arc(coreduce = \emptyset, node = 1)$, where $b \in \mathbb{B}$
- $R(f) =$
  - We denote $S = \{i_1 < ... < i_k\}$ the support set of f.
  - $R(f) = arc(coreduce = S, node = node(then = R(\tilde{f}[0 \leftarrow 1]), else = R(\tilde{f}[0 \leftarrow 0])))$

Uniqueness We prove canonicity by induction on n the arity of the represented function.

Initialization Let R be a representative of a function of arity 0. We consider the root arc a, so $a.coreduce = \emptyset$ (the only set of cardinality lower than zero). However the `in-arity` of $\emptyset$ is 0, so $a.node$ represent a constant function. Hence either $R = arc(coreduce = \emptyset, node = 0)$ representative of the constant function 0 or $R = arc(coreduce = \emptyset, node = 1)$ representative of the constant function 1.

Induction Assuming that the hypothesis holds for every $0 \leq k \leq n$. Let R be a representative of f, we denote a its root arc.

- We assume that $a.coreduce \neq \{0, ..., n\}$. Then the arity of $g = \phi(a.node)$ is lower strictly lower than n. However g has a unique representation (recurrence hypothesis) and g has no useless variable (because of the reduction rules on applied on node $a.node$). So the only useless variable in f are the one that does not appear in $a.coreduce$. Therefore R is unique.
- We assume that $a.coreduce = \{0, ..., n\}$. We denote a_1 (respectively a_0) the root arc of the representation (unique by recurrence hypothesis) of $f_1 = f[0 \leftarrow 1] = \psi(a.node.then)$ (respectively $f_0 = f[0 \leftarrow 0] = \psi(a.node.else)$).
  - We assume that $f_1 = f_0$ then $a_1 = a_0$ then either induce that $0 \notin a.coreduce$ (which induce a contradiction with $S = \{0, ..., n\}$) or R does not satisfies the reduction rules.
  - Therefore $f_1 \neq f_0$.
    - We assume that $a_1.coreduce \cup a_0.coreduce \neq \{0, ..., n-1\}$. Then it exist $0 \leq i \leq n-1$ such that x_i is useless for both f_1 and f_0. Hence $i+1$ is useless for f. So R does not satisfies the reduction rules.
    - Therefore $a_1.coreduce \cup a_0.coreduce = \{0, ..., n-1\}$. Then $a = arc(coreduce = \{0, ..., n\},\ node = node(then = a_1,\ else = a_0)$. Therefore R is unique.

7.3 Reduction in construction

Moreover, as we already define the main algorithm for construction, we just have to write the specific sub-routines:

```
RDT-consensus(then, else, cmp){
  if (cmp = 0) & then.coreduce = else.coreduce then{
    we denote {i_1 < ... < i_k} = then.coreduce
    Solved arc(then = {i_1 + 1 < ... < i_k + 1}, node = then.node)
  }else{
    we denote U = {i_1 < ... < i_k} = then.coreduce ∪ else.coreduce
    we denote A = {a_1 < ... < a_l} the indexes of then.coreduce in
        U.
    we denote B = {b_1 < ... < b_m} the indexes of else.coreduce in
        U.
    Partial (coreduce = {0, i_1 + 1, ..., i_k + 1}, reduce = node(
        then = arc(coreduce = A, node = then.node), else =
        arc(coreduce = B, node = else.node) ) )
```

```
  }
}
```

```
compose({i_1 < ... < i_n}, arc(coreduce = {j_1 < ... < j_k}, node =
    node){
  return arc(coreduce = {i_{j_1} < ... < i_{j_k}}, node = node)
}
```

7.4 Reduction in operators

Moreover, as we already define the main algorithm for computing AND, we just have to write the specific sub-routines:

```
and-unpack(reduced-problem){
  return (reduced-problem.arcX, reduced-problem.arcY)
}
```

```
and-solver(arcX, arcY){
  if (terminal cases){
    return result //removing useless variables does not
        add terminal cases
  }else{
    we denote U = {i_1 < ... < i_k} = then.coreduce ∪ else.coreduce
    we denote A = {a_1 < ... < a_l} the indexes of then.coreduce in
        U.
    we denote B = {b_1 < ... < b_m} the indexes of else.coreduce in
        U.
    if arcX ≤ arcY
    then{
      Partial (coreduce = U, subproblem = (arcX = arc(
          coreduce = A, node = arcX.node), arcY = arc(
          coreduce = B, node = arcY.node)))
    }else{
      Partial (coreduce = U, subproblem = (arcY = arc(
          coreduce = A, node = arcX.node), arcX = arc(
          coreduce = B, node = arcY.node)))
    }
  }
}
```

8 Conclusion

In this lecture, we introduced a formal definition of BDDs and explained how to manipulate it. As mentioned in introduction these manipulations can be used to solve any Boolean function related problem especially the problem of satisfying a quantified Boolean formula which is known to PSPACE-Complete.

Then, we explained how to use complemented arcs in order to slightly reduce the memory usage, while increasing the complexity of the structure. However, this compression scheme is very interesting because it allow to compute the NOT operator in constant time without accessing to the unique table.

Then, we introduced a new formalism in order to more strictly separate memory accesses and syntactical canonicity (called the Canonical Shared Forest) from node computation and semantic canonicity (called Reduced Decision Tree).

References

[1] Karl S. Brace, Richard L. Rudell, and Randal E. Bryant. Efficient implementation of a BDD package. In *Proceedings of the 27th ACM/IEEE Design Automation Conference*, DAC '90, pages 40–45, New York, NY, USA, 1990. ACM.

[2] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Comput.*, 35(8):677–691, August 1986.

[3] F. Corno, M.S. Reorda, and G. Squillero. Rt-level itc'99 benchmarks and first atpg results. *Design Test of Computers, IEEE*, 17(3):44–53, Jul 2000.

[4] Justin E. Harlow, III, and Franc Brglez. Design of experiments and evaluation of bdd ordering heuristics, 2001.

[5] Holger H. Hoos and Thomas Stützle. Satlib: An online resource for research on sat. pages 283–292. IOS Press, 2000.

[6] D. Jankovic, R. S. Stankovic, and R. Drechsler. Decision diagram optimization using copy properties. In *Proceedings Euromicro Symposium on Digital System Design. Architectures, Methods and Tools*, pages 236–243, 2002.

[7] D. M. Miller and R. Drechsler. Dual edge operations in reduced ordered binary decision diagrams. In *Circuits and Systems, 1998. ISCAS '98. Proceedings of the 1998 IEEE International Symposium on*, volume 6, pages 159–162 vol.6, May 1998.

[8] D. M. Miller and R. Drechsler. Implementing a multiple-valued decision diagram package. In *Proceedings. 1998 28th IEEE International Symposium on Multiple- Valued Logic (Cat. No.98CB36138)*, pages 52–57, May 1998.

[9] D. M. Miller and R. Drechsler. On the construction of multiple-valued decision diagrams. In *Proceedings 32nd IEEE International Symposium on Multiple-Valued Logic*, pages 245–253, 2002.

[10] S. Minato, N. Ishiura, and S. Yajima. Shared binary decision diagram with attributed edges for efficient boolean function manipulation. In *27th ACM/IEEE Design Automation Conference*, pages 52–57, Jun 1990.

[11] Alan Mishchenko. An introduction to zero-suppressed binary decision diagrams. Technical report, in 'Proceedings of the 12th Symposium on the Integration of Symbolic Computation and Mechanized Reasoning, 2001.

[12] Fabio Somenzi. Binary decision diagrams. 1999.

[13] S. Yang. Logic synthesis and optimization benchmarks user guide: Version 3.0. Technical report, 1991.