二分決定グラフの非明示的表現法とその操作法
Implicit Representation and Manipulation of Binary Decision Diagrams

山内 仁
Hitoshi YAMAUCHI
大阪府立大学工学部
University of Osaka Prefecture

石浦菜岐佐
Nagisa ISHIURA
大阪大学工学部
Osaka University

高橋浩光
Hiromitsu TAKAHASHI
岡山県立大学情報工学部
Okayama Prefectural University

平成7年4月20日(木)・21日(金)
於 軽井沢プリンスホテル
西館 国際会議場 浅間

主催　電子情報通信学会　回路とシステム研究専門委員会
非線形問題研究専門委員会
VLSI設計技術研究専門委員会
ディジタル信号処理研究専門委員会
コンカレント工学研究専門委員会
協賛　IEEE Circuits and Systems Society Tokyo Chapter

二分決定グラフの非明示的表現法とその操作法

Implicit Representation and Manipulation of Binary Decision Diagrams

山内 仁
Hitoshi YAMAUCHI
大阪府立大学工学部
University of Osaka Prefecture

石浦菜岐佐
Nagisa ISHIURA
大阪大学工学部
Osaka University

高橋浩光
Hiromitsu TAKAHASHI
岡山県立大学情報工学部
Okayama Prefectural University

本稿では, 論理関数の表現法として Binary Decision Diagram (BDD) の非明示的 (implicit) 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 により表現する方法で, 元の BDD の構造に規則性がある場合, 大幅な記憶量の削減が達成される. BDD の持つ表現の一意性を犠牲にしていないため, 関数の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. 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.

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ている. 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 を用いても表現できない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, これを BDD で表すものである. 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. また, 表現の一意性を保存しているため, 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(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. BDD 中の変数節点の数をその BDD のサイズという. また, BDD は順序付けられた m 個の初期節点 $\vec{i} = (i_1, \cdots, i_m)$ を持つ. 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.

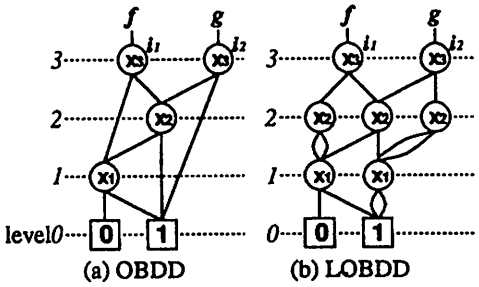


図 1: 二分決定グラフ

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

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. 本稿で扱う BDD は順序付き BDD (Ordered BDD; 以下 OBDD と略す) と呼ばれるものである. これは全ての節点 v について

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

が成り立つものである. 図 1(a) (b) は共に OBDD である.

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

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

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, 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, 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 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.

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

LOBDD の等価な節点の削除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 は, 変数の順序を固定すると一意に定まる.

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 L_f, L_g 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 $L_{f \circ g}$ を求めるというものである. この演算は,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 product machine に相当するものである. LOBDD L_f, L_g の二項演算 $\circ$ に関する積とは図 2 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, 例えば論理積の場合には図 2 のように値が定められる.

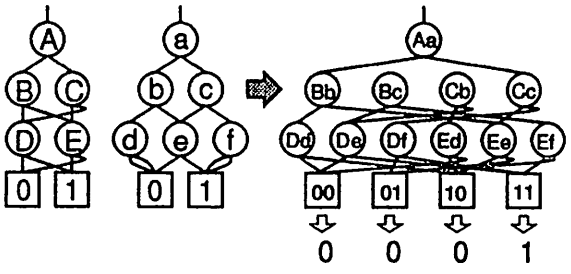


図 2: 積 LOBDD の構築

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.

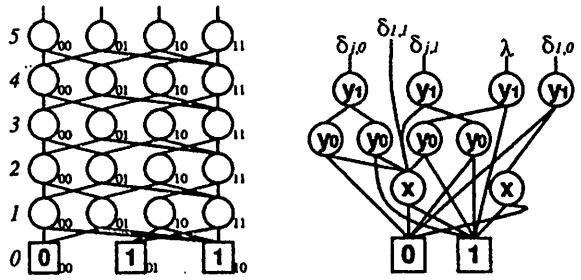
定義 2.2 *LOBDD* $L_f=(\vec{N}^f,\vec{e}^f,c^f,\vec{i}^f)$, $L_g=(\vec{N}^g,\vec{e}^g,c^g,\vec{i}^g)$ の二項演算子 $\circ$ に関する積 *LOBDD* $prod(L_f,L_g,\circ)=(\vec{N},\vec{e},c,\vec{i})$ は次により定義される.

- $N_j=N_j^f \times N_j^g$.
- $e_j((y^f,y^g),x)=(e_j^f(y^f,x),e_j^g(y^g,x))$.
- $c((y^f,y^g))=c^f(y^f)\circ c^f(y^g)$.
- $\vec{i}=((i_1^f,i_1^g))$. □

$f \circ g$ の結果を表す LOBDD は $L_{f\circ g}=prod(L_f,L_g,\circ)$ により与えられる.

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 *i*LOBDD と略す) について説明する.



(a) LOBDD (b) 非明示的表現

図 3: LOBDD とその非明示的表現

3.1 LOBDD の非明示的表現 (*i*LOBDD)

*i*LOBDD は LOBDD の節点を 2 進符号化し, 枝の接続関係を論理関数で表し, これを BDD で表したものである.

図 3 (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 の節点の 0 枝, 1 枝はそれぞれレベル 4 の 00, 11 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

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

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

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

となる. 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

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

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 3(b)) が *i*LOBDD である.

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 *i*LOBDD 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. □

*i*LOBDD が表現する論理関数とは, *i*LOBDD 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 3 のように，レベル間の接続関係が類似している (あるいは等しい) と，これを表す OBDD 内ではサブグラフ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，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 同じ論理関数を表す 2 つの *iLOBDD* I_f と I_g は，次の条件が満たされるとき一致する．

1) I_f と I_g は同じ変数順で既約な *LOBDD* を表す．

2) I_f と I_g は同じ節点符号のもとで *LOBDD* を表す．

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

1) は I_f と I_g 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_f と I_g 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 0 と決めれば 3) 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) 任意の符号による iLOBDD を本符号によるもの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 4(a) 参照)．

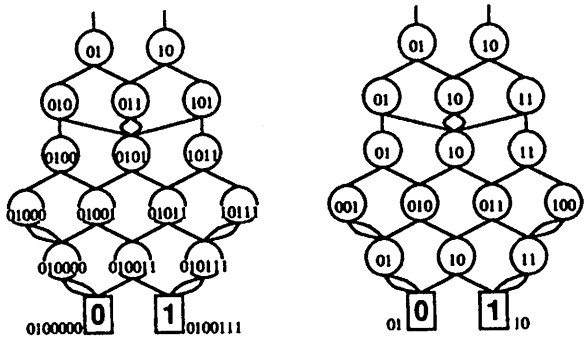
i) 初期節点 s_k には k の 2 進符号を割り当てる．

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．ただし，$\delta(u,x) = v$ なる (u,x) が複数存在すれば，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．

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，$\lceil \log(W_j+1) \rceil$ ビットでその節点を符号化するものである．符号化の規則は次により与えられる (図 4(b) 参照)．

i) LOBDD を初期節点から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．



(a) 最左パス符号　　(b) 2 進符号

図 4: 標準符号

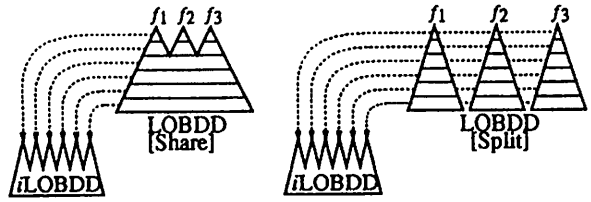
ii) 各レベル j について，節点を i) の番号順にソートし，その順に $1, \cdots, W_j$ の 2 進表現を与える．

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 5: 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

非明示的演算とは，iLOBDD I_f と I_g 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．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_f, I_g と論理演算子 $\circ$ から $f \circ g$ を表す 1 出力 iLOBDD $I_{f \circ g}$ を求める．

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

I_f, I_g が表す LOBDD をそれぞれ L_f, L_g とする. $prod(L_f, L_g, \circ)$ を表す iLOBDD I を I_f と I_g の演算 $\circ$ に関する積 iLOBDD と呼ぶ. $I_f = (\vec{\delta^f}, \lambda^f, \vec{s^f})$, $I_g = (\vec{\delta^g}, \lambda^g, \vec{s^g})$ から, その積 iLOBDD $I = (\vec{\delta}, \lambda, \vec{s})$ を構築するアルゴリズムは 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: $s_1 \leftarrow s_1^f || s_1^g$.
2) 2a) を $j = n, \cdots, 1$ について繰り返す.
   2a) 枝: $\delta_j(y_f || y_g, x) \leftarrow \delta_j^f(y_f, x) || \delta_j^g(y_g, x)$.
3) 定数節点: $\lambda(y_f || y_g) \leftarrow \lambda^f(y_f) \circ \lambda^g(y_g)$.

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. これは, 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7参照). LOBDD 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) $E_j : \mathcal{B}^{w_j} \times \mathcal{B}^{w_j} \to \mathcal{B}$ 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, $v = (v_1, \cdots, v_n)$, $u = (u_1, \cdots, u_n)$ のとき, $v \equiv u = (\overline{v_1 \oplus u_1} \cdot \cdots \cdot \overline{v_n \oplus u_n})$ である.

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 (図6参照). 次に, 符号 y を入力とし, y が属する集合の代表元を出力とする関数 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.

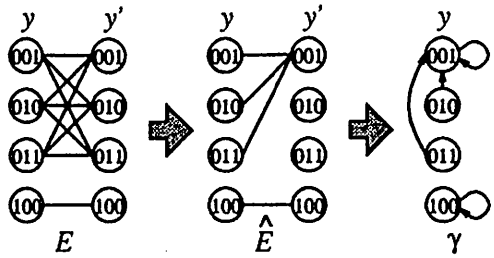


図 6: compatibility projection を用いた代表元の選択

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7参照).

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.

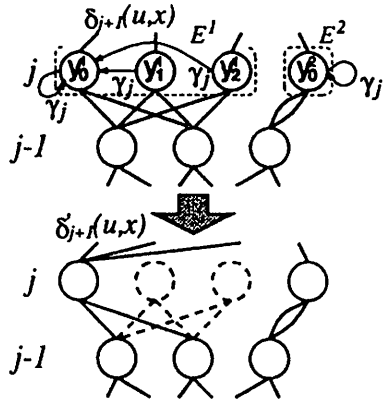
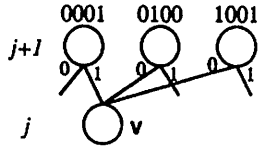


図 7: LOBDD の既約化

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

再符号化



min{00011,01000,10010}= 00011
図 8: 最左パス符号における符号の選択

図8 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, 候補中, 2 進数と見た場合に最小であ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.

1) レベル n の節点に符号を与える.

2) for $j = n-1$ down to 0

2a) j レベルの各節点 y_k に対し, 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.

2b) 各 C_j^k について代表元を選ぶ.

2c) 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 : $\mathcal{B}^{w_j} \times \mathcal{B}^{w'_j} \to \mathcal{B}$ を用いる. レベル j における関係関数 C_j は次のように計算できる. ここで $\bar{P}_{\hat{C}_{j+1}} : \mathcal{B}^{w'_{j+1}} \to \mathcal{B}^{w_{j+1}}$ はレベル $j+1$ の再符号化された新しい符号を入力とし, 元の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「数値として最小である符号」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, 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) $C_j(y,y') \leftarrow \exists\check{y}\ \exists x\ ((y \equiv \delta_{j+1}(\bar{P}_{\hat{C}_{j+1}}(\check{y}),x)) \cdot (y' \equiv \check{y}||x))$.

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

5.1 表現のサイズ

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) 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 の表現には, 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おり, 節点数は純粋な OBDD の節点数より少ない. 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各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 OBDD に勝るものはないが, split 方式では 32 中 27 の回路で記憶量の削減が見られ, 平均で 44% に削減された.

多出力関数の表現法という観点から見ると, 明示的な LOBDD では share 方式が優れるが, 非明示的な iLOBDD 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, 実験を行った他の回路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

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

との比較より, 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 3 は, いくつかの論理関数について OBDD, iLOBDD (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. OBDD のサイズはおおよそ $O(n^2)$ で増大するが, iLOBDD では $O(n^{1.35})$ と小さい. 一般に規則性を持つ対称関数に対しては, 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 個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) はその逆の変数順の場合である. 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 $O(2^n)$ 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 に劣るが, 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 $O(n)$, (e) で指数オーダーとなることが知られているが, iLOBDD では, (d) で

表 1: 実験結果 (MCNC ベンチマーク)

回路	in	out	OBDD	LOBDD (share)	LOBDD (split)	iLOBDD (share)	iLOBDD (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

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.

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

5.2 演算時間

評価用の演算パッケージにおいて, n ビット加算関数を表す iLOBDD の構築に要した時間を表 4 に示す. n ビット加算関数を表す iLOBDD を, 1 変数論理関数を表す iLOBDD から XOR, AND, OR の演算を施すことによって構築した. 変数の順序はインターリーブで msb から lsb の順である. 実験はワークステーション HP712/80 (記憶量 128MB) 上で行った.

表 4: 実行時間

n	CPU (sec)
4	54.43
8	216.98
12	659.48
16	1851.35

結果より, iLOBDD の演算に要する計算量は大きいことがわかる. これは iLOBDD に対する演算処理が複雑なためと考えられる.

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記憶量削減を図る手法, およびこの新しい論理関数の表現上で演算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が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, 多大な御指導, 御助言を頂きました, 大阪大学の白川 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くとともにパッケージを提供して頂いた, CMU の E. Clarke 教授, 京都大学の浜口清治講師に感謝致します.

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).