BDD Compression

Joan Thibault

May 22, 2017

Abstract

Reduced Ordered Binary Decision Diagram ((RO)BDD) [**?**, **?**] 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 exists such as Zero-suppressed Decision Diagram (ZDD) [**?**], Multi-valued Decision Diagram (MDD) [**?**, **?**] and variations of the reduction rules such as "output inverter" [**?**], "input negation" [**?**], "shifting variables" [**?**], "dual edges" [**?**] or "copy node" [**?**].

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 to extract 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 node and tends to reduce the overall memory cost. However, several drawback arise: no in-place sifting (permutation of adjacent variables), bigger nodes of variable size and it slightly complexify the manipulation.

We implemented both the "extract U" and the "output negation" variants in an OCaml program and tested it against several benchmarks [**?**, **?**, **?**]. 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

Checking if two logic circuits (i.e. a Directed Acyclic Graph which nodes represent binary operators) are equivalent (i.e. implement the same Boolean function) is known to be a NP-complete problem. This problem arises when checking that a circuit has been properly optimized (i.e. the circuit is better according to some metric, but the function is unchanged). In order to solve this problem, a possible approach is to translate it into a CNF formula (performed in linear time in the number of nodes in both circuits by introduction new variables) and then use a SAT-solver. Alternatively, one can compile both circuits into ROBDDs (a structure which has been proven by Bryant et al. to be canonical in 1986) then check that the two representative are identical. If the structure was build using hash-consing, the identity test can be performed in constant time, however the compilation might cost exponential time. A (RO)BDD (Reduced Ordered Decision Diagram) is a Directed Acyclic Graph, is similar to Shannon's Decision Tree, except that (1) identical sub-trees are merged and (2) nodes with both edges leading to the same node are removed. BDDs can be used to efficiently solve the Circuit SATisfiability problem (CSAT), the Quantified SAT (QSAT), and counting problems such as #SAT, #CSAT, #SAT, #CSAT. Thus, BDDs have many applications: Bounded Model Checking, Planning, Software Verification, Automatic Test Pattern Generation, Combinational Equivalence Checking, Combinatorial Interaction Testing, etc.

An historical reason to efficiently solve CSAT is to check whether two logic circuits are equivalent. Thus, allowing to certify that optimization performed on an electronic design are correct and did not introduced any error. An instance of CSAT can easily be converted into an instance of SAT by introducing an additional variable for each wire in the circuit. Thus, we can use regular SAT-solvers to solve CSAT. An other, possible a

and many others.. Finite Algebrae are widely used in computer science. In various fields such as computer aided design, artificial intelligence, combinatorics. In these fields, many problems can be formulated as a sequence of operations over finite set functions. Hence, efficiently manipulating them allow to solve problems that would stay intractable with any naive approach without the need to design (i.e. implement, prove and optimize) specific algorithms for each new problem. A particular case is made of Boolean Functions which can be used to represent any other functions over finite sets. Finding a true (or false) assignment to a function represented by a formula/circuit (or constraints) is become feasible even for large instances (more than thousands of variables) with breakthrough in SAT-solvers and constraint-solvers. However, other problems such has circuit optimization, counting problems, or solving quantified formulas are still pretty intractable (around hundreds of variables). This lecture will focus on one of the available solutions : manipulation of Reduced Ordered Binary Decision Diagrams also called (RO)BDDs.

The interest of BDDs is due to their canonicity witch allow to compute the equivalence between Boolean functions in constant time (instead of exponential time usually). An other interest is that for each binary operators you apply $(\wedge, \vee, +)$ on two BDDs of respective size n and m then the resulting graph has size smaller than $n \times m$. A similar result is available for quantifiers $\exists$ and $\forall$ which has an output of size at most quadratic in its input's size. Computing $\neg$ can be performed in linear time and space when the resulting BDD has exactly the

same size than the original one. These results (that we won't demonstrate) that the structure wont have an unpredictable size blow up when applying operators (which might happen for other representation such has the Sum Of Product normal form).

This lecture will be organized as follow : first we will introduce and define BDDs, then we will focus on how to compress them using additional reduction rules. We will make a first attempt to formalize these reduction rules. After what, we will use this formalism to introduce a new compression scheme in order to detect and remove useless variables.

2 Preliminaries

Binary Decision Diagrams represent Boolean functions. In this section we introduce the notation we use for Boolean formulas and recall the basic results that we need for manipulating BDD manipulation. We denote conjunction by $\wedge$, disjunction by $\vee$, negation by $\neg$. We denote $\longrightarrow_S$, the Shannon operator (n.b. $x \longrightarrow_S y, z = (x \wedge y) \vee (\neg x \wedge z)$).

Restrictions

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.

For each Boolean function f, g of arity n, we denote $f \star_i g = (x_0, x_1, ..., x_n) \rightarrow x_0 \longrightarrow_S f(x_1, ..., x_n), g(x_1, ..., x_n)$. Nb : $(f \star_i g)[i \leftarrow 1] = f$ and $(f \star_i g)[i \leftarrow 0] = g$.

Expansion Theorem (for restrictions)

Let f be a function of arity n, then $f = f[i \leftarrow 1] \star_i f[i \leftarrow 0]$

Cofactors

For every Boolean function f of arity n, we denote $f[x_i = b] = (x_1, ..., x_n) \rightarrow f(x_1, .., x_{i-1}, b, x_{i+1}, ..., x_n)$ the function of arity n called the i-th positive cofactor of f if $b = 1$, the negative one otherwise (n.b. $f[x_i = b]$ does not depend on x_i).

For every Boolean function $f[i \leftarrow 0] \neq f[i \leftarrow 1]$ then f depends on its i-th variable. We define the *support* of f the set of variable on which f depends.

Cofactors have the following properties:

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

Nb : restrictions have similar properties

Expansion Theorem (for cofactors)

Let f be a function of arity n, then $f = x_i \longrightarrow_S f[x_i = 1], f[x_i = 0]$

3 Binary Decision Diagram (BDD) and Canonicity

Definition of BDD

A Binary Decision Diagram is a directed acyclic graph $(V \cup \mathbb{B}, \Psi \cup E)$ representing a vector of Boolean functions $F = (f_1, ..., f_k)$. The nodes are partitioned into two sets : V is the set of internal nodes. Each node $v \in V$ has an *var* identifying a variable, and two outgoing arcs (in E): *then* and *else*. $\mathbb{B}$ are terminal nodes. The arcs are partitioned into two sets : E is the set of arcs which has both a *source* and a *node* (or *destination*) field and Ψ the set of arcs (which has a cardinality of k) that has only a *node* (or *destination*) field (no *source* field).

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

- $\phi(0) = 0$, $\phi(1) = 1$
- $\psi(arc) = \phi(arc.node)$
- $\phi(node) = node.var \longrightarrow_S \psi(node.then), \psi(node.else)$

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

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)

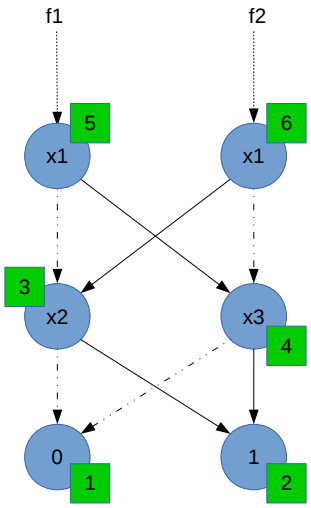

Figure 1: For each internal node, the outgoing full (respectively dashed) arrows represent the `then` (respectively `else`) arc which is followed when the variable is true (respectively false). Each numbered green square represent the identifier affected to each node during the construction process

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 dia-

gram. 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.

```
1  CONS(var, then, else){
2    if(then.node = else.node){
3      return then
4    }else if (mynode = node(var, then, else) in unique
         table){
5      return mynode's identifier
6    }else{
7      id = new identifier
8      store (mynode, id) in unique table
9      return arc(node = id)
10   }
11 }
```

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.

```
1  AND(f, g){
2    if f > g{
3      return AND(g, f)
4    }else if(terminal case){
5      return terminal((f, g))
6    }else if (memoization table has entry (f, g) {
7      return memoization((f, g))
8    }else{
9      let x be the top variable of {f, g}
10     f1, f0 = cofactor(x, f)
11     //if f does not depends on x, then f1=f0=f
12     g1, g0 = cofactor(x, g)
13     //idem for g
14     //but either f or g has to depend on x
15     r1 = AND(f1, g1)
16     r0 = AND(f0, g0)
17     r = CONS(x, r1, r0)
18     insert {key = (f, g); value = r} in memoization
19     return r
20     }
21 }
```

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 $\neg f$.

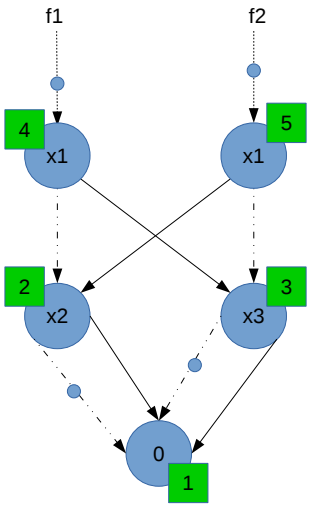

Figure 2: Same BDD as in Figure 1, when introducing complemented arcs (represented by an with a small circle) and maintaining the reduction rule that then arcss (represented by full arrows) can't be complemented arcs.

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.

```
1 RDT.CONS(arc1, arc0){
2   match RDT-consensus(then, else, cmp) with
3     | Solved result ⟶ result
4     | Partial (coreduce, reduce) ⟶
5       arc(coreduce, CSF.PUSH(reduce))
6 }
```

`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))
}
```

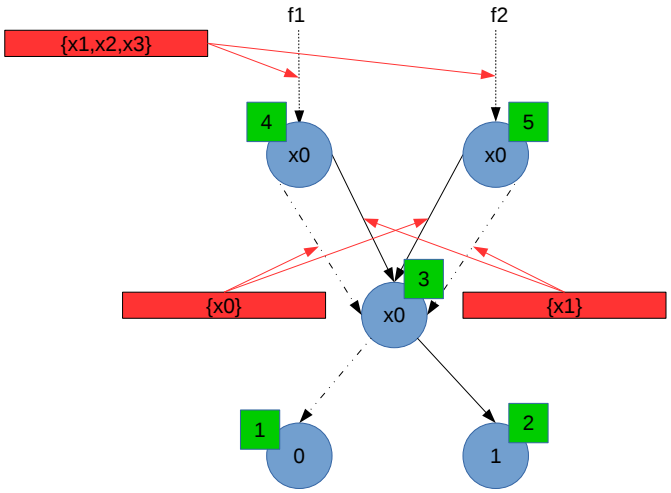

Figure 3: Same BDD as in Figure 1, when detecting and removing useless variables. Each red rectangle contain the support set of each arc's function.

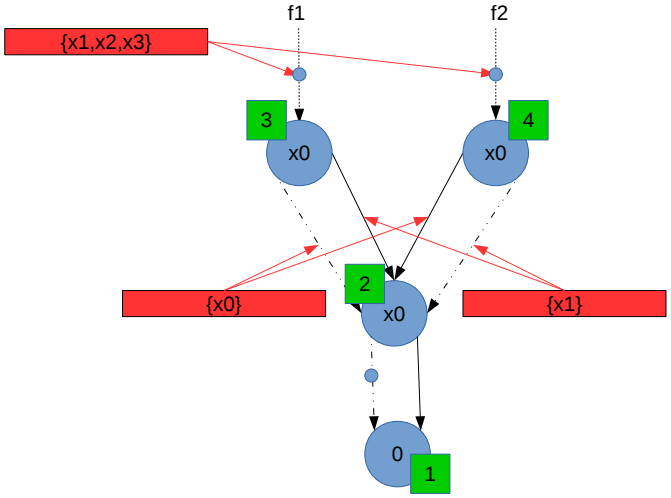

Figure 4: Same BDD as in Figure 2 and 3, with both additional reduction rules : complemented arcs and extraction of useless variables.

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:

```
1 RDT-consensus(then, else, cmp){
2   if (cmp = 0) & then.coreduce = else.coreduce then{
3     we denote {i_1 < ... < i_k} = then.coreduce
4     Solved arc(then = {i_1 + 1 < ... < i_k + 1}, node = then.node)
5   }else{
6     we denote U = {i_1 < ... < i_k} = then.coreduce ∪ else.coreduce
7     we denote A = {a_1 < ... < a_l} the indexes of then.coreduce in
          U.
8     we denote B = {b_1 < ... < b_m} the indexes of else.coreduce in
          U.
9     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).