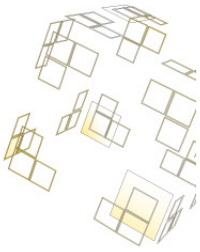
UNIVERSITEIT TWENTE.

INTERNSHIP REPORT

Constant Reduced Decision Diagram in Meddly



Author:
Maryam HAJIGHASEMI

Supervisor:
Prof. Gianfranco CIARDO
Prof. Andrew MINER
Prof. Jaco VAN DE POL

November 2014

Contents

List of Figures

Prologue

Binary Decision Diagrams (BDDs) are used in symbolic model checking to represent set of states. Variations of decision diagrams have been introduced as extensions of BDDs. The first extension is removing the constraint of having only binary values, such as Multi-way Decision Diagrams (MDDs) that can have variables with different domains. It is also possible to extend the range of the encoded function from boolean to integer or real ranges. Both multi-terminal and edge-valued decision diagrams extend this range.

Meddly is a C++ Library that supports both multi-terminal and edge-valued MDDs, and different reduction rules [2]. It is an ongoing project at Iowa State University (ISU), lead by Prof. Ciardo and Prof. Miner. Currently it supports *quasi*, *fully*, and *identity*-reduction rules. In my internship, I implemented two new reduction rules, *constant* and *CIdentity* reduction rules and some basic operations that are useful for model checking.

The concepts that were used during implementation are introduced in Chapter 1. In Chapter 2, the implemented reduction rules and algorithms of some operations are described. This chapter is specially written to guide other researchers to change/add a reduction rule in Meddly. Finally, possible conclusion and future works are presented.

Chapter 1

Decision Diagrams

In this chapter different types of Decision Diagrams (DDs) are discussed. First domain extended version of Binary DDs (Multi-way DDs) is defined in section 1.1. Then a dimensional extension is introduced, which is used for transition relations (2L-level MDDs). A terminal range extended version of MDD called Multi-terminal MDD, is also described in section 1.2. In section 1.3, Edge-Valued MDDs are explained with an associated value for each edge.

In the following, $\mathbb{B}$ is denoted by the set $\{0, 1\}$ of boolean values, and $\mathbb{N}, \mathbb{Z}, \mathbb{R}$ represent the natural, integer and real numbers, respectively.

1.1 Multi-way Decision Diagrams (MDDs)

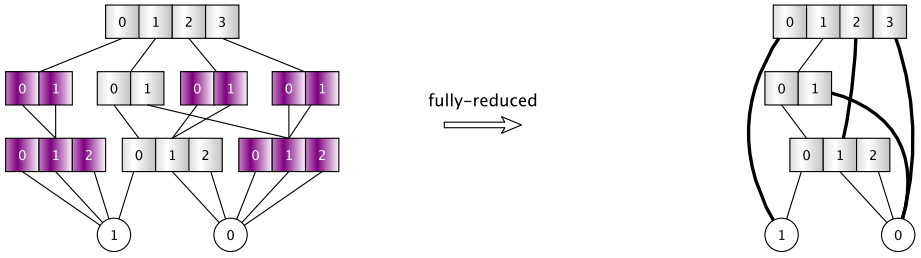
Multi-way DD [4] is an extension of Binary DD [1][3], since it allows different domains for variables in a decision diagram. Assume variables $x_1, \cdots, x_L$ with an order $x_1 \prec \cdots \prec x_L$, and the domain $\widehat{\mathcal{X}} = \mathcal{X}_L \times \cdots \times \mathcal{X}_1$, then MDDs encode functions of the form $\widehat{\mathcal{X}} \to \mathbb{B}$.

For each x_k, where $1 \leqslant k \leqslant L$, the domain $\mathcal{X}_k$ can be one of the following:

- $\mathcal{X}_k = \{0, 1, \cdots, n_k - 1\}$ for some $n_k \in \mathbb{N}$
- $\mathcal{X}_k = \mathbb{N}$
- $\mathcal{X}_k = \mathbb{Z}$

According to this definition, a boolean variable can be defined by specifying $\mathcal{X}_k = \{0, 1\}$ ($n_k = 2$). Moreover, any other discrete set can also be mapped to natural numbers.

Assuming an arbitrary domain $\widehat{\mathcal{X}}$, and a value $\tau \in \{0, 1\}$ called *transparent* value, an MDD is an *acyclic directed edge-labeled graph* where:



(A) (B)

FIGURE 1.1: *quasi*-reduced vs *fully*-reduced MDD ($\tau = 0$)

- 0 and 1 are the only *terminal* nodes (nodes without any outgoing edges), and are at level 0. A *terminal* node with value τ is called *transparent* node.
- Each *nonterminal* node p is at a level k, $L \geq k \geq 1$, which is denoted as $p.lvl = k$.
- A *nonterminal* node p at level k has exactly n_k outgoing edges, each of which labeled with a different i_k, $i_k \in \mathcal{X}_k$, and pointing to a node q; where $p.lvl > q.lvl$. It can be written as: $p[i_k] = q$. If an edge points to the *transparent* terminal node τ, it is called a *transparent* edge, otherwise it is called *opaque*.
- A decision diagram is *canonical*, if there is a unique representation for a given function that is encoded by a given class of decision diagrams. The following properties should be satisfied for a canonical representation:
  - There should be no*duplicate* nodes, which means for each nodes p and q at level $k > 0$, if $p \neq q$, then there should be $p[i_k] \neq q[i_k]$ for some $i_k \in \mathcal{X}_k$.
  - It should be reduced according to one reduction rule, as explained below. For example, there should be no redundant nodes in *fully*-reduced, or a constant node in *c*-reduced.
- It is also assumed that there is no *transparent* non-terminal nodes in DDs. In other words, for each nodes p at level $k > 0$, there should be $p[i_k] \neq \tau$ for some $i_k \in \mathcal{X}_k$.

Reduction rules Decision diagrams can be reduced in size and some edges can skip levels according to a defined reduction rule. Each reduction rule defines an interpretation for skipping levels in a way that the reduced diagram is canonical. Some reduction rules are based on *transparent* edge definition. In all cases, it is not allowed to have two nodes in the same level and the same edges pattern. In Meddly, following are supported reduction rules $\rho(k)$ for level k, $1 \leq k \leq L$:

- **Quasi-reduced** ($\rho(k) = Q$): if the MDD be *quasi*-reduced at level k, there is no level skipping (only transparent edges can skip over level k).

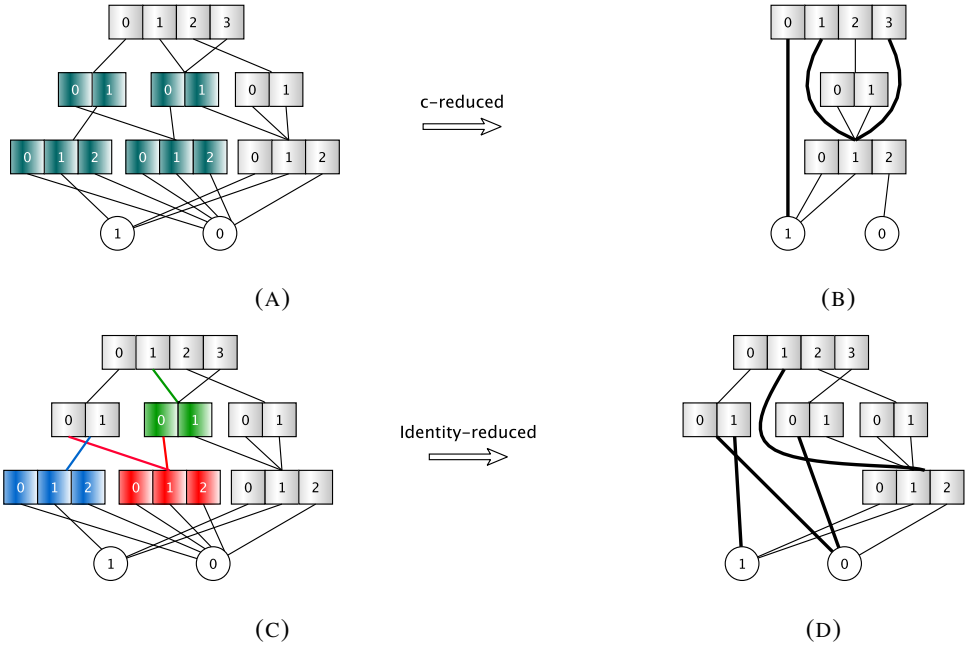


FIGURE 1.2: *quasi*-reduced vs *c*-reduced and *identity*-reduced MDD ($\tau = 0, c = 1$)

- **Fully-reduced** ($\rho(k) = F$) : The *fully*-reduction rule does not allow any redundant node, which is a node p that for all $i_k \in \mathcal{X}_k$, $p[i_k]$ point to a particular node.

  In Figure 1.1 the *transparent* value is considered zero ($\tau = 0$). The left diagram is an example of *quasi*-reduced MDD. The colored nodes on it are redundant nodes. These nodes are reduced on the *fully*-reduced MDD on the right. The tick lines show edges that skip some levels.

- **c-reduced** ($\rho(k) = c$) : The c-reduction rule skips node p for some constant value $c \in \mathcal{X}_k$, if for all $i_k \neq c$, $p[i_k] = \tau$. This means, in a c-reduced MDD there is no node that all of its edges are transparent except the outgoing edge for value c. This rule generalizes the *zero-suppressed* reduction rule, where c is always 0 [6].

  An example of this reduction rule is represented in Figure 1.2, where the *transparent* value is assumed to be zero ($\tau = 0$) and the *constant* value to be one ($c = 1$). The green nodes are reduced in diagram (B), using the 1-reduction rule, since for all edges $i_k \neq 1$, which point to zero terminal node or to a node that is also skipped with 1-reduction rule.

- **Identity-reduced** ($\rho(k) = I$) : The *identity*-reduction rule [7] forbids reaching node q at level k with only one outgoing edge $q[i] \neq \tau$, that is pointed by $p[i]$, where p is a node at level $k + 1$. Figure 1.2d, is an example of *identity*-reduced MDD. All colored nodes in diagram (C) have the defined attributes. The blue and green nodes are pointed by 1-edge of their upper level and all edges except 1 lead to the zero *terminal* node. The blue node will be reduced as the result, but the green node is not omitted, since it is also pointed by 3-edge of its upper level. The red node is pointed by two 0-edges, and all of its edges

(A) (B)

FIGURE 1.3: *full* vs *sparse* representation of Figure 1.1a

point to 0. So it satisfies the described properties of *identity*-reduction rule and will be reduced.

Each decision diagram can be stored using two representations, *full* or *sparse* [4]. These two representations are given in Figure 1.3, for the *quasi*-reduced example of Figure 1.1a . In the *full* representation, Figure 1.3a, a node is skipped if all of its edges point to the *transparent* node. In the *sparse* representation given in Figure 1.3b, parts of those nodes that are leading to the nodes pointing to the *transparent* node are omitted. These parts are colored in Figure 1.3a. Therefore, this representation is more compact compared to the previous one.

An MDD node p at level k encodes the function $f_p(i_1, \cdots, i_k) \in \mathbb{B}$, where $i_1, \cdots, i_k$ are an evaluation for variables $x_1, \cdots, x_k$, defined recursively by

$$f_p(i_1, \cdots, i_k) = \begin{cases} p & \text{if } k = 0, \text{ i.e., } p \in \{0, 1\} \\ g_{i_k, p[i_k]}(i_1, \cdots, i_{k-1}) & \text{if } k > 0, \end{cases}$$

where given MDD node q at level $h, l \geq h$, and $i \in \mathbb{Z}$,we let

$$g_{i,q}(i_1, \cdots, i_l) = \begin{cases} f_q(i_1, \cdots, i_l) & \text{if } l = h \\ g_{i_l, q}(i_1 \cdots, i_{l-1}) & \text{if } q \neq \tau \wedge l > h \wedge (\rho(l) = F \vee (\rho(l) = I \wedge i = i_l) \vee \rho(l) = i_l) \\ \tau & \text{otherwise} \end{cases}$$

In the above definition, f returns a terminal value in case node p is a *terminal* node, and otherwise calls the recursive function g. If the input node q be at the same level as highest available level in the given MDD (i.e., which means no level is skipped), function f will be called to check terminal cases. However, if a level is skipped with one of the reduction rules except the *quasi*, the recursive function g is called for the same node until reaching level h where q is in. If any of these two conditions do not apply to node q, it means this node is a *transparent* node with value τ .

$$
\begin{aligned}
(x_2, x_1) &\to (x_2', x_1') \\
(0,0) &\to (2,1) \\
(0,1) &\to (2,0) \\
(1,0) &\to (1,0) \\
(1,0) &\to (1,1) \\
(2,0) &\to (0,0) \\
(2,1) &\to (1,0)
\end{aligned}
$$

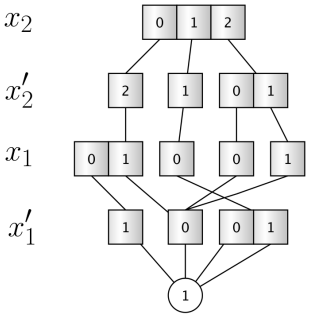


FIGURE 1.4: *sparse* representation of $2L$-level MDD

Symbolic encoding of transition relations: Sets of states can be represented with MDDs, as a function $f : \widehat{\mathcal{X}} \to \mathbb{B}$. Transition relations can be represented by functions of the form $\widehat{\mathcal{X}} \times \widehat{\mathcal{X}} \to \mathbb{B}$. If L is the number of variables in domain $\widehat{\mathcal{X}}$, and total number of variables in an MTMDD is V, then a transition relation needs two sets of L variables ($V = 2L$), which are *unprimed* $\{x_1, \cdots, x_L\}$ to refer to "from" states and *primed* $\{x_1', \cdots, x_L'\}$, which refer to "to" states. This $2L$-level MDD can be interleaved, where the ordered variables are $x_L \succ x_L' \succ \cdots \succ x_1 \succ x_1'$. In Figure 1.4, an example of interleaved $2L$-level MDDs is represented, where $\widehat{\mathcal{X}} = x_2 \times x_1 = \{0,1,2\} \times \{0,1\}$, $x_2 \succ x_2' \succ x_1 \succ x_1'$, the transitions are as defined on the left.

1.2 Multi-Terminal MDDs (MTMDDs)

MTMDDs allow arbitrary range $\mathcal{X}_0$ that represent possible values for functions which are encoded by a decision diagram (*terminal* nodes). It can be $\mathbb{B}, \mathbb{N}, \mathbb{Z}, \mathbb{R}^{\geq 0}$, and $\mathbb{R}$. Thus it will encode functions of the form $\widehat{\mathcal{X}} \to \mathbb{R}$. The above definition of MDD also holds for MTMDD, by replacing all Boolean sets for *terminal* nodes with range $\mathcal{X}_0$.

An example of MTMDDs is represented in Figure 1.5 where $\tau = 0$ and $c = 1$. The left diagrams are *quasi*-reduced and the purple nodes are redundant and are skipped in *fully*-reduced diagram (A). The green nodes are the ones that all edges $i_k \neq 1$ points to zero. These nodes are skipped in the *c*-reduced diagram (B). In the third row, blue edges pointing to blue nodes are identity edges, which are reduced in the *identity*-reduced diagram (C).

1.3 Edge-Valued MDDs (EVMDDs)

EVMDDs represent functions with a non-Boolean range, but these values are not found as *terminal* nodes [4] [5]. Instead, there is a single *terminal* node Ω with no value, and an integer value is assigned to each edge of the diagram. Thus each value is distributed over the edges

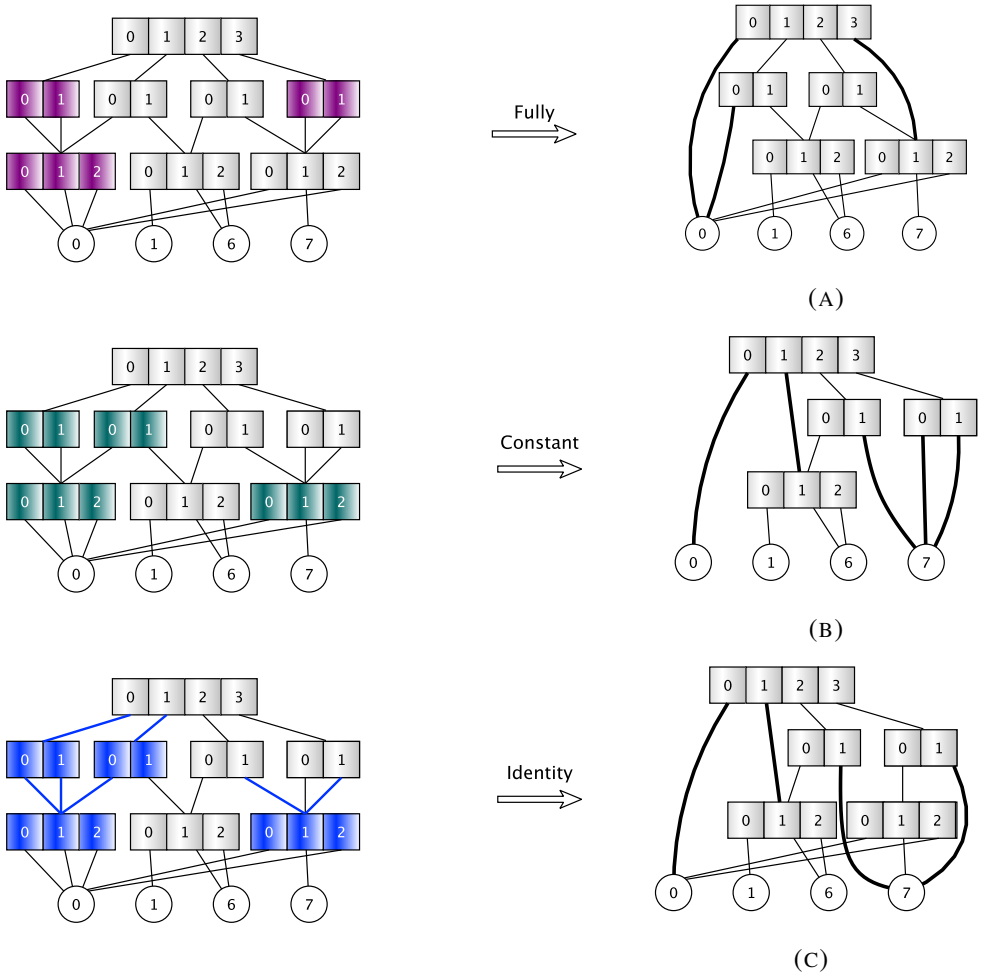
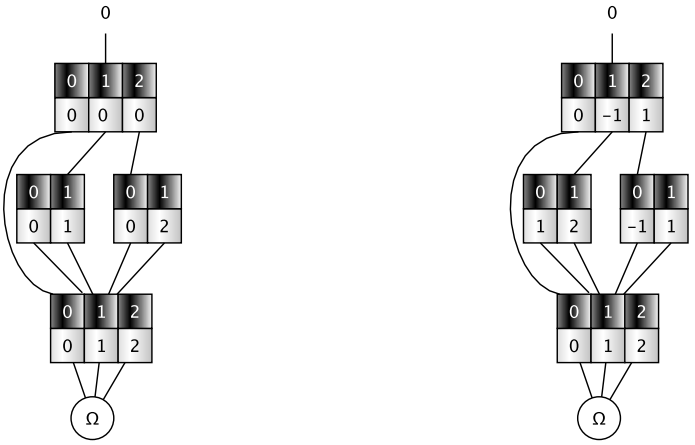


FIGURE 1.5: *quasi, fully, constant* and *identity*-reduced MTMDD ($c = 1, \tau = 0$)

along a path, and the value of the function is obtained by adding the values along the path to Ω. In other words, EVMDD encodes the value of terminal nodes over the edges. EVMDDs encode functions of the form $\widehat{\mathcal{X}} \rightarrow \mathbb{Z}$.

For canonicity, nodes are normalized in a way that 0-edge (outgoing edge from value 0) is assigned to 0 value. The root node has a "dangling arc" that will be added to the path value for function evaluation. Figure 1.6 illustrates an example of EVMDD. Both diagrams in this Figure represent function $f : (x_1, x_2, x_3) \rightarrow (x_1.x_2) + x_3$, where diagram (A) is a canonical representation. However, diagram (B) is not canonical, since it includes nodes that the 0-edge is assigned to a non-zero value.

The **positive Edge Value MDDs (EV$^+$MDDs)** are the same as EVMMDs but with different normalization rules. All outgoing edge values of a node should be non-negative or ∞^+, and at least one of them be 0. The dangling arc then becomes the minimum value of the encoding function. Assuming an arbitrary domain $\widehat{\mathcal{X}}$, a range $\mathcal{X}_0$, an EV$^+$MDD is an acyclic directed edge-labeled graph where:

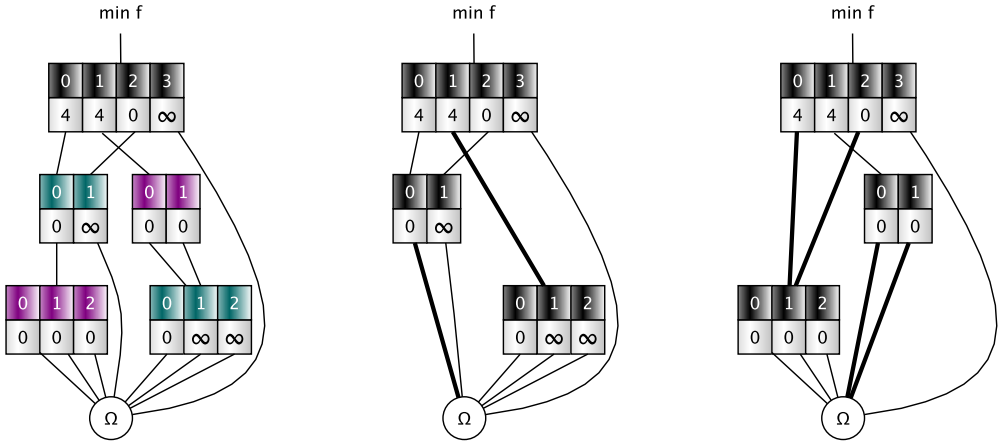


(A) Canonical (B) non-canonical

FIGURE 1.6: EVMDD representation of function $f : \{0, 1, 2\} \times \{0, 1\} \times \{0, 1, 2\} \to \mathbb{Z}$, $(x_1, x_2, x_3) \to (x_1.x_2) + x_3$

- Ω is the only *terminal* node, and is at level 0. It is also considered as a *transparent* node.
- Each *nonterminal* node p is at a level k, $L \geq k \geq 1$($p.lvl = k$).
- A *nonterminal* node p at level k has exactly n_k outgoing edges. Each of them labeled with a different i_k, $i_k \in \mathcal{X}_k$, with value $v \in \mathcal{X}_0$ ($p[i_k].val = v$), and pointing to a node q ($p[i_k].ch = q$), where $p.lvl > q.lvl$. It can be written as: $p[i_k] = \langle v, q \rangle$. If an edge points to the *transparent* terminal node Ω and has value ∞^+, it is called *transparent* and otherwise *opaque* edge.
- For a canonical representation, the following properties should hold:
  - There should not be any *transparent* non-terminal nodes. In other words, for each node p at level $k > 0$, there should be $p[i_k] \neq \langle \infty^+, \Omega \rangle$ for some $i_k \in \mathcal{X}_k$.
  - No duplicate node is allowed. Given nodes p and q at level $k > 0$, if $p \neq q$, then there should be $p[i_k] \neq q[i_k]$ for some $i_k \in \mathcal{X}_k$.
  - All edges with ∞^+ value should point to Ω (if $p[i_k] = \langle \infty^+, q \rangle$, then $q = \Omega$).
  - As mentioned earlier, all values should be normalized in a way that the minimum value of the leaving edges of each node be zero ($min\{p[i_k].val : i_k \in \mathcal{X}_k\} = 0$).

Figure 1.7, represents *quasi*-reduced, *fully*-reduced and *c*-reduced EV$^+$MDD. A redundant node in EV$^+$MDD is a node where all edges point to the same node, and have the same value. Based on the normalization rule, there should be an edge with value 0. Therefore all edges of a redundant nodes should be assigned to 0. In diagram (A), which is *quasi*-reduced, the two purple nodes are redundant according to this definition. As can be seen, these nodes are skipped in diagram (B) which illustrates the *fully*-reduced of the same function.



(A) (B) (C)

FIGURE 1.7: *quasi*-reduced, *fully*-reduced and c-reduced EV$^+$MDDs ($c = 0$)

A node will be reduced in c-reduced diagram, if for all edges $i_k \neq c$, $p[i_k] = \langle \infty^+, \Omega \rangle$. The green nodes in diagram (A) have this attribute and are skipped on diagram (C), which is *0*-reduced ($c = 0$). As it is shown in all diagrams, edges with value ∞^+ skip all lower levels.

In EV$^+$MDD, node p at level k will be removed as a *identity* reduced node, if it is pointed by an edge from index j of node q that is denoted as $q[j].ch = p$, and for all $i_k \in \mathcal{X}_k \setminus \{j\}$, $p[i_k] = \langle \infty^+, \Omega \rangle$ and $q.lvl = p.lvl + 1$.

An EV$^+$MDD node p at level k encodes the function $f_p(i_1, \cdots, i_k) \in \mathbb{Z}$, where $i_1, \cdots, i_k$ are evaluations for variables $x_1, \cdots, x_k$ and defined recursively by

$$f_p(i_1, \cdots, i_k) = \begin{cases} 0 & \text{if } k = 0, \text{ i.e., } p = \Omega \\ p[i_k].val + g_{i_k, p[i_k].ch}(i_1, \cdots, i_{k-1}) & \text{if } k > 0 \text{ and } p[i_k].val \neq \infty^+ \\ \infty^+ & \text{otherwise} \end{cases}$$

where given EV$^+$MDD node q at level $h, l \geq h$, and $i \in \mathbb{Z}$,we let

$$g_{i,q}(i_1, \cdots, i_l) = \begin{cases} f_q(i_1, \cdots, i_l) & \text{if } l = h \\ g_{i_l,q}(i_1 \cdots, i_{l-1}) & \text{if } l > h \wedge (\rho(l) = F \vee (\rho(l) = I \wedge i = i_l) \vee \rho(l) = i_l) \\ \infty^+ & \text{otherwise} \end{cases}$$

The difference of this definition with MDD is that the terminal value is calculated recursively by adding the value of edges, instead of using *terminal* nodes. Note that the *terminal* node Ω has the value 0.

EV*MDDs are the multiplicative version of EV$^+$MDDs. The *transparent* edges are the ones with value 0 and point to Ω. The normalization rule is also different, i.e., values should be

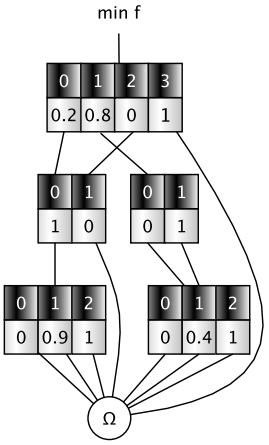


FIGURE 1.8: EV*MDD example

normalized so that $max\{p[i_k].val : i_k \in \mathcal{X}_k\} = 1$. Figure 1.8 represents an example of EV*MDD.

An EV*MDD node p at level k encodes the function $f_p(i_1, \cdots, i_k)$, where $i_1, \cdots, i_k$ are evaluations for variables $x_1, \cdots, x_k$, defined recursively by

$$f_p(i_1, \cdots, i_k) = \begin{cases} 1 & \text{if } k = 0, \text{ i.e., } p = \Omega \\ p[i_k].val \cdot g_{i_k, p[i_k].ch}(i_1, \cdots, i_{k-1}) & \text{if } k > 0 \end{cases}$$

where given EV*MDD node q at level $h, l \geq h$, and $i \in \mathbb{Z}$,we let

$$g_{i,q}(i_1, \cdots, i_l) = \begin{cases} f_q(i_1, \cdots, i_l) & \text{if } l = h \\ g_{i_l, q}(i_1 \cdots, i_{l-1}) & \text{if } l > h \wedge (\rho(l) = F \vee \rho(l) = I \wedge i = i_l \vee \rho(l) = i_l) \\ 0 & \text{otherwise} \end{cases}$$

Chapter 2

Meddly

Multi-terminal and Edge-valued Decision Diagram LibrarY (Meddly) is an open-source C/C++ library, that support MTMDDs and EVMDDs as its name also suggests[2]. In this chapter, first the Meddly terminology and available reduction rules and operations on different decision diagrams are introduced. Then in Section 2.2 the new implemented reduction rules and available operations for them are discussed. Finally, the verification of implemented code using automatic testing, is represented in Section 2.3.

2.1 Structure of Meddly

As described before, decision diagrams are directed acyclic graphs. In Meddly, DDs can be used to represent a function with a finite number (K) of variables, and each variable x_k can have finite values $0 \cdots n_k$. In order to have a canonical representation no *duplicate* nodes can appear in DDs and a specific ordering is also required.

Types of DDs :A set of DDs with specific type of DD, an associated domain, and named nodes is called a *forest*. The following forms of DDs, with different reduction rules and ranges are available in Meddly [2]

- MDD : as described in section 1.1, which is a function of form $f : \widehat{\mathcal{X}} \to \mathbb{B}$
- MxD : this term is used for $2L$-level decision diagram concept, described in Section 1.2,with encoding functions $f : \widehat{\mathcal{X}} \times \widehat{\mathcal{X}} \to \mathbb{B}$
- MTMDD : encoding functions of form $f : \widehat{\mathcal{X}} \to \mathcal{R}$, where $\mathcal{R}$ can be a subset of $\mathbb{N}$ or $\mathbb{R}$ (Section 1.2)

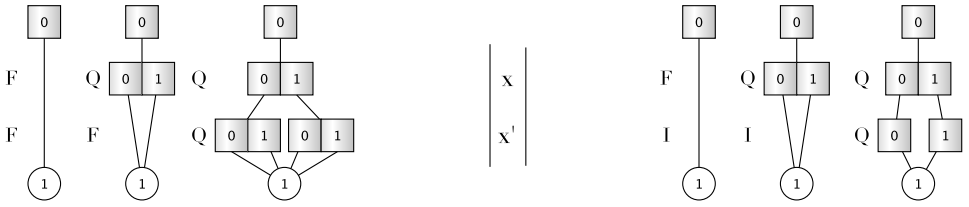
- MTMxD : based on the definitions of MTMDD and MxD, Multi-terminal MxD is also defined for multi-terminal transition relation which encodes functions of the form $f : \widehat{\mathcal{X}} \times \widehat{\mathcal{X}} \to \mathcal{R}$
- EV$^+$MDD : encoding functions of the form $f : \widehat{\mathcal{X}} \to \mathbb{N} \cup \{\infty\}$, which is explained in Section 1.3
- EV*MxD : based on the definitions of EV*MDD and MxD, EV*MxD is also defined for edge-valued transition relation which encodes functions of the form $f : \widehat{\mathcal{X}} \times \widehat{\mathcal{X}} \to \mathbb{R}^{\geq 0}$.

Available operations: Meddly supports the following operations for the mentioned forms of DDs [2]

- *Unary*: Complement and Copy (copy a DD from one type to another compatible one) explained in section 2.2.4
- *Binary:* (more explanation on section 2.2.2)
  - on Booleans: Union, Intersection, Difference.
  - on integers and reals: +, -, * , / .
- *Relational:* $=, \neq, <, \leq, >, \geq$, min , max.
- *Symbolic:* Reachable states can be calculated by Meddly using initial state and transition relation. One step reachable states are calculated by using Pre-image and Post-image. Pre-image calculates reachable states in a backward step and Post-image algorithm, which is also explained in section 2.2.3, calculates reachable states in a forward step.

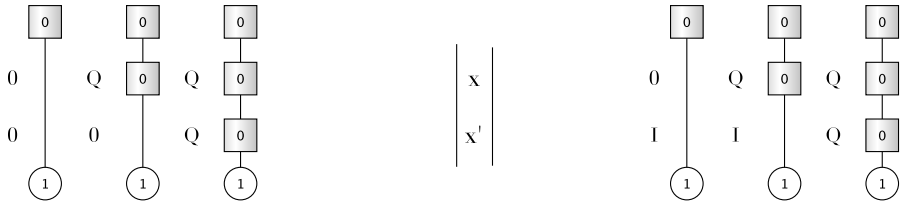
Reduction rules: Currently Meddly doesn't support different reduction rules for different levels, instead it only allows one rule for all levels of a given DD. Meddly supports *quasi* and *fully*-reductions for all types of DDs. In Meddly, *identity* reduction rule is only supported for relations, i.e., $2L$-level, with a restriction: Identity reduction rule is applied to primed levels and *fully* reduction rules is being used for unprimed levels.

Figure 2.1, shows all the reduction rules supported by Meddly. Assume binary variables x and x' of a transition relation are skipped. The equivalent *quasi* reduced diagram is calculated by replacing each skipped level according to reduction rule definition, from upper levels to the terminals. In Figure 2.1a, both levels are *fully* reduced, so if a level is skipped it means that both edges of the node point to a same node. Thus as the first step, a redundant node is added for variable x, and the reduction rule is now *quasi* for this level. The same step is done for both edges of this node that skip level x'. In Figure 2.1b , for primed and unprimed levels *identity* and *fully* reduction rules are being used, respectively. The first step is the same as described for the previous example, however for the primed level, the skipped level means x' has the same value as x and the other edge points to *transparent* node 0.



(A) (B)

FIGURE 2.1: Skipping level interpretation using existing reduction rules in Meddly (*sparse* representation)



(A) (B)

FIGURE 2.2: Skipping level interpretation using new reduction rules in Meddly (*sparse* representation)

2.2 The new implemented reduction rules

As mentioned earlier, Meddly is an ongoing project with the aim of supporting different reduction rules for a variety of DDs, depending on the requirements and features of the input data. However, it is still constrained to the reduction rules that it supports. The author task for the first step, was to implement the *constant* reduction rule for all types of available DDs, defined in Section 1.1. Figure 2.2a, represents an example of this reduction rule, where binary variables x and x' are skipped with constant value 0 ($c = 0$). Each skipped level for a diagram reduced by this rule means all edges except the edge from constant value, points to the *transparent* node. Therefore, the equivalent *quasi* reduced node for skipping variable x and x' has an outgoing edge from 0.

After adding *c*-reduction rule now we have all the possible reduction rules supported by Meddly (i.e., *quasi*, *fully*, *identity* and *c*-reduction rules), are obtained. Additionally, a combination of *constant* and *identity* rules for different types of DDs for relations, called CIdentity, was also added in Meddly. In this reduction rule, *c*-reduction and *identity*-reduction rules are used to reduce unprimed and primed levels, respectively. The interpretation of skipping levels for these reduction rules is shown in Figure 2.2b, which provides the same result as using *c*-reduction rule. The given example in Figure 2.3 shows the difference of these two reduction rules in skipping primed levels.

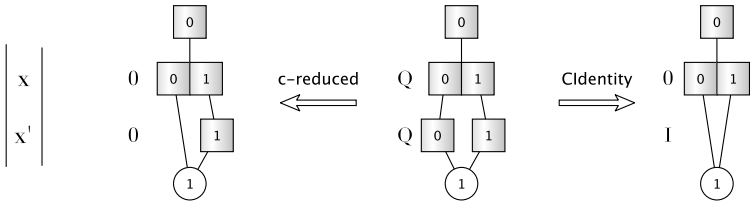


FIGURE 2.3: Comparing *constant* and *CIdentity* reduction (*sparse* representation)

Section 2.2.1, describes generation of a diagram using *constant* and *CIdentity* reduction rules. The operations over these diagrams should be supported. The three supported operations, i.e., `apply_base`, relational product and *copy* are explained in Sections 2.2.2, 2.2.3 and 2.2.4, respectively.

Note that the following sections are written with the focus to help researchers who would like to add a new reduction rule to Meddly. The challenging effort is to find where the correct places are to add the new code. Hence, in the following the operations algorithm, and how and where they should use reduction rule methods are explained .

2.2.1 Adding a new reduction rule

These steps are necessary to add a new rule in Meddly:

- Add the rule name to the enumeration "`reduction_rule` " in forest policies and get/set operations for it
- Define a function that checks, for all types of DDs that whether the input node should be reduced according to set reduction rule. In this case, the method `isConstant` checks if the node reduction rule is *Constant*, which means the diagram is *c*-reduced or it is *CIdentity*-reduced and the node is in unprimed level,so it still uses *c*-reduction rule. Then if the node has the described attribute for *c*-reduction rule, it returns $true$, otherwise $false$. The existing method `isIdenitiy`, needs some modifications to check primed levels of CIdentity as well.
- Modification of `CreateReducedNode` operation: This operation gets a node and returns the equivalent reduced node according to set reduction rule. Algorithm 1 shows how it works.
- As described in the next section, operations in Meddly create reduced nodes before calculating the result, hence, there should be a function to create node for the new defined reduction rule. This is done by functions like `initConstantReader` that create a

Algorithm 1 `CreateReducedNode` implementation

```
1:  function CreateReducedNode(in,node)        ▷ in: value of variable that is pointed by
2:      nnz, c ← 0
3:      for 1 ≤ i ≤ node.size() do
4:          if node[i]!=0 then
5:              nnz++, c ← i
6:      if nnz = 1 then
7:          if IsIdentity(node, in) then return node[in]
8:          else if IsConstant(node, c) then return node[c]
9:      if nnz = node.levelSize then
10:         if IsRedundant(node) then return node[0]
11:     if nnz = 0 then return transparentNode
```

node on the skipped level and then associate all of its children pointing to zero except the leaving edge from constant value.

2.2.2 Basic operations (Union, Intersection , ...)

All the basic operations for sets such as Union and Intersection are implemented in one base function in Meddly. In the list of Meddly's operations, there is an `apply_base` operation that can be used for some of necessary operations. This is done by overloading the `CheckTerminal` operation for each of these operations separately. The base function methodology is shown in Algorithm 2. In lines 2-3, the terminal cases are checked if the two nodes are either at the same level or not using the c-reduction rule. Since skipped levels are considered redundant nodes for checking terminal cases in `CheckTerminal` function of the implementation, it would result in a wrong answer for c-reduced diagrams. There are two solutions for this problem: First, excluding c-reduced diagrams to be checked for terminal cases, when a level is skipped. Second, implementing a different `CheckTerminal` function for c-reduced diagrams for each operation. In this case, the former solution is adopted.

After checking terminal cases the result root node level and size are calculated according to input values in lines 5-7. Then in lines 9 through 13, if each of two input diagrams skipped the node in the result level, it will be created with respect to the reduction rule being used. Then for all children of nodes at the resulting level, the operation will be called recursively from line 16 . At the end, the reduced version of the resulting node will be calculated in line 17.

In all of the operations, the resulting diagram will be cached to prevent recalculation if the same operation needs to be done for the same inputs later. Therefore there is a cache checking at the beginning of each function. This process involves searching for the available cached result. If the result was not cached before, the progress goes on and at the end the result is added to cache for later use.

Algorithm 2 `apply_base` implementation for MDDs

```
1:  function MDD_APPLY_BASE(A,B)
2:      if (!A.isConstantReduced() and !B.isConstantReduced() ) or A.level = B.level
        then
3:          if CheckTerminal(A, B, res) then return res
4:
5:      resultLevel ← Max(A.lvl, B.lvl)
6:      resultSize ← GetSize(resultLevel)
7:      result ← NodeBuilder(resultLevel, resultSize)
8:
9:      if A.lvl ≠ resultLevel then                ▷ recreate reduced node by Constant or Fully
10:         A ← A.initReducedNode(resultLevel, A)
11:
12:     if B.lvl ≠ resultLevel then                ▷ recreate reduced node by Constant or Fully
13:         B ← B.initReducedNode(resultLevel, B)
14:
15:     for 0 ≤ i < resultSize do
16:         result[i] = MDD_apply_base(A[i], B[i])
17:     return CreateReducedNode(−1, result)
```

As Algorithm 3 and 4 show, the implementation of `apply_base` operation for MxD is done in the same way as MDDs. The difference is that the primed and unprimed levels calculation are in different functions, since there are more reduction rules for primed levels. Furthermore, in Meddly a negative value is used as the primed level of an unprimed level using the same positive value. This makes definition of the next level different. In MDD, the lower level of k is $k-1$, but in MxD if $k > 0$, which means it is an unprimed level, it will be $-k$, otherwise $-k-1$.

Algorithm 3 `apply_base` implementation for unprimed levels of MxDs

```
1:  function MDD_APPLY_BASE_UNPRIMED(A,B)
2:      if CheckTerminal(A, B, res) then return res
3:
4:      resultLevel ← Max(|A.lvl|, |B.lvl|)        ▷ Primed level = - unPrimed level
5:      resultSize ← GetSize(resultLevel)
6:      result ← NodeBuilder(resultLevel, resultSize)
7:
8:      if A.lvl ≠ resultLevel then   ▷ recreate reduced node by Constant,CIdentity or Fully
9:          A ← A.initReducedNode(resultLevel, A)
10:
11:     if B.lvl ≠ resultLevel then ▷ recreate reduced node by Constant or CIdentity or Fully
12:         B ← B.initReducedNode(resultLevel, B)
13:
14:     for 0 ≤ i < resultSize do
15:         result[i] = MDD_apply_base_primed(i, −resultLevel, A[i], B[i])
16:     return CreateReducedNode(−1, result)
```

Algorithm 4 `apply_base` implementation for primed levels of MxDs

```
1:  function MDD_APPLY_BASE_PRIMED(in,level,A,B)
2:
3:      resultSize ← GetSize(level)
4:      result ← NodeBuilder(level, resultSize)
5:
6:      if A.lvl ≠ level then ▷ recreate reduced node by Constant, Fully, Identity or CIdentity
7:          A ← A.initReducedNode(level, in, A)
8:
9:      if B.lvl ≠ level then ▷ recreate reduced node by Constant , Fully, Identity or CIdentity
10:         B ← B.initReducedNode(level, in, B)
11:
12:     for 0 ≤ i < resultSize do
13:         result[i] = MDD_apply_base_unprimed(A[i], B[i])
14:     return CreateReducedNode(in, result)
```

2.2.3 Relational product

Another important operation for model checking is relational product, which calculates the set of states that are reachable in one step, using a set of states and a transition relation. Algorithm 5 illustrates how this operation is implemented. The same as `apply_base` operation, it first checks terminal cases in lines 2-4, initializes resulting level and size in lines 6-8 and then creates the reduced node in the set (if there is any), in line 11. After this step, it immediately checks $relation$ for redundant nodes in lines 12 and 13, and if it is the case, the process continues recursively. Otherwise, if an unprimed level is skipped, it will be created in line 22 and then it goes through all of its children. For each child the skipping primed level will be created in lines 27-28. After that, it calculates the result for each child and add all of them together in lines 28-35. At the end, possibility of being reduced is checked in line 36.

2.2.4 Copy

This operation copies a Diagram from one forest to another one. Here, It was used to verify the correctness of our implementation of new rules. The usage will be explained in more details in the next section. In Meddly, the `Copy` operation checks if both source and destination forests have the same reduction rules. If the reduction rules are the same, it computes the result by creating the same node in the destination forest. In algorithm 6, the other case is considered, i.e., when reduction rules are different. As before, this operation check terminal cases in line 2, which are reaching terminal node 0 or the source diagram is an empty set. In both cases the resulting diagram in any forest would be terminal node 0. The next step, which is in lines 3-5, creates the reduced node if there is any, and then recursively copies its children to the resulting

Algorithm 5 `setXrel`(relational product) implementation

```
1:  function SETXREL(S,R)
2:      if R = 0 or S = 0 then return 0                          ▷ Terminal cases
3:      if R.isTerminal() and S.isTerminal() then
4:          return ProcessTerminals(S, R)
5:                                                               ▷ result initialization
6:      resultLevel ← Max(|R.lvl|, S.lvl)
7:      resultSize ← GetSize(resultLevel)
8:      result ← NodeBuilder(resultLevel, resultSize)
9:
10:     if S.lvl ≠ resultLevel then      ▷ recreate reduced node by Constant or Fully in Set
11:         S ← S.initReducedNode(resultLevel, S)
12:     if S.lvl > |R.lvl| then                          ▷ This level is skipped in relation
13:         if !R.isConstantReduced() and !R.isCIdentityReduced() then
14:                                                      ▷ Unprimed level reduced by fully
15:             for 1 ≤ i < resultSize do
16:                 result[i] = setXrel(S[i], R)
17:     else
18:         for 1 ≤ i < resultSize do
19:             result[i] = 0
20:                     ▷ recreate reduced node by Constant, CIdentity or Fully in Relation
21:         if R.isPrimed() then
22:             R ← R.initReducedNode(resultLevel, R)
23:         for 0 ≤ iz < R.NumNode() do
24:             i ← R.index[iz]
25:             if S[i]=0 then continue
26:                         ▷ recreate reduced node by Constant, Fully, Identity or CIdentitiy
27:             if IsLevelAbove(−resultLevel, R[iz].lvl) then
28:                 R ← R.initReducedNode(resultLevel, i, R[iz])
29:             for 0 ≤ jz < R.NumNode() do
30:                 j ← R.index[jz]
31:                 newState ← setXrel(S[i], R[jz])
32:                 if result[j] = 0 then
33:                     result[j] ← newState
34:                     continue
35:                 result[j]+ = newState
36:     return CreateReducedNode(−1, result)
```

diagram from source to destination forest in lines 11 and 12. Finally the diagram will be checked for possibility of being reduced in line 13.

Algorithm 6 `Copy` implementation

```
1:  function Copy(in,k,A)
2:      if A = 0 or k = 0 then return 0                          ▷ Terminal cases
3:      if IsLevelAbove(k, A.lvl) then
4:                                  ▷ recreate reduced node by Constant, Fully, Identity or Fully
5:          A ← A.initReducedNode(k, A)
6:
7:      result ← NodeBuilder(k, A.NumNode())
8:      nextk ← nextk.isRelation ? −k : k − 1
9:
10:     for 0 ≤ i < A.NumNode() do                               ▷ copy children recursively
11:         result.index[i] ← A.index[i]
12:         result[i] ← Copy(A.index[i], nextk, A[i])
13:     return CreateReducedNode(in, result)
```

2.3 Testing

The new reduction rules and changed operations need to be tested. In Meddly there is a folder named ”test” that includes testers for several operations, using different reduction rules. For example, `Copy` operation is checked by copying more than 20 randomly generated functions, from all possible *forest* to another one, and checks if it has the same result as creating the same function in the destination forest. The new reduction rules were verified and `Copy` operation were adopted for them using the same code.

For the other operations, first different functions were generated , then an operation was chosen to calculate the result. This was done in two different forests, one with the new reduction rules and the other using previous rules which were verified before. Then, by copying the result from one forest to another, the result can be compared and checked whether the operations work correctly.

Conclusion

We introduced a couple of decision diagram reduction rules, each of which is useful for different models. We also described the supported DDs and reduction rules in Meddly and how new reduction rules can be added. This report documents the added reduction rules and help new developers of Meddly to continue its development.

For future investigations, one can compare the efficiency of new reduction rules, i.e., *Constant* and *CIdentity* , with the results for different data models using different reduction rules. Also, by assigning reduction rules to each level of DDs, it is probable that the size of DDs decreases. This can be useful depending on the nature of case-study. Finally, choosing the reduction rules can be decided by Meddly based on the given data structure as an input. This can be taken as granted to make the DD as compact as possible and reduce the burden of having to decide which rule should be used.

Bibliography

[1] S.B. Akers. Binary decision diagrams. *Computers, IEEE Transactions on*, C-27(6):509–516, June 1978.

[2] J. Babar and A. Miner. Meddly: Multi-terminal and edge-valued decision diagram library. *Quantitative Evaluation of Systems, International Conference on*, 0:195–196, 2010.

[3] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *Computers, IEEE Transactions on*, 35(8):677–691, August 1986.

[4] G. Ciardo. Data representation and efficient solution: A decision diagram approach, 2007.

[5] G. Ciardo and R. Siminiceanu. Using edge-valued decision diagrams for symbolic generation of shortest paths. In *Proc. Fourth International Conference on Formal Methods in Computer-Aided Design (FMCAD), LNCS 2517*, pages 256–273. Springer-Verlag, 2002.

[6] S. Minato. Zero-suppressed bdds and their applications. *International Journal on Software Tools for Technology Transfer*, 3(2):156–170, 2001.

[7] M Wan and G. Ciardo. Symbolic state-space generation of asynchronous systems using extensible decision diagrams. In *Proceedings of the 35th Conference on Current Trends in Theory and Practice of Computer Science*, SOFSEM '09, pages 582–594, Berlin, Heidelberg, 2009. Springer-Verlag.