システムＬＳＩ設計技術　94－6
（2000．1．11）

二分決定グラフの非明示的表現における節点符号化法

山内 仁　　高橋浩光

岡山県立大学情報工学部情報通信工学科
〒719-1197 岡山県総社市窪木 111
Tel: (0866) 94-2111
yamauchi@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し, 問題となっている. これに対して, 著者らは論理関数を表現する BDD のグラフとしての形質を非明示的に表現する BDD の非明示的表現法 (implicit representation of BDDs; 以下 iBDD) を提案し, さらにその演算効率を向上させる手法を提案してきている. 本稿では, iBDD の表現に要する記憶量を大きく左右する, 非明示的に表現している BDD の節点に割り振る符号の符号化手法についていくつか提案する.

キーワード　二分決定グラフ, ゼロサプレス BDD, 非明示的表現, 論理関数, 計算機援用設計

Node Codings for Implicit Representation of Binary Decision Diagrams

Hitoshi Yamauchi　　Hiromitsu Takahashi

Department of Communication Engineering,
Faculty of Computer Science and System Engineering, Okayama Prefectural University
111 Kuboki, Soja, Okayama 719-1197, Japan
Phone: +81-866-94-2111
yamauchi@c.oka-pu.ac.jp

Abstract　Binary Decision Diagrams (BDD) is used on computer aided design of VLSI, because of that's goodness of memory spacing on computers for representing boolean functions. Recently, though, big size boolean functions such that are not able to be represented by BDD appear on VLSI design. We presented an implicit representation of BDDs (iBDD) which is a new representation method by representing characteristics of BDD in meaning of graph implicitly, and its improvement representation method to manipulate faster. In this paper, some node codings are presented. Codings of nodes of BDDs represented implicitly, is effective to iBDD's size.

key words　BDD, zero-supressed BDD, implicit representation, boolean function, computer aided design

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, 論理 CAD の分野だけでなく幅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, さらに記憶効率の良い表現法が求められている.

これに対して, 著者らは BDD の非明示的な表現法とその操作法 (implicit representation of BDDs; 以下 *i*BDD と略す) を提案している [5], [6], [7]. *i*BDD 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, この論理関数を BDD として表すものである. 非明示的表現に必要な記憶量は必ずしも元の BDD のサイズには依存せず, BDD のグラフとしての形質に規則性があれば大幅な記憶量削減が達成されるという優れた特徴を持つ.

*i*BDD により非明示的に表現される BDD は Shared BDD と呼ばれる複数の論理関数を一つの表現で表すものであり, 効率的な表現が可能となっている. さらに表現する BDD に否定枝を用いることにより, 更なる効率化が図られている. また, *i*BDD 上での演算には BDD でも利用されている節点キャッシュ, 演算キャッシュを用いることにより処理時間が短縮されている.

*i*BDD の内部表現として用いる論理関数は組み合わせ論理であるため, 従来の BDD ではなく, 組み合わせ論理の表現に優れるゼロサプレス BDD[8] を用いている. これにより, 内部表現に BDD を用いた場合に比べ記憶量を大幅に削減される [7].

しかしながら, *i*BDD を構築する際に非明示的に表現する BDD の各節点に割り当てる符号により, その表現の記憶量が大きく左右されることが分かっており, 本稿ではいくつかの割り当て符号の符号化法を提案する.

それぞれの提案手法について表現の記憶量を比較する実験を行った結果, 提案した符号化手法により節点数の変動はあまりないが, 演算時間に大きな差が生じることが確かめられた.

以下, 2 節で BDD およびゼロサプレス BDD について簡単に述べたあと, 3 節で *i*BDD について述べる. 次に 4 節で *i*BDD が非明示的に表現する BDD の節点に割り当てる符号の符号化法をいくつか提案し, 続く 5 節でベンチマークに対する実験結果を示す. 最後に 6 節で結論を述べる.

2 BDD とゼロサプレス BDD

2.1 二分決定グラフ

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

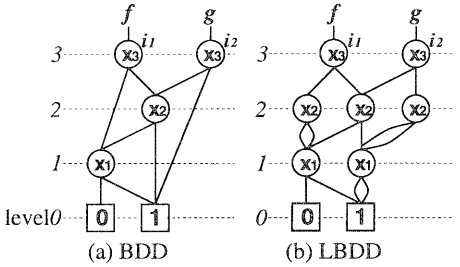


図 1: 二分決定グラフ

$f = x_3x_2 + x_1$, $g = x_3 + x_2 + x_1$ を表現する BDD である.

n 変数論理関数を表現する 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 (0 節点, 1 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.

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

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** と呼び, それぞれその変数節点に割り振られた変数の割当が 0, 1 の時にたどる枝である. 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

$$l(v) > l(e(v,0)) \text{ かつ } l(v) > l(e(v,1))$$

が成り立つものである. 図 1(a), (b) は共に OBDD である. 以下では, 特に断らない限り OBDD を単に BDD と呼ぶ.

BDD 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 f_v は次のように定義される.

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

BDD のうち, 全ての変数節点 v について

$$l(e(v,0)) = l(v) - 1 \text{ かつ } l(e(v,1)) = l(v) - 1$$

が成り立つもの, すなわち, 枝にレベルの飛び越しがない BDD を**levelized BDD** (以下 LBDD と略す) という. 図 1(b) は (a) の BDD と同じ論理関数を表す LBDD である. LBDD を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 *LBDD* L は 4 つ組 $L = (\vec{N}, \vec{e}, c, \vec{i})$ である. ただし,

- $\vec{N} = (N_0, \cdots, N_n)$ は節点集合のベクトルであり, N_j はレベル j の節点の集合である.
- $\vec{e} = (e_1, \cdots, e_n)$ は枝を表す関数のベクトル. $e_j : N_j \times \mathcal{B} \to N_{j-1}$ はレベル j の節点から $j-1$ の節点への枝を表し, $e_j(v,x)$ はレベル j の節点 v の x 枝に接続する節点を表す.
- $c : N_0 \to \mathcal{B}$ は定数節点に割り当てられている論理値である.
- $\vec{i} = (i_1, \cdots, i_m)$ は初期節点のベクトル.

BDD において節点 u, v が次を満たすとき, u と v は**等価**であるという.

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

また, 節点 v が次を満たすとき, v は **冗長** であるという.

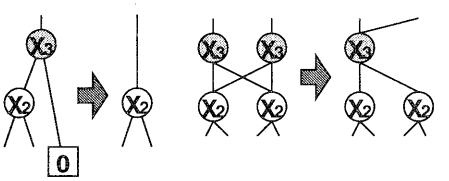
$$e(v,0) = e(v,1) \qquad \cdots v \text{ が変数節点の場合.}$$

BDD の等価な節点と冗長な節点を全て削除することを **BDD の既約化**といい, この結果得られる BDD を **既約な BDD** という. 任意の論理関数 f に対して, f を表す既約な BDD は, 変数の順序を固定すると一意に定まることが知られている [2]. 既約な BDD に対し, レベルの飛び越しがなくなるように冗長な節点を加えると **既約な LBDD** が得られる. 既約な BDD と同様に, 任意の論理関数 f に対し, f を表す既約な LBDD は, 変数の順序を固定すると一意に定まる.

2.2 ゼロサプレス BDD

ゼロサプレス BDD (Zero-Supressed BDD; 以下, ZBDD と略す) は論理関数よりも組合せ集合の表現に優れた BDD の派生の一つであり, BDD の既約化規則を次の様に変更したものである.

1. 1 枝が 0 節点に接続している節点を削除する (図 2(a)),
2. 等価な節点は一方を削除する (図 2(b)).



(a) 1 枝が 0 節点に接続する節点の削除　　(b) 等価な節点の削除

図 2: ゼロサプレス BDD の既約化

この既約化のもとである論理組合せを表現する場合, 変数割り当てが 0 であるような変数の節点は 1 枝が 0 節点に接続することと等価であるのでその節点は削除される. すなわち, 変数割当が 1 であるような変数の節点のみで表現できるということになる. このことは, 組合せ集合を表現する場合に BDD による表現ではケアセット全ての変数毎に節点が必要となるのに対して ZBDD による表現ではケアセットが必要以上に大きくても必要最小限の節点で表現できることを表している.

3 つの組合せ集合 $\{01,10\}, \{001,010\}, \{0001,0010\}$ を同時に表現する BDD と ZBDD の例を図 3 に示す.

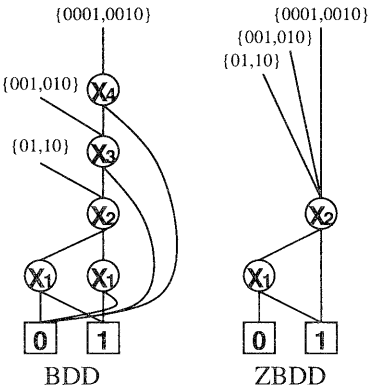


図 3: BDD と ZBDD

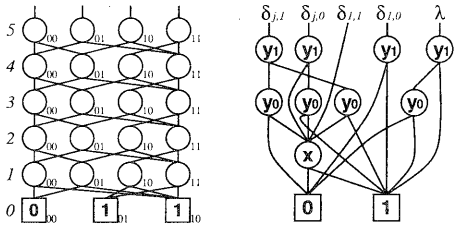
3 二分決定グラフの非明示的表現

二分決定グラフの非明示的表現 (iBDD) は, LBDD を非明示的に表現する手法である. 本節では, その表現法と操作法について述べる.

3.1 表現法

3.1.1 LBDD の非明示的表現

iBDD は LBDD の全ての節点に対して 2 進符号化を施し, その符号を用いて枝の接続関係を論理関数で表し, この論理関数を ZBDD で表したものである. 図 4 に LBDD と



(a) LBDD (b) 非明示的表現

図 4: LBDD とその非明示的表現

その LBDD を非明示的に表現したときの ZBDD を示す.

図 4(a) の LBDD 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. 節点の符号長は符号化手法によって異なる. また, レベル毎に符号長が異なっても構わない. 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, この関係は次のように表すことができる.

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

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次式になる.

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

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. $1-0$ のレベル間の接続関係は次式で表される.

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

定数節点 00, 01, 10 の論理値はそれぞれ 0, 1, 1 であるから, 定数節点のラベルから論理値を得る 2 入力 1 出力の論理関数 λ は次式で表される.

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

このように LBDD 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 ZBDD で表現したもの (図 4(b)) が iBDD である.

3.1.2 iBDD の内部表現

iBDD の内部では, 非明示的に表現している LBDD の各節点の ID を関数化した特徴関数を用いて処理が行われる. これらの特徴関数は粗な集合を表現していることが多いため, このような集合の表現に優れてている ZBDD を iBDD の内部表現として用いる.

ZBDD を用いた場合, レベル毎に符号長が異なっていても上位の不要な変数節点の 1 枝は 0 節点に接続することになり, ZBDD ではこのような節点は削除されるので結果的に符号長を考慮する必要が無くなる. このことは, BDD を用いた場合に発生する, 記憶量の無駄がなくなるだけでなく, 符号長の変化への対応, 複数回の特徴関数の生成の必要もなくなることを意味する.

また, 節点数の削減が達成されると演算の手数も減少し, 内部表現での演算時間の短縮も記憶量の削減とともに期待できる.

3.1.3 複数の論理関数表現

複数の論理関数を iBDD で表現する場合,

1. 複数の論理関数を表現する iBDD を一つだけ用いる share 方式,
2. 一つの論理関数を表現する iBDD を複数個用いる split 方式,

の二通りの取り扱い方が考えられる. split 方式では, 節点の符号長が小さくなる, 複数の論理関数間で符号の共有ができるという利点がある一方で, 多くの論理関数を扱う場合には iBDD を実現するための情報を個々の論理関数毎に保持するために記憶量のオーバーヘッドが大きくなるという欠点がある. share 方式では, 記憶量のオーバーヘッドは小さくて済むが, 節点の符号長が大きくなる. iBDD においては, 膨大な量の論理関数を取り扱うことを念頭においているために個々の論理関数表現のオーバーヘッドは無視できないことから, 本稿では share 方式を採用する.

3.1.4 否定枝

iBDD が表現する LBDD の節点数を削減することにより, それぞれの節点に割り付ける符号の符号長を短くすることができる. 符号長が短くなれば, 接続関係を表す論理関数の変数の数を少なくすることができ, iBDD を構成する ZBDD の節点数を削減することができる.

そこで, レベル毎に各レベルに属する節点が否定枝を持つかどうかを表す論理関数を導入する. 否定枝とは, 初期節点から定数節点へ至るパス上に通過した否定枝の数だけ論理を反転するという属性枝である [11]. 否定枝を無秩序に使用すると表現の一意性が失われるため, 次のような

規則を設ければよいことが知られている.

- 定数節点は 0 節点のみとし, 1 節点は 0 節点の否定とする.
- 0 枝には否定枝を用いない.

すなわち, それぞれの節点は, 1 枝が否定枝であるか否かの二通りしかないことになる. よって, 節点の符号を入力とし, その節点が否定枝を持っている時には 1, 持っていない場合には 0 を出力する論理関数によって否定枝を表現することができる.

同様に, 論理反転枝など他の属性枝を導入し, 更なる節点数の削減を図ることも有効であると考えられる.

3.1.5 *i*BDD 表現の定式化

以上より, 符号化 σ のもとで n 変数 m 出力論理関数を表す iBDD 表現を次のように定義し直す. ただし, レベル j の節点の符号長を w_j, 符号化を $\sigma_j : N_j \to \mathcal{B}^{w_j}$ とする.

定義 3.1 n 変数 m 出力の *LBDD* $L = (\vec{N}, \vec{e}, c, \vec{i})$ を表す *iBDD* I は 4 つ組 $I = (\vec{\delta}, \vec{\nu}, \lambda, \vec{s})$ である. ただし,

- $\vec{\delta} = (\delta_1, \cdots, \delta_n)$, $\delta_j : \mathcal{B}^{w_j} \times \mathcal{B} \to \mathcal{B}^{w_{j-1}}$ はレベル j の節点の枝が接続する節点の符号であり, $\delta_j(\sigma_j(v), x) = \sigma_{j-1}(e_j(v, x))$ を満たす.
- $\vec{\nu} = (\nu_1, \cdots, \nu_n)$, $\nu_j : \mathcal{B}^{w_j} \to \mathcal{B}$ はレベル j の節点の 1 枝が否定枝か否かを表し, 節点 v の 1 枝が否定枝であるとき, $\nu_j(\sigma_j(v)) = 1$ となる.
- $\lambda : \mathcal{B}^{w_0} \to \mathcal{B}$ は定数節点に割り当てられている論理値であり, $\lambda(\sigma_0(v)) = c(v)$ を満たす.
- $\vec{s} = (s_1, \cdots, s_m)$, $s_j = (l_j, \tilde{s}_j)$ は初期節点であり, $l_j = l(i_j)$, $\tilde{s} = \sigma_{l_j}(i_j)$ を満たす.

iBDD が表現する論理関数とは, iBDD が非明示的に表現している LBDD が表現する論理関数を意味する. また, iBDD を構成する論理関数群を表現する多出力 ZBDD のサイズを ***i*BDD 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(a) のように, レベル間の接続関係が類似している, あるいは等しい場合には, この LBDD を表す iBDD を構成する ZBDD 内で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.

3.1.6 *i*BDD の一意性

BDD では, 変数の順序が固定され, かつ, 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る. iBDD 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 *iBDD* I と I' は, 次の条件が満たされるとき一致する.

1) I と I' は同じ変数順で既約な *LBDD* を表現する.
2) I と I' は同じ節点符号のもとで *LBDD* を表現する.
3) どの節点も表さない符号に対する関数の値が, I と I' において等しい.

1) は I と I' が表現する LBDD の形が等しくなる条件である. 2) のもとでは, 表現する LBDD の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 と I' 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をすべて 0 と決めれば 3) は満たされる.

3.2 演算手法

iBDD 同士の二項演算は, BDD における演算手法と同様の, 初期節点から再帰的に演算する手法により実現できる. また, 既に生成されている節点を複数回生成するといった無駄な操作を防ぐために必要な節点キャッシュ, および同じ演算を繰り返さないように, 過去に行った演算結果を保持する演算キャッシュも BDD と同様に導入する.

3.2.1 再帰的な演算手法

iBDD 同士の二項演算の手法として BDD の演算手法と同様な, 初期節点から枝を順にたどって演算を行う再帰的な方法を用いる.

これは iBDD が表現している LBDD の各節点が, それぞれ一つの論理関数を表現しており, 演算 $f \circ g$ が, 次の Shannon の展開式として表現できることに基づいている.

$$f \circ g = \bar{v} \cdot (f|_{v=0} \circ g|_{v=0}) \vee v \cdot (f|_{v=1} \circ g|_{v=1})$$

3.2.2 節点キャッシュ

演算を再帰的に行うときに発生する節点と等価な節点が既に iBDD 中に存在するとき, 新規に節点を生成すると, 論理関数の一意的な表現を維持できない. そこで, 既に存在する節点を節点キャッシュに登録しておき, 新たに節点を発生させるときにその節点キャッシュを参照し, 等価な節点を複数生成させないようにする必要がある. 節点キャッシュは, 次の 2 種類の論理関数によって実現される.

- $NodeCacheExist_j : \mathcal{B}^{w_{j-1}} \times \mathcal{B}^{w_{j-1}} \times \mathcal{B} \to \mathcal{B}$
  0 枝, 1 枝に接続する節点の符号と否定枝の有無からそのような特徴を持つ節点がレベル j に存在するか否かを表す **節点キャッシュ存在関数**.

- $NodeCacheID_j : \mathcal{B}^{w_{j-1}} \times \mathcal{B}^{w_{j-1}} \times \mathcal{B} \to \mathcal{B}^{w_j}$
  0 枝, 1 枝に接続する節点の符号と, 否定枝の有無からそのような特徴を持つレベル j の節点の符号を返す **節点キャッシュ符号関数**.

さらに, これらのレベルごとの節点キャッシュの論理和を構築し iBDD 全体の節点キャッシュを導入する. これにより, 同じ接続関係を持つ他のレベルの節点についても同じ符号が割り当てられ, 接続関係の相似性が高まり, それを表す BDD が複数のレベルで共有されて, 全体の表現のサイズが小さくなることが期待できる.

3.2.3 演算キャッシュ

過去に演算を行い, 結果が既に分かっているものについて演算を繰り返さないようにすることによって, 表現している LBDD の節点数に比例する手数で演算を行うことができる. この目的のために演算キャッシュを導入する. 演算キャッシュは, 過去に演算を行った二項演算を演算キャッシュに登録しておき, 次に同じ演算を行うときは, その演算結果を取り出す事により, 同じ演算を繰り返さないようにするものである. 演算キャッシュは, 以下の 4 種類の論理関数を用いて実現できる. ただし, c は演算の種類を表す符号の符号長とする.

$$
\begin{aligned}
&ApplyCacheExist: && \mathcal{B}^{w_{l(f)}} \times \mathcal{B}^{w_{l(g)}} \times \mathcal{B}^{c} \to \mathcal{B} \\
&ApplyCacheID_j: && \mathcal{B}^{w_{l(f)}} \times \mathcal{B}^{w_{l(g)}} \times \mathcal{B}^{c} \to \mathcal{B}^{w_j} \\
&ApplyCacheLevel_j: && \mathcal{B}^{w_{l(f)}} \times \mathcal{B}^{w_{l(g)}} \times \mathcal{B}^{c} \to \mathcal{N} \\
&ApplyCacheNeg_j: && \mathcal{B}^{w_{l(f)}} \times \mathcal{B}^{w_{l(g)}} \times \mathcal{B}^{c} \to \mathcal{B}
\end{aligned}
$$

3.2.4 演算のアルゴリズム

以上より, 2 つの論理関数 f と g の演算 $\circ$ に関する二項演算のアルゴリズムは以下の通りである. ただし, アルゴリズム中の $GetNode$ は節点キャッシュを参照し, 枝の接続が同じ節点が既に存在すればその節点を返し, なければ新規に作成してその節点を返す処理を表す.

1. 演算キャッシュ中に該当する演算がある場合, その演算結果を返す.
2. 演算キャッシュ中に該当する演算がない場合
   (a) f または g が定数のとき, 演算子の種類に応じた処理を行う.
   (b) f, g いづれも定数でないとき,
      i. f と g の初期節点が同じレベルのとき
         A. $h_0 = f_0 \circ g_0$; $h_1 = f_1 \circ g_1$;
         B. $h = GetNode(h_0, h_1)$;
      ii. f のレベルが g より上位のとき
         A. $h_0 = f_0 \circ g$; $h_1 = f_1 \circ g$;
         B. $h = GetNode(h_0, h_1)$;
      iii. g のレベルが f より上位のとき, f と g を入れ替えて ii. と同様に処理する.

4 iBDD の節点符号化手法

iBDD の内部では, 非明示的に表現する BDD の節点符号を元にその関係を関数化した特徴関数を用いてほとんどの処理が行われることは既に述べた通りである. このことは, 非明示的に表現する BDD の節点符号をどのように与えるかにより, iBDD の領域的, 時間的効率が大きく左右されることを意味する.

本節では, 各節点に付与する符号化の手法を提案する.

4.1 順次符号

本手法は iBDD の構築時に出現した順に小さい符号を割り当てる手法である.

直前に割り当てた符号を記憶しておけば, 次に割り当てる符号はほとんど計算することなく求めることができ, 符号化の際のオーバーヘッドは提案手法の中では最小である.

4.2 枝優先符号

新たな節点に符号を割り当てる際に, その節点の一方の枝に割り当てられている符号をそのままその節点の符号とする手法である. 符号を割り当てる節点のどちらの枝をの節点符号を用いるかにより, 0 枝優先, 1 枝優先, 小符号枝優先, 大符号枝優先が考えられる.

0 枝優先, 1 枝優先 符号を割り当てる節点の 0 枝 (1 枝) に接続している節点の符号をそのままその節点の符号として割り当てる. この手法では, 0 枝 (1 枝) の接続を表現する論理関数が他の論理関数と共有され, 全体的な表現のサイズが低く抑えられることが期待される.

小符号枝優先, 大符号枝優先 符号を割り当てる節点の枝に接続している節点の符号を数値比較して小さい (大きい) 符号をそのままその節点の符号として割り当てる. 小符号枝優先を用いれば符号長が不必要に長くなる可能性を低下すると期待できる.

5 実験結果および考察

提案した各種の符号化手法の有効性を確かめるために MCNC および ISCAS85 のベンチマーク回路を対象にしてそれらの論理関数を表現し, 表現に要する内部表現の ZBDD の節点数および表現を得るのに要した時間を計測した.

実験は, Sun Ultra10 Model 440 (512MB) 上で, 実験用に作成した評価パッケージを用いてベンチマーク回路の回路記述から直接 *i*BDD を構築した. *i*BDD の内部表現で用いる ZBDD については Somenzi により開発, 提供されている CUDD パッケージを用いた [12].

表 1 および表 2 に各ベンチマーク回路に対する実験結果を示す. 表 1 は左から順に回路名, 入力数, 出力数, 各符号化手法を用いた場合の内部表現の ZBDD の節点数である. また, 表 2 については各符号化手法を用いた場合の構築に要した時間 (秒) も掲載している. 時間の計測には `time` コマンドを用いた.

表最下行は, それぞれのベンチマーク回路における節点数の比 (対 順次符号) の平均値である. ただし, MCNC ベンチマーク回路に対する結果については表に掲載していない回路を含めたベンチマーク回路全てにおける平均, ISCAS85 ベンチマーク回路に対する結果については表に掲載した回路のみにおける平均となっている.

MCNC ベンチマーク回路に対する結果では, 符号化法による節点数の変化はあまり見られない. この原因としては, MCNC ベンチマーク回路の規模がいずれも小規模であるために符号の効果が出にくいためであると考えられる.

ISCAS85 ベンチマーク回路に対する結果では, 符号化法により節点数の変化が見られる. 特に小符号枝優先符号では順次符号に対して 1 ～ 2 割程度の節点数の削減が達成されている. 小符号枝優先符号では符号長が不必要に長くならないだけでなく, 異なるレベルに既に存在する符号を利用することで, *i*BDD の内部表現における節点存在関数や枝関数などとの共有が起りやすいためであると考えられる.

実験結果より, 提案した符号化法では順次符号および小符号枝優先符号の効率が他より良い傾向にあることが認められる. 順次符号は符号化法としてはもっとも簡単なものの一つであるが, この手法の有効性は符号長を短く抑えることができる点であると考えられる. また, 小符号枝優先符号は *i*BDD のレベル間の相関を利用して表現のサイズを小さくするという性質に沿った符号化法であるといえるため, 比較的良好な結果が得られたと考えられる.

表 1: 実験結果 (MCNC ベンチマーク)

回路	in	out	節点数			
			順次	0 枝	小符号	大符号
181	14	8	2,652	3,179	2,956	3,228
5xp1	7	10	332	332	327	338
add6	12	7	1,143	1,264	1,285	1,290
adr4	8	5	202	215	214	215
alu1	12	8	122	121	119	113
alu2	10	8	479	493	497	514
alu3	10	8	502	512	513	511
apla	10	12	676	643	688	672
bw	5	28	457	446	462	449
clip	9	5	935	926	951	916
con1	7	2	75	72	74	74
dc1	4	7	78	81	77	81
dc2	8	7	256	242	246	253
dist	8	5	686	696	694	672
dk17	10	11	466	464	467	467
dk27	8	9	130	127	133	125
f51m	8	8	115	122	115	133
misex1	8	7	137	129	134	134
mlp4	8	8	696	701	693	683
radd	8	5	306	298	298	302
rd53	5	3	63	64	63	70
rd73	7	3	131	129	122	119
rd84	8	4	189	176	174	187
risk	8	31	241	240	241	240
root	8	5	219	217	220	215
rot8	8	5	209	206	204	209
sao2	10	4	638	657	662	661
sex	9	14	269	287	267	288
sqn	7	3	270	262	263	263
sqr6	6	11	237	225	230	228
wim	4	7	59	58	59	58
z4	7	4	179	180	175	175
z5xp1	7	10	136	140	130	152
z9sym	9	1	134	107	123	110
比			1.000	0.999	0.998	1.011

表 2: 実験結果 (ISCAS85 ベンチマーク)

回路	入力	出力	順次		0 枝		小符号		大符号	
			節点数	時間	節点数	時間	節点数	時間	節点数	時間
c17	5	2	33	0	33	0	33	0	34	0
c432	36	7	10,275	93	12,432	113	10,886	113	11,681	98
c499	41	32	93,270	504	88,393	658	81,072	675	97,292	604
c1355	41	32	125,050	1,842	116,257	4,006	100,675	4,079	126,162	3,693
c1908	33	25	82,303	2,461	80,619	6,183	74,504	10,460	77,815	5,984
比			1.000		1.014		0.928		1.033	

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の削減を図る論理関数の表現手法である *i*BDD の表現に要する記憶量を大きく左右する, 非明示的に表現する BDD の節点に割り振る符号の符号化法をいくつか提案した.

MCNC および ISCAS85 のベンチマーク回路に対して行った実験により, 節点数の変動は数パーセントにとどまるものの, 符号化法により演算に要する時間が大幅に変動することが確かめられた.

今後の課題としては, 今回の結果を踏まえて効率の良い符号化手法の開発が挙げられる. また, 非明示的に表現する論理関数表現として BDD だけではなく ZBDD にも対応させること, および更なる記憶量の削減, 演算時間の短縮を図ることが挙げられる.

謝辞

本研究を進めるにあたり, 御討論頂きました岡山県立大学情報工学部情報通信工学科高橋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] H. Yamauchi, N. Ishiura and H. Takahashi: "Implicit representation and manipulation of binary decision diagrams," *IEICE Trans. on Fundamentals*, vol. E79-A, no. 3, pp. 354–362 (Mar. 1996).

[6] 山内 仁, 高橋浩光: "二分決定グラフの非明示的表現の効率的な演算手法," 信学技報, VLD97-88, pp. 69–76 (Oct. 1997).

[7] 山内 仁, 高橋浩光: "ZBDD による二分決定グラフの非明示的表現," 信学技報, VLD98-36, pp. 39–46 (July 1998).

[8] S. Minato: "Zero-Suppressed BDDs for Set Manipulation in Combinatorial Problems," *Proc. ACM/IEEE 30th DAC*, pp. 272–277 (June 1993).

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

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

[11] 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).

[12] F. Somenzi: `http://vlsi.colorado.edu/~fabio/`.