社団法人　電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
TECHNICAL REPORT OF IEICE.
CAS94-126, VLD94-142, ICD94-250(1995-03)

二分決定グラフの非明示的表現法とその操作法

山内 仁†, 石浦菜岐佐††, 高橋浩光†††

†大阪府立大学工学部
数理工学科
〒593 堺市学園町 1-1
yamauchi@na.ms.osakafu-u.ac.jp

††大阪大学工学部
情報システム工学科
〒565 吹田市山田丘 2-1
ishiura@ise.eng.osaka-u.ac.jp

†††岡山県立大学情報工学部
情報通信工学科
〒719-11 総社市窪木 111
tak@c.oka-pu.ac.jp

あらまし　二分決定グラフ (Binary Decision Diagrams; 以下 BDD) は, 論理関数を計算機上で効率良く表現する方法として, 集積回路の計算機援用設計の分野で広く用いられているが, 近年の回路の大規模化により, BDD でも記憶領域が不足するような論理関数がしばしば出現することが問題となっている. これに対し, 本稿では, BDD の非明示的な表現法を提案する. これは, BDD のグラフとしての形質を論理関数によって間接的に表し, さらにこれを BDD で表す「BDD の BDD 表現」であり, 元の BDD の構造に規則性がある場合には大幅な記憶量の削減が達成される. さらに本稿では, 論理関数の二項演算を非明示的に表現された BDD 上で直接行う手法を提案する. MCNC, ISCAS85 ベンチマーク回路に対する実験では表現のサイズを平均でそれぞれ 44%, 48% に削減することができた.

キーワード　二分決定グラフ, 非明示的表現, 論理関数, 論理設計検証

Implicit Representation and Manipulation of Binary Decision Diagrams

Hitoshi Yamauchi†, Nagisa Ishiura††, Hiromitsu Takahashi†††

†Department of Mathematical Sciences, College of Engineering, University of Osaka Prefecture
Sakai, Osaka, 593 Japan
yamauchi@na.ms.osakafu-u.ac.jp

††Department of Information Systems Engineering, Faculty of Engineering, Osaka University
Suita, Osaka, 565 Japan
ishiura@ise.eng.osaka-u.ac.jp

†††Department of Communication Engineering, Faculty of Computer Science and System Engineering, Okayama Prefectural University
Souja, Okayama, 719-11 Japan
tak@c.oka-pu.ac.jp

Abstract　In this article, a new way of representing and manipulating Boolean functions is proposed, for the purpose of handling "huge" Boolean functions which are difficult to handle even by BDDs (binary decision diagrams). An implicit way of representing graphs by BDDs is applied to BDDs themselves; by giving a binary code to each node in a BDD, the graph structure of the BDD is represented via Boocan functions and then by BDDs. The regularity in the original BDD makes the resulting BDDs simple, which leads to significant reduction in memory requirement. On the other hand, there is little drawback to equivalence checking, since the canonicity of the representation is preserved. An implicit algorithms for binary operations (apply operations), reduction and recoding on implicit BDDs are also proposed. In experiments on the MCNC and the ISCAS benchmark circuits, the size of the representation is reduced to 44% and 48%, respectively, on the average.

key words　binary decision diagrams, implicit representation, Boolean function, login design verification

1 はじめに

二分決定グラフ (Binary Decision Diagrams; 以下 BDD と略す) は Akers[1] によって考案された論理関数の表現法であり, Bryant[2] によって効率的な演算手法が考案されて以来広く用いられるようになった. BDD は論理設計検証 [3], 論理合成, テスト生成 [4] などにおけるツールの効率を飛躍的に向上させたが, 応用が広がるにつれ, このBDD をもってしても表現できないような大規模な論理関数が多く現れるようになり, さらに効率の良い表現法が求められている.

これに対し, 本稿では BDD の非明示的な表現法とその操作法を提案する. BDD の非明示的 (implicit) 表現法とは, BDD のグラフとしての形質を論理関数によって間接的に表現し, さらにこれを BDD で表すものである. 順序回路の検証やテスト生成の分野では状態遷移グラフを非明示的に表現することにより 2^{50} を越える状態を持つものを計算機上で扱う手法 [3, 4, 9] が発表されているが, 本手法はこれを BDD に対して適用したものである. 非明示的な表現のサイズは必ずしも元の BDD のサイズに依存せず, BDD の構造に規則性があれば大幅な記憶量削減が達成される. 一方, 表現の一意性を犠牲にしていないため, 等価性判定を表現のサイズに比例する時間で行うことができる.

実験の結果, 加算関数やセレクタ関数などの記憶効率のオーダーが改善され, MCNC, ISCAS85 のベンチマークにおいても多くのもので BDD よりも必要記憶量が少くて済むことが確かめられた.

以下, 2章で BDD を定式化した後, 3章で BDD の非明示的表現の詳細を述べる. また 4章では, BDD の非明示的表現の間の論理演算手法について述べる. 最後に各種の論理関数, ベンチマークに対する実験結果を5章で示し, 6章で結論を述べる.

2 二分決定グラフ

2.1 OBDD と LOBDD

二分決定グラフ (BDD) は非巡回有向グラフによる論理関数の表現である.

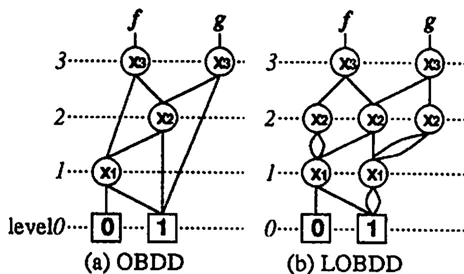


図 1: 二分決定グラフ

図 1(a)(b) はいずれも論理関数 $f = x_3x_2 + x_1$, $g = x_3 + x_2 + x_1$ を表現する BDD である. BDD 中の非終端節点は変数 ($\in \{x_1, x_2, \cdots, x_n\}$) によってラベル付けされており, **変数節点**と呼ばれる. 変数節点 v にラベル付けされた変数を $var(v)$ で表す. 終端節点は論理値 0 または 1 によってラベル付けされており, **定数節点**と呼ばれる. 定数節点 v にラベル付けられた論理値を $c(v)$ で表す. また, BDD は順序付けられた m 個の初期節点 $\vec{i} = (i_1, \cdots, i_m)$ (図中では $\vec{i} = (f, g)$) を持つ. i_k は変数節点, 定数節点のいずれでも良い. BDD の各節点 v には次式により与えられるレベル $l(v)$ が定義される.

$$\begin{cases} v \text{ が定数節点のとき } l(v) = 0, \\ v \text{ が変数節点で } var(v) = x_i \text{ のとき } l(v) = i. \end{cases}$$

変数節点からは順序付けられた 2 本の枝が出ており, これを 0 枝, 1 枝と呼ぶ. 変数節点 v の 0 枝, 1 枝に接続する節点をそれぞれ $e(v, 0)$, $e(v, 1)$ で表す. 以下, 本稿の図中では 0 枝を左側, 1 枝を右側に示す. BDD にはいくつかの種類があるが, 本稿では順序付き BDD (Ordered BDD; 以下 OBDD と略す) を扱う. これは全ての節点 v について

$$l(v) > l(e(v, 0)), \quad l(v) > l(e(v, 1))$$

が成り立つものである. 図 1(a) (b) はいずれも OBDD である. OBDD 中の変数節点の数を OBDD のサイズということにする.

OBDD の各節点はそれぞれ一つの論理関数を表現する. 節点 v が表現している論理関数は次のように定義される.

$$f_v = \begin{cases} c(v) & \cdots v \text{が定数節点の場合}, \\ \overline{var(v)} \cdot f_{e(v,0)} + var(v) \cdot f_{e(v,1)} & \\ & \cdots v \text{が変数節点の場合}. \end{cases}$$

OBDD のうち, 全ての変数節点 v について

$$l(e(v, 0)) = l(v) - 1, \quad l(e(v, 1)) = l(v) - 1$$

が成り立つもの, すなわち, 枝にレベルの飛び越しがない OBDD を levelized OBDD (以下 LOBDD と略す) という. 図 1(b) は (a) の OBDD と同じ論理関数を表す LOBDD である. LOBDD の初期節点のレベルは全て n である. この LOBDD を改めて次のように定義する.

定義 2.1 n 変数 m 出力論理関数を表現する *LOBDD L* は 4 つ組 $L = (\vec{N}, \vec{e}, c, \vec{i})$ である. ただし,

- $\vec{N} = (N_0, \cdots, N_n)$: N_j はレベル j の節点の集合.
- $\vec{e} = (e_1, \cdots, e_n)$: $e_j(v, x)$ は写像 $N_j \times \mathcal{B} \to N_{j-1}$ であり, レベル j の節点 v の x 枝に接続する節点を表す.
- c : $c(v)$ は写像 $N_0 \to \mathcal{B}$ であり, 定数節点に割り当てられている論理値を表す.
- $\vec{i} = (i_1, \cdots, i_m)$: $i_k \in N_n$ は初期節点. □

LOBDD 中の変数節点の数を LOBDD のサイズということにする.

LOBDD のレベル j において次を満たす節点 u, v を等価な節点という.

$$\begin{cases} c(u) = c(v) & u, v \text{ が定数節点の場合}, \\ e_j(u, 0) = e_j(v, 0) \text{ かつ } e_j(u, 1) = e_j(v, 1) & \\ & u, v \text{ が変数節点の場合}. \end{cases}$$

LOBDD の等価な節点の削除 (図 2) をこれがなくなるまで行うことを LOBDD の既約化といい, この結果得られる LOBDD を既約な LOBDD という. 既約な LOBDD から冗長な節点 ($e(v, 0) = e(v, 1)$ を満たす節点 v) をすべて除去して得られる OBDD が既約な OBDD である. OBDD の場合 [2] と同様, 任意の論理関数 f に対し, f を表す既

約な LOBDD の形は, 変数の順序を固定すると一意に定まることが示せる.

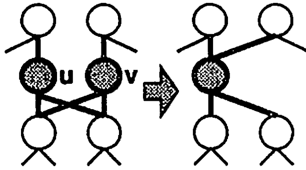


図 2: 等価な節点の削除

2.2 LOBDD の演算

OBDD, LOBDD に対する演算には, 二項論理演算, 単項論理演算, 等価性判定, 代入演算, などがある. このうち, 最も本質的で重要なものは二項論理演算である.

LOBDD の二項論理演算は, 論理関数 f, g を表す 2 つの LOBDD と二項演算子 $\circ$ (論理積, 論理和など) より論理関数 $f \circ g$ を表す LOBDD を求めるというものである. この演算は, apply 演算とも呼ばれ, 2 つの論理関数を表すグラフをトラバースする再帰的アルゴリズムにより実現されるが [2], 本稿ではこれを一般化し, 積 LOBDD の考え方を用いて定式化する.

積 LOBDD とは, 順序機械の「積機械 (product machine)」に相当するものである. 二つの LOBDD L_1, L_2 の二項演算 $\circ$ に関する積とは図 3 のように 2 つの LOBDD の対応するレベルの節点の対を新たな節点とし, v から v', u から u' に枝があるとき (u, v) から (u', v') に枝を設けたものである. ただし, 定数節点では論理演算 $\circ$ を行い, 例えばこれが論理積であれば図 3 のように定数節点の値が定められる.

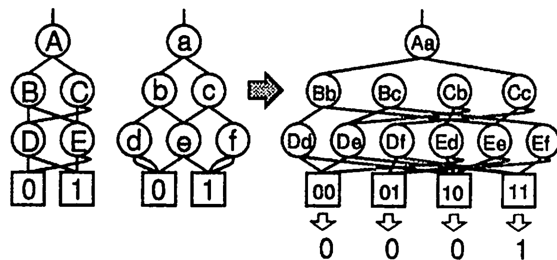


図 3: 積 LOBDD の構築

積 LOBDD の形式的な定義は次のとおりである. ただし, ここでは各 LOBDD の初期節点は 1 つであり, LOBDD は 1 つの論理関数を表すものとする.

定義 2.2 *2 つの LOBDD* $L_1 = (\vec{N^1}, \vec{e^1}, c^1, \vec{i^1})$, $L_2 = (\vec{N^2}, \vec{e^2}, c^2, \vec{i^2})$ *の二項演算子* $\circ$ *に関する積 LOBDD を* $L_{prd} = prod(L_1, L_2, \circ)$ *で表す. ただし,* $prod(L_1, L_2, \circ) = (\vec{N^{prd}}, \vec{e^{prd}}, c^{prd}, \vec{i^{prd}})$ *は次により定義される.*

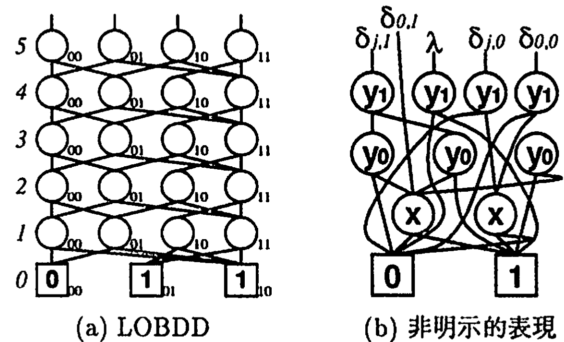
- $N_j^{prd} = N_j^1 \times N_j^2$.
- $e_j^{prd}((y^1, y^2), x) = (e_j^1(y^1, x), e_j^2(y^2, x))$.
- $c^{prd}((y^1, y^2)) = c^1(y^1) \circ c^1(y^2)$.
- $\vec{i^{prd}} = ((i_1^1, i_1^2))$.

□

論理関数 f と g を表す LOBDD L_f と L_g (いずれも 1 出力とする) に対する論理演算 $\circ$ の結果 $h = f \circ g$ を表す LOBDD L_h は $L_h = prod(L_f, L_g, \circ)$ により与えられ, これに既約化を施せば h を表す LOBDD の既約形が求められる.

3 二分決定グラフの非明示的表現

本稿では LOBDD を非明示的に表現し, 操作することを考える. 以下本章では **LOBDD の非明示的表現** (implicit representation of LOBDD: 以下 iLOBDD と略す) について説明する.



(a) LOBDD (b) 非明示的表現

図 4: LOBDD とその非明示的表現

3.1 LOBDD の非明示的表現 (iLOBDD)

iLOBDD は LOBDD の節点を 2 進符号化し, 枝の接続関係を論理関数ととらえ, これを BDD で表したものである. まず図 4 の例を用いてその概念を説明する.

図 4(a) の LOBDD のレベル 5 とレベル 4 との接続を考える. まずそれぞれのレベルの節点に対して, レベル毎に独立に 2 進符号 (図中では節点の右横に書き添えたもの) を与える. このとき, レベル 5 の 01 という符号の節点の 0 枝, 1 枝はそれぞれレベル 4 の 00, 11 という符号の節点に接続している. レベル 5 の節点 y から出る x 枝の接続先を関数 $\delta_5(y, x)$ で表すとすると, この関係は

$$\delta_5((0,1),0) = (0,0), \quad \delta_5((0,1),1) = (1,1)$$

と表すことができる. ここで δ_5 は 3 入力 2 出力の論理関数と見ることができるので, レベル 5 の全ての節点に対する関係を論理式で表すと

$$\begin{aligned}\delta_5((y_1,y_0),x) &= (\delta_{5,1}((y_1,y_0),x),\ \delta_{5,0}((y_1,y_0),x)) \\ &= (y_1y_0 + y_1x + y_0x,\ y_1\overline{y_0} + x)\end{aligned}$$

と表すことができる. 同様に 4 − 3, 3 − 2, 2 − 1 の各レベル間について $\delta_4, \cdots, \delta_2$ を求めると, この例では δ_5 と同じ関数が得られる. 1 − 0 のレベル間では

$$\delta_1((y_1,y_0),x) = (x,\ y_1\overline{x})$$

となる.

定数節点 00, 01, 10 の論理値はそれぞれ 0, 1, 1 であるから, これを 2 入力 1 出力の関数 λ で表すと

$$\lambda((y_1,y_0)) = y_1 + y_0$$

が得られる.

このように LOBDD の枝の接続関係, 定数節点の論理値は δ_j, λ という論理関数によって表現することができる

が, これらの論理関数を OBDD で表現したもの (図 4(b)) が iLOBDD である. 図 4(a) の LOBDD のサイズは 20 だが, その非明示的表現 ((b) の多出力 OBDD) のサイズは 9, と半分以下になる.

一般に, n 変数 m 出力論理関数を表現する LOBDD L のレベル j の節点を w_j ビットで 2 進符号化すると, この LOBDD を表す iLOBDD I は次のような 3 つ組として表される. ただし, $\sigma_j : N_j \to \mathcal{B}^{w_j}$ は L の j レベルの節点の符号を表すものとする.

定義 3.1 *n 変数 m 出力の LOBDD $L = (\vec{N}, \vec{e}, c, \vec{i})$ を符号化 σ のもとで表す iLOBDD I は 3 つ組 $I = (\vec{\delta}, \lambda, \vec{s})$ である. ただし,*

- $\vec{\delta} = (\delta_1, \cdots, \delta_n)$: 写像 $\delta_j : \mathcal{B}^{w_j} \times \mathcal{B} \to \mathcal{B}^{w_{j-1}}$ は j レベルの節点の枝が接続する節点の符号を表し, $\delta_j(\sigma_j(v), x) = \sigma_{j-1}(e_j(v, x))$ を満たす.
- $\lambda : \mathcal{B}^{w_0} \to \mathcal{B}$ は定数節点に割り当てられている論理値を表し, $\lambda(\sigma_0(v)) = c(v)$ を満たす.
- $\vec{s} = (s_1, \cdots, s_m)$ は初期節点を表し, $s_j = \sigma(i_j)$ を満たす. □

iLOBDD が表現する論理関数とは, iLOBDD が非明示的に表現している LOBDD が表現する論理関数を意味する. また, iLOBDD を構成する論理関数群を表現する多出力 OBDD のサイズを iLOBDD のサイズという.

非明示的表現法による記憶量の削減は次の 2 点で働く.

i) 複数のレベル間の接続関係の類似性

図 4 の例のように, あるレベル間の接続関係が他のレベル間の接続関係と等しい, または類似しているとき, それらを OBDD で表すと節点が共有されて節点数が少なくなる.

ii) レベル間の接続関係の規則性

あるレベル間の接続関係が規則的であるとき, これを表す論理関数 δ_j が簡単になり, 必要記憶量が減少する.

OBDD や LOBDD の場合には, 変数の順序が定められていて, 既約化が行われていれば, 任意の論理関数に対して表現が一意に定まった. iLOBDD の場合には次の条件のもとで表現が一意に定まる.

性質 3.1 *同じ論理関数 f を表す 2 つの iLOBDD I_1 と I_2 は, 次の条件が満たされるとき一致する.*

1) I_1 と I_2 は同じ変数順で既約な LOBDD を表している.

2) I_1 と I_2 は同じ節点符号のもとで LOBDD を表している.

3) どの節点も表さない符号に対する関数の値が, I_1 と I_2 において等しい. □

1) は I_1 と I_2 の表す LOBDD の形が等しくなる条件である. 2) のもとでは, 節点を表す符号に対する δ_j, λ の値が一意に定まる. 3) は, 全ての符号語に対して関数の値が等しいことを保証するものであり, これによって I_1 と I_2 の関数 δ_j, λ が完全に一致する.

対応する節点が存在しない符号語に対する関数の値はすべて論理値 0 と決めれば 3) の条件は満たされる. あるいは generalized cofactor[9] を用いてもこの条件は満たされる.

3.2 節点の符号化

iLOBDD の記憶効率は符号化に大きく依存する. 本稿では iLOBDD の節点の符号化として**最左パス符号と 2 進符号**を提案する. これらの符号は,

i) 各論理関数に対して符号が一意に定まる,

ii) 後に述べるとおり, 任意の符号による iLOBDD を本符号による iLOBDD に変換する非明示的アルゴリズムが存在する,

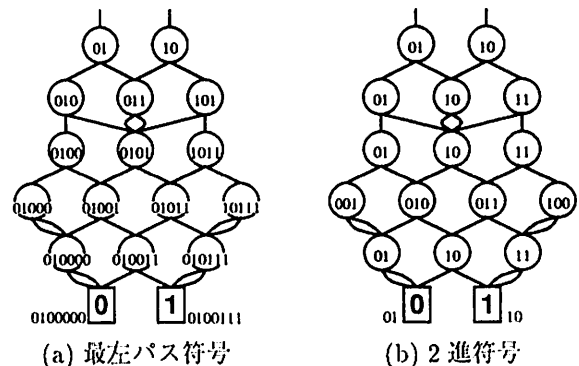
という特長を持っている. この性質により, 各論理関数に対する iLOBDD を一意に決めることができるため, 等価性判定を容易に行うことが可能になる. この意味で, 我々はこれらの符号化を iLOBDD の標準符号と呼ぶ.

最左パス符号

最左パス符号は, 入力変数の数を n, 初期節点の数を m として, j レベルの節点に $\lceil \log(m+1) \rceil + (n-j)$ ビットの符号を用いるものであり, 以下の規則により各節点 v の符号 $\sigma(v)$ を定める (図 5(a) 参照).

i) 初期節点 s_k には k の 2 進符号を割り当てる. $(\sigma(s_k) = binary(k))$

ii) 節点 u の x 枝が v に接続しているとき, u の符号 $\sigma(u)$ の末尾に x を連接したもの $(\sigma(u) \| x$ と表す) を v の符号とする. $(\sigma(v) = \sigma(u) \| x)$ ただし, $\delta(u, x) = v$ なる (u, x) の組合せが複数存在するときは, その中で符号を 2 進数として見たときに最も小さくなるものを選ぶ.



(a) 最左パス符号　(b) 2 進符号

図 5: 標準符号

2 進符号

2 進符号は, レベル j の節点が W_j 個であるとき, $\lceil \log(W_j + 1) \rceil$ ビットを用いてその節点を符号化するものであり, 符号化の規則は次により与えられる.

i) 1 番目の初期節点 i_1 から始めて LOBDD を深さ優先順にたどり, 各節点に対して訪問順に昇順の番号をつける. 番号を付けられなかった節点があれば, 順次 $i_2, i_3, \cdots$ から同様に深さ優先に節点をたどり, 番号をつける.

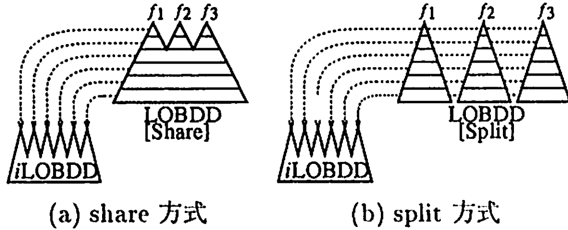
ii) 各レベル j について, 節点を i) でつけた番号順にソートし, 番号の小さい順に $1, \cdots, W_j$ の 2 進表現を符号として与える.

3.3 複数の論理関数の表現法

iLOBDD を用いて m 個の論理関数を同時に表現する方法としては, 次の二つが考えられる.

i) m 出力の iLOBDD により m 個の論理関数を同時に表現する. (**share 方式**)

ii) 1 出力の iLOBDD を m 個用意することにより m 個の論理関数を表現する. (**split 方式**)



(a) share 方式　(b) split 方式

図 6: 複数の関数の表現法

LOBDD の場合には, サブグラフの共有ができない split 方式の方のサイズが必ず大きくなるが, 非明示的表現を構成する OBDD はサブグラフを共有する方式なので, 必ずしも split 方式が不利とは限らない. 逆に, split 方式では share 方式に比べて符号長を短くできたり, 関数毎の LOBDD の規則性を符号に反映できるため, かえってサイズが小さくなることがあり得る. 後に実験で示すとおり, 実際には多くの場合に split 方式の方が有利となる.

4 二分決定グラフの非明示的操作法

前章では, BDD の非明示的表現法が従来の BDD より記憶効率に優れることを述べたが, さらに BDD に対する演算を非明示的に行うことができれば, 中間結果の記憶量の制約からこれまで扱い切れなかった大規模な論理関数の計算が行える可能性が出てくる. 演算を非明示的に行うとは, iLOBDD I_1 と I_2 が表現している論理関数に対する論理演算の結果を直接 iLOBDD の形で得ることである. 本章では, BDD に対する演算のうち, もっとも本質的で計算量の大きな二項演算 (apply 演算) に対する非明示的アルゴリズムを提案する.

4.1 二項演算の処理の流れ

二項演算を行うためには複数の論理関数を表現する必要があるが, その表現法としては前章で述べた split 方式を用いるものとする. すなわちここでは論理関数 f と g を表す 2 つの 1 出力 iLOBDD I_1, I_2 と論理演算子 $\circ$ から $h = f \circ g$ なる論理関数 h を表す 1 出力 iLOBDD I を求めるものとする.

この二項演算は, 2.2 で定義した積 LOBDD を I_1 と I_2 から非明示的に構築することにより実現できる. 本稿では, さらにこれを非明示的に既約化し, 標準符号に変換するアルゴリズムを示す.

積 iLOBDD の構築

二つの iLOBDD I_1, I_2 が表す LOBDD をそれぞれ L_{I_1}, L_{I_2} とする. L_{I_1} と L_{I_2} の演算 $\circ$ に関する積 LOBDD を表す iLOBDD I_{prd} を I_1 と I_2 の演算 $\circ$ に関する積 iLOBDD と呼ぶ.

$I_1 = (\vec{\delta^1}, \lambda^1, \vec{s^1})$, $I_2 = (\vec{\delta^2}, \lambda^2, \vec{s^2})$, $I_{prd} = (\vec{\delta^{prd}}, \lambda^{prd}, \vec{s^{prd}})$ とすると I_1, I_2 および $\circ$ から I_{prd} を構築する積 iLOBDD 構築のアルゴリズムは 3.2 の符号の定義をほぼそのまま手続き化することにより得られる.

1) 初期節点: $s_1^{prd} \leftarrow s_1^1 \| s_1^2$.

2) 2a) を $j = n, \cdots, 1$ について繰り返す.

　2a) 枝: $\delta_j^{prd}(y_1 \| y_2, x) \leftarrow \delta_j^1(y_1, x) \| \delta_j^2(y_2, x)$.

3) 定数節点: $\lambda^{prd}(y_1 \| y_2) \leftarrow \lambda^1(y_1) \circ \lambda^2(y_2)$.

既約化

2.1 でも述べた通り, LOBDD の既約化とは, 等価な節点を可能な限り削除することである. LOBDD の等価な節点の削除は, 等価な節点の集合を見つけ出してその中から代表元を一つ選択し, 上位のレベルからその他の節点に接続している枝を全てその代表元へつなぎ換える操作と考えることができる. (図 8参照). LOBDD の場合にはレベルの飛び越しがないので, レベル 0 から始めて順にレベル n まで各レベルの等価な節点の削除を行っていけば既約化が完了する. したがって, LOBDD の既約化の手順は次のようになる.

for $j = 0$ **to** $n-1$

1) レベル j の節点を等価な節点の集合 $E^1, \cdots, E^p$ に分ける.

2) 各 E^k について代表元 y_0^k を一つ選ぶ.

3) 各 E^k について E^k の節点 y_l^k に入る枝を y_0^k へつなぎ換える.

以下, 上記 1)〜3) の処理を iLOBDD 上で非明示的に行う方法を詳しく述べる.

1) レベル j の等価な節点の集合 $E^1, \cdots, E^p$ への分割

このステップの結果の表現には, 等価関係関数 (equivalence relation function) $E_j : \mathcal{B}^{w_j} \times \mathcal{B}^{w_j} \rightarrow \mathcal{B}$ を用いる. これは, 2 つの節点の符号を入力としてその節点が同じ集合に属するときのみ 1 を返す論理関数である. j レベルの等価関係関数 E_j は次のように計算できる.

a) $j = 0$ のとき

$E_0(y, y') \leftarrow \lambda(y) \equiv \lambda(y')$

b) $j > 0$ のとき

$E_j(y, y') \leftarrow \forall x(\delta_j(y, x) \equiv \delta_j(y', x))$

ただし, $p \equiv q$ は $\overline{p \oplus q}$ であり, 両者の値が等しいときにのみ 1 となる.

2) 等価な節点の集合からの代表元の選択

この操作は compatibility projection[7] を用いて実現する. これは等価関係関数が与えられたとき, 等価関係にある符号の集合から, 数値として見た場合に最小である符号を代表元として選択するものである. 結果は, 節点符号 y, y' を入力とし, y' が y の属する等価集合の代表元であるときに限り 1 を返す関数 $\hat{E}(y, y')$ によって与えられる. これを

$$\hat{E}(y, y') \leftarrow cproj(E(y, y'))$$

と表記することにする.

例えば図 7 の左に示すような等価関係関数 E が与えられた場合 (枝は等価な節点を表す), E に対して compatibility

projection を施すと図中央の関係関数 $\hat{E}$ が得られる.

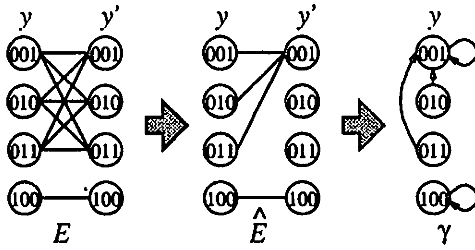


図 7: compatibility projection を用いた代表元の選択

次に, 節点の符号 y を入力とし, y が属する等価な節点集合の代表元の符号を出力とする関数 γ を $\hat{E}$ より求める. j レベルのこの関数は

$$\gamma_j(y) \leftarrow \exists y' \ \hat{E}_j(y, y')$$

により求めることができる. ただし, $\exists y f(y, \vec{x})$ は存在限定演算を表し, $\exists y \ f(y, \vec{x}) = f(0, \vec{x}) + f(1, \vec{x})$ である.

3) 上位レベルからの枝のつなぎ換え

変更後の枝の関数 δ'_{j+1} は

$$\delta'_{j+1}(y, x) \leftarrow \gamma_j(\delta_{j+1}(y, x))$$

により求めることができる (図8参照).

以上をまとめると, iLOBDD の既約化のアルゴリズムは次のように表せる.

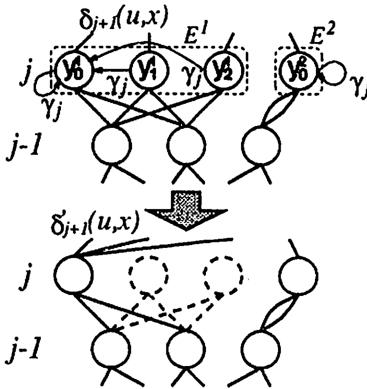


図 8: iLOBDD の既約化

1) $E_0(y, y') \leftarrow \lambda(y) \equiv \lambda(y')$.
2) 2a)～2d) を $j = 1, 2, \cdots, n$ について繰り返す.
   2a) $\hat{E}_{j-1}(y, y') \leftarrow cproj(E_{j-1}(y, y'))$.
   2b) $\gamma_{j-1}(y) \leftarrow \exists y' \ \hat{E}_{j-1}(y, y')$.
   2c) $\delta'_j(y, x) \leftarrow \gamma_{j-1}(\delta_j(y, x))$.
   2d) $E_j(y, y') \leftarrow (\delta'_j(y, 0) \equiv \delta'_j(y', 0)) \cdot (\delta'_j(y, 1) \equiv \delta'_j(y', 1))$.

再符号化

iLOBDD の二項演算を行うにあたり, 積 iLOBDD のステップにより得られた iLOBDD の節点の符号長はオペランドの iLOBDD I_1, I_2 の節点の符号長の和となっている. また, 既約化のステップでは符号長は変化しないので, 二項演算を繰り返すと符号は際限なく長くなる. 符号長を適切な範囲に保つという意味からもこの再符号化のステップは不可欠である. 以下では, 任意の符号に基づく iLOBDD I を標準符号に基づいて同じ LOBDD を表す iLOBDD I' に変換するアルゴリズムを示す.

3.2 の定義のとおり, 最左パス符号の符号は, 初期節点 (レベル n) から始めてレベル $n-1, \cdots, 0$(定数節点) の順に決定される. 再符号化の処理もこの順にレベル毎に行う.

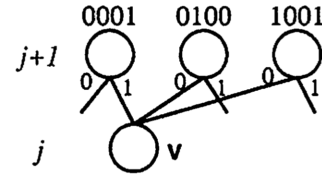


図 9: 最左パス符号における符号の選択

図9は, レベル $j+1$ の最左パス符号からレベル j の最左パス符号を得る様子を示したものである. 節点 v の符号の候補となるのは 00011, 01000, 10010 の 3 つである. すなわち, $\delta_{j+1}(y, x) = v$ なる y と x を連接したものが v の符号の候補となる. v の最左パス符号には, これらの候補の中で 2 進数と見た場合に最小となるものが選択される.

最左パス符号への変換は, 初期節点 (レベル n) から始めて, レベル $n-1, \cdots, 0$(定数節点) の順にこの操作を行っていくとにより実現される.

レベル n の節点に符号を与える.

for $i = n-1$ down to 0

1) j レベルの各節点 y_k に対し, その節点に割り当てられ得る最左パス符号の候補の集合 C_j^k を作成する.
2) 各 C_j^k について代表元を選ぶ.
3) j+1 レベルの枝の関数を, 選択した代表元を出力するように変換する.

以下, 各ステップの詳細を述べる.

1) 同一節点に割り当てられ得る符号の集合の作成

このステップの結果の表現には, 節点の元の符号と最左パス符号 (符号長 w'_j) を入力としてそれらの符号が同じ節点を表しているときのみ 1 になるという関係関数 $C_j : \mathcal{B}^{w_j} \times \mathcal{B}^{w'_j} \rightarrow \mathcal{B}$ を用いる.

j レベルにおける関係関数 C_j は次のように計算できる. ここで $\bar{P}_{\hat{C}_{j+1}} : \mathcal{B}^{w'_{j+1}} \rightarrow \mathcal{B}^{w_{j+1}}$ はレベル $j+1$ の元の符号を入力とし, 再符号化された新しい符号を出力する関数である.

a) $j = n$ のとき

$C_n(y, y') \leftarrow (y \equiv s_1) \cdot (y' \equiv binary(1))$

b) $j \leq n-1$ のとき

$C_j(y, y') \leftarrow \exists \check{y} \ \exists x \ ((y \equiv \delta_{j+1}(\bar{P}_{\hat{C}_{j+1}}(\check{y}), x)) \cdot (y' \equiv \check{y} \| x))$

2) 代表元の選択

代表元の条件が「数値としてみた場合に最小となる符号」であるから C_j に対して前述の compatibility projection を施せば, 代表元との対応を与える関係関数 $\hat{C}_j$ が得られる. さらに既約化の場合と同様に, 節点の元の符号を入力とし, その節点に対応する代表元を出力する関数 $P_{\hat{C}_j}(y)$ およびその逆関数 $\bar{P}_{\hat{C}_j}(y')$ を求める.

3) 上位レベルからの枝の出力の変換

変

更後の枝の関数は $\delta'_{j+1}(y',x) \leftarrow P_{\hat{C}_j}(\delta_{j+1}(\bar{P}_{\hat{C}_{j+1}}(y'),x))$ により求めることができる.

以上をまとめると iLOBDD の最左パス符号による再符号化のアルゴリズムは次のように表せる.

1) $C_n(y,y') \leftarrow (y \equiv s_1)\cdot(y' \equiv binary(1))$.

2) $P_{\hat{C}_n}(y) \leftarrow \exists y'\ \hat{C}_n(y,y')$,
$\bar{P}_{\hat{C}_n}(y') \leftarrow \exists y\ \hat{C}_n(y,y')$.

3) 3a) ~ 3d) を $j=n-1,\cdots,0$ について繰り返す.

3a) $C_j(y,y') \leftarrow \exists\tilde{y}\ \exists x\ ((y \equiv \delta_{j+1}(\bar{P}_{\hat{C}_{j+1}}(\tilde{y}),x))\cdot(y' \equiv \tilde{y}||x))$.

3b) $\hat{C}_j(y,y') \leftarrow cproj(C_j(y,y'))$.

3c) $P_{\hat{C}_j}(y) \leftarrow \exists y'\ \hat{C}_j(y,y')$,
$\bar{P}_{\hat{C}_j}(y') \leftarrow \exists y\ \hat{C}_j(y,y')$.

3d) $\delta'_{j+1}(y',x) \leftarrow P_{\hat{C}_j}(\delta_{j+1}(\bar{P}_{\hat{C}_{j+1}}(y'),x))$.

4) $\gamma'(y') \leftarrow \gamma(\bar{P}_{\hat{C}_0}(y'))$.

2 進符号による iLOBDD は, 一旦最左パス符号に変換した後に符号圧縮を施すことにより得られる. この符号圧縮処理も非明示的に行うことができる [5].

5 実験結果および考察

MCNC, ISCAS85 のベンチマークおよびいくつかの組合せ論理回路を対象にその論理関数を表現し, 表現のサイズを比較した. 実験の手順は以下のとおりである.

1) ベンチマーク, 論理回路の回路記述から既約な OBDD を作成.

2) 既約な OBDD を既約な LOBDD に変換.

3) 既約な LOBDD から iLOBDD を構築.

ただし, iLOBDD の符号化には 2 進符号を用い, 対応する節点のない符号語に対する論理関数の値は generalized cofactor 演算 [9] により定めた. OBDD の演算は, 文献 [8] の SBDD パッケージを用いて行った.

表 1 は MCNC ベンチマークに対する結果である. 各欄の意味は, 左から順に回路名, 入力数, 出力数, 既約な OBDD の節点数, 既約な LOBDD(share, split 両方式) の節点数, iLOBDD(share, split 両方式) の節点数である. ただし, OBDD ならびに iLOBDD の表現に用いた SBDD パッケージには否定枝, 入力反転枝, 変数シフト枝が実装されているため, 節点数は純粋な OBDD の節点数より少なくなっている. このパッケージによる 1 節点あたりの記憶量は約 20 バイトである. 表最下段は, 表に掲載していないものを含めたすべての回路に対するそれぞれの表現の節点数の比の平均である. 変数順は文献 [6] の最適変数順 (OBDD のサイズ最小) を用いた.

OBDD と iLOBDD(share, split) の比較より, 本稿の非明示的表現の効果を見ることができる. share 方式では一つも OBDD に勝るものはないが, split 方式では 33 中 28 の回路で記憶量の削減が見られ, 平均で 44% に削減されていることがわかる.

多出力関数の表現法という観点から見ると, 明示的な LOBDD 表現では share 方式が優れるが, 非明示的な iLOBDD 表現では逆に split 方式が優れる. LOBDD と iLOBDD とを比較すると節点数は split 方式で 11% にまで削減されており, 非明示的表現そのものの効果は大きいといえる. この傾向は, 実験を行った他の論理関数についても見られた.

表 2 は ISCAS85 ベンチマーク回路に対して同様の実験を行った結果である. 論理関数の変数順は, 回路記述から抽出したままの変数順を用いた. OBDD と iLOBDD との比較より, MCNC ベンチマークより規模の大きな論理関数に対しても記憶量が削減されることがわかる. c1908 を除く回路で節点数が大幅に減少し, 必要記憶量は平均で 48% に削減されている.

表 2: 実験結果 (ISCAS85 ベンチマーク)

回路	in	out	OBDD	iLOBDD (split)
c432	36	7	1706	964
c499	41	32	25331	7274
c1355	41	32	25331	7274
c1908	33	25	27205	31325
average			1.00	0.48

表 3 は, いくつかの論理関数について OBDD 表現, iLOBDD 表現 (split) を比較したものである. 各表のオーダーは得られているデータから回帰分析により得たものである.

表 3 (a) は n 入力多数決関数に対する結果である. 従来の OBDD のサイズはおおよそ $O(n^2)$ で増大するが, iLOBDD の増加は $O(n^{1.35})$ と小さい. 一般に何らかの規則性を持つ対象関数に対しては, 同様の効果があると考えられる.

表 3 (b), (c) はセレクタ関数 ($\log n$ 個の制御入力により, n のデータ入力から一つを選んで出力する関数) に対する結果である. (b) は制御入力を上位に配置する変数順を選んだ場合で, (c) はその逆の変数順を選んだ場合である. OBDD の場合, (b) の変数順では $O(n)$ の節点数で済むが, 逆順 (c) では $O(2^n)$ のサイズとなることが知られている. これに対し, iLOBDD では, (b) の変数順では OBDD に劣るものの, 逆順では $O(1)$ と指数的な記憶量削減を達成しており, いずれの変数順でも多項式で押えられている.

表 3 (d)(e) は 2 進数の加算関数 (キャリーを含めて $2n+1$ 入力 $n+1$ 出力) に対する結果である. (d) は 2 つの数を msb から lsb の順にインターリーブさせる変数順, (e) は 2 つの数を直列に lsb から msb の順に入力する変数順である. OBDD では, (d) では $O(n)$, (e) では指数オーダーとなることが知られているが, iLOBDD では, (d) では定数オーダー, (e) でも $O(n)$ のオーダで済んでおり, 大きな記憶量削減が達成されている.

表 1: 実験結果 (MCNC ベンチマーク)

回路	in	out	OBDD	LOBDD (share)	LOBDD (split)	*i*LOBDD (share)	*i*LOBDD (split)
add6	12	7	28	97	221	61	8
alu1	12	8	15	96	222	50	3
alu2	10	8	52	117	256	101	21
alu3	10	8	51	119	245	106	19
apla	10	12	88	140	389	146	36
dk17	10	11	54	114	302	100	13
sao2	10	4	80	110	166	122	31
average			1.00	2.12	4.17	1.78	0.44

このように, いくつかの関数については大きなオーダーの改善が見られ, OBDD では指数サイズであったものが *i*LOBDD では多項式サイズで表現できるものもある. 逆に, 一般に OBDD が多項式サイズである限り *i*LOBDD も多項式サイズであることが示せる.

表 3: 種々の関数のクラスに対する結果

(a) 多数決関数

n	OBDD	*i*LOBDD
4	6	3
16	72	28
64	1056	175
256	16512	830
	$O(n^{1.91})$	$O(n^{1.35})$

(b) セレクタ関数 (制御入力を上位に配置)

n	OBDD	*i*LOBDD
4	4	7
16	16	66
64	64	317
	$O(n)$	$O(n^{1.28})$

(c) セレクタ関数 (データ入力を上位に配置)

n	OBDD	*i*LOBDD
4	15	1
8	255	1
16	65535	1
	$O(2^n)$	$O(1)$

(d) 加算関数 (インターリーブで msb から lsb の順)

n	OBDD	*i*LOBDD
4	21	8
8	41	8
16	81	8
32	161	8
	$O(n)$	$O(1)$

(e) 加算関数 (直列で lsb から msb の順)

n	OBDD	*i*LOBDD
4	64	22
6	238	36
8	916	50
10	3610	64
12	14368	78
	$O(2^{0.98n})$	$O(n)$

6 結論

BDD のグラフとしての構造を論理関数によって間接的に表すことにより必要記憶量の削減を図る手法, およびこの新しい論理関数の表現上で二項演算, 等価性判定などを効率良く行う手法を示した.

各種の論理関数において, 従来の BDD 表現と提案手法による表現との必要記憶量の比較を行った結果, 提案手法が従来の BDD に比べて記憶量を削減することが確認された. 本手法により, 従来の BDD では記憶量の面から扱えない巨大な論理関数を取り扱えることが期待される.

今後の課題としては, LOBDD ではなく OBDD そのものを非明示的に表現し操作する方法の開発, 非明示的表現の効率をさらにあげる良い符号化法の開発などがあげられる.

謝辞

本研究を進めるにあたり, 共同研究の機会を与えて下さるとともに多大な御指導, 御助言を頂きました, 大阪大学工学部情報システム工学科の白川 功教授に対し, ここに厚く御礼申し上げます. また, 本研究に関して貴重なコメントを頂きました, CMU の E. Clarke 教授, 京都大学の浜口清治講師に感謝致します. 最後に, 御討論頂いた, 大阪大学工学部情報システム工学科白川研究室の皆様に感謝致します.

参考文献

[1] S. B. Akers: "Binary decision diagrams," *IEEE Trans. Comput.*, vol. C-27, no. 6, pp. 509–516 (June 1978).

[2] R. E. Bryant: "Graph-based algorithms for Boolean function manipulation," *IEEE Trans. Comput.*, vol. C-35, no. 8, pp. 677–691 (Aug. 1986).

[3] J. R. Burch, et al.: "Sequential circuit verification using symbolic model checking," in *Proc. ACM/IEEE 27th DAC*, pp. 46–51 (June 1990).

[4] 崔他: "論理関数処理に基づく順序回路のテスト生成法," 信学論, J76-A, 6, pp. 835–843 (June 1993).

[5] K. Hamaguchi and E. Clarke: private communication (Oct. 1994).

[6] N. Ishiura, H. sawada and S. Yamjima: "Minimization of binary decision diagrams based on exchanges of variables," in *Proc. IEEE ICCAD*, pp. 472–475 (Nov. 1991).

[7] B. Lin and A. R. Newton: "Implicit manipulation of equivalence classes using binary decision diagrams," in *Proc. IEEE ICCD* (1991).

[8] S. Minato, N. Ishiura and S.Yajima: "Shared binary decision diagram with attributed edges for efficient Boolean function manipulation," in *Proc. ACM/IEEE 27th DAC*, pp. 52–57 (June 1990).

[9] H. J. Touati, et al.: "Implicit state enumeration of finite state machines using BDD's," in *Proc. IEEE ICCAD*, pp. 130–133 (Nov. 1990).