設 計 自 動 化 60－19
（1991．12．13）

# 二分決定グラフからの
# 非冗長積和論理の高速生成手法

湊 真一
NTT LSI研究所

あらまし　論理関数の効率的表現手法の1つである二分決定グラフ (BDD: Binary Decision Diagrams) から、非冗長積和論理を高速に生成する手法を提案する。本手法は、Morreale のアルゴリズムを BDD 向けに改良・整理したものである。従来の積和論理の最小化・簡単化アルゴリズムは、いずれも真理値表や積項集合の操作に基づいているのに対し、本手法は BDD の操作により、直接、積和論理を生成するため処理効率がよく、従来手法では適用が困難であった大規模な論理関数に対しても、現実的時間内にコンパクトな積和論理を生成することができる。最後に、本手法により非冗長積和論理の統計的性質を調べた実験結果を示す。

# Fast Generation of Irredundant Sum-of-Products from Binary Decision Diagrams

Shin-ichi MINATO
NTT LSI Laboratories
3-1, Morinosato Wakamiya, Atsugi-shi, Kanagawa Pref., Japan

Abstract　We present a fast method for generating irredundant sum-of-products (ISOPs) from Binary Decision Diagrams (BDDs), which are very efficient representation of Boolean functions. Our method is based on Morreale's algorithm, and we adapted it to BDD manipulation. In our method, we generate ISOPs from BDDs *directly*, while conventional sum-of-products reduction algorithms commonly manipulate redundant cube sets or truth tables. The method enables us to generate compact PLAs from large scale circuits, some of which no previous method managed to flatten into PLAs. Finally, we show the statistical properties of ISOPs with some experimental results.

## 1 はじめに

論理関数を計算機上で効率よく表現し、高速に演算を行うことは、論理合成や検証、テスト生成等の論理設計システムを実現するための重要な基盤技術である。従来、真理値表や積項集合による表現法が多く用いられてきたが、Akers[1]、Bryant[2]らが提案した二分決定グラフ（BDD: Binary Decision Diagrams）による表現法が、近年盛んに研究されている。

二分決定グラフ（以下、単にBDDと呼ぶ）による処理は、記憶効率や処理速度の面で優れており、これまでよりも大規模な回路を現実的に扱うことができる。従来、真理値表や積項集合の操作を用いていたアルゴリズムを、BDDによる処理に置き換えることによって、大幅な処理効率向上が得られる場合があり、いくつかの有効な応用例が報告されている[3][4]。

一方、積和論理式（2段論理、キューブ集合、PLAとも呼ばれる）による表現も、実際の論理設計システムでは多用されている。積項集合の操作に関するアルゴリズムは、古くから多くの研究がなされており、本質的に積和論理式が必要とされる場合もあるため、今後も使用され続けるものと考えられる。

一般に、積和論理式からBDDを生成することは容易である。すなわち、一般の組合せ回路と同様に、BDD同士の論理演算を繰り返せばよい。これに対して、BDDから積和論理式を生成する方法としては、かなり冗長な積和論理式を生成する自明な手法しか知られていなかった。

本稿では、BDDから、非冗長な積和論理式を高速に生成する手法について述べる。本手法はMorreale の提案した *recursive operator*[5] をもとにして、BDD向けに改良・整理したものである。従来、積和論理式の最小化・簡単化については、MINI, ESPRESSO[6]等、数多くの手法が知られているが、いずれも真理値表や積項集合の操作により冗長な部分を削除していくものである。これに対して本手法は、

- BDDから直接、コンパクトな積和論理式を生成する。途中で冗長な積項を生成しない。
- ルールベースではなくアルゴリズム的手法である。同じ論理関数に対して結果は一意に決まる。
- 多出力の積和論理式も扱える。

等の特長をもつ。

本手法を実現し、実験を行った結果、従来手法よりもはるかに高速であることが示された。従来手法では適用が困難であった大規模な論理関数に対しても、現実的な時間内に結果を得ることができる。結果の素子数については最小とは限らないが、最小に近い値を得ることができる。

以下の本文では、まず2章でBDDとその処理について簡単に説明する。次に3章で非冗長積和論理式とその性質について述べる。4章でBDDから非冗長積和論理を生成するアルゴリズムについて述べ、5章では、本手法の実験結果を示し、評価を行う。最後に6章で、本手法により非冗長積和論理式の統計的性質を調べた実験の結果を示す。

## 2 BDDによる論理関数の処理

まず、BDDについて簡単に説明し、他の表現法との変換手法について述べる。

### 2.1 BDD

BDDは、図1に示すような論理関数の有向グラフによる表現である。これは、Shannonの展開を再帰的に繰り返すことによって得られる二分決定木のグラフに対して、冗長なノードの削除と、同形なサブグラフの共有を可能な限り行ったものである。このとき、展開する変数の順序を固定すれば、論理関数を一意に表現することができる。（詳細は[2]を参照）

BDDを計算機上で扱う際に、複数のBDDの間においても互いにサブグラフを共有させ、処理中に現れる全BDDを、1つのグラフで管理することによって、処理効率を一層向上させることができる（図2）。これを共有二分決定グラフ(SBDD: Shared BDD)と呼んでいる[7]。さらに、論理の否定等の機能を表す属性エッジ[7]を用いることにより、演算時間や使用記憶量を削減することができる。

本稿で使用しているBDD処理系では、上記のような効率化手法を採用している。処理系の主な特長は、以下の通りである。

- BDD生成後は、論理の一致判定が定数時間で行える。
- 否定演算が定数時間で行える。
- BDD同士の論理演算が、グラフのサイズにほぼ比例する時間で行える。
- 記憶効率がよく、多くの実用的な論理関数を扱える。

BDDは、変数の順序づけによってサイズが著しく変化するという問題がある。常に最適な順序を見つけることは難しいが、発見的手法により、多くの場合、比較的よい順序を得ることができる[8][7][9]。

### 2.2 他の表現法との変換

組合せ回路からBDDを生成することは、BDDの基本的な演算を適用することにより、機械的に行うことができる。すなわち、回路の入力側からBDD同士のAND/OR演算を繰り返して内部信号線の関数を次々に求めていくことにより、最終的に外部出力を含む全信号線の論理関数をBDDで表すことができる。このときの処理時間はグラフのサイズにほぼ比例する。回路の規模や性質、入力変数の順序によっては、BDDのサイズが爆発的に増大し、実行不能となる場合もある。

積和論理式からBDDを生成する方法も、組合せ回路から生成する場合と同様である。多くの場合、積和形よりもBDDの方がコンパクトであるため、積和論理式からBDDを生成することは比較的容易である。また、回路の段数が小さい方が、回路情報を用いた変数順序づけの発見的手法が的中しやすいと考えられる。

一方、BDDからコンパクトな組合せ回路を生成することは、一般には難しい。現在までに次のような手法が提案されている。

- セレクタによる多段回路への変換[1]
- ブール行列乗算に基づく樹枝状多段回路の合成[10]
- '1'パス列挙による積和論理の生成[1]

セレクタによる多段回路とは、BDDの各ノードを2-to-1 data selectorに置き換えることによって得られる自明な回路である（図3）。素子数はBDDのサイズに比例し、段数は入力数に比例する。ブール行列乗算による方法は、BDDを各レベルで水平に分割し、部分回路を樹枝状に組合せたもので、素子数は一般にはかなり多くなるが、段数は入力数の対数となる。これらの手法による多段論理の合成結果は、もとのBDDのサイズに大きく依存する。コンパクトなBDDで表現可能な論理関数のクラス[11]に対しては効果的であるが、一般の制御系論理に対しては、まだ実用的レベルに達しているとは言えない。

BDDから積和論理式を生成する方法としては、'1'パスを列挙する方法が知られている。これは、BDDのrootノードから'1'の値の終

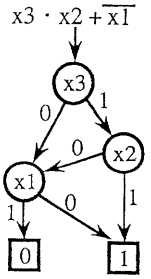



図1：二分決定グラフ (BDD)

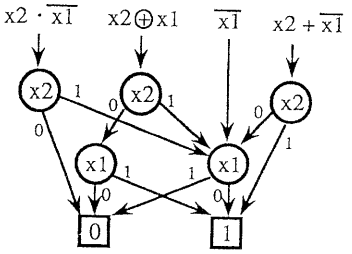
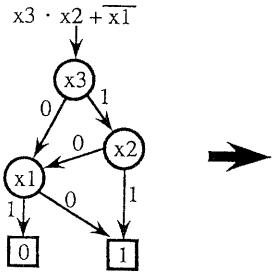



図2：共有二分決定グラフ (SBDD)



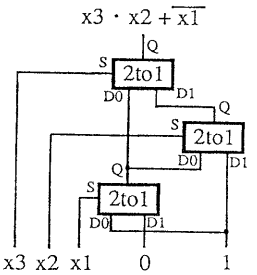




図3：セレクタ回路への変換

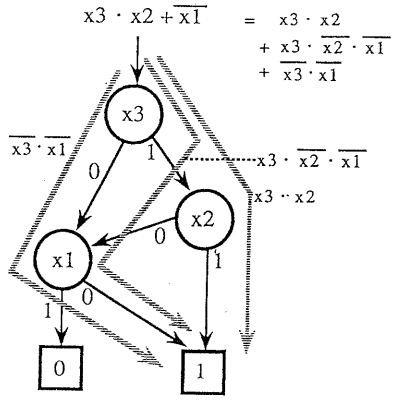



図4：1-パス列挙による積和論理式

端ノードに至るパスをすべて列挙し、各パスを通るための入力値の組合せを積項で表したものである（図4）。生成結果の素子数は、各パス上のノード数の総和に比例する。この方法によって得られた積項集合は、すべて disjoint（被覆が重なっていない）という性質がある。BDDでは、冗長なノードが削除されているので、そのノードに相当する入力変数は積項から取り除かれている。つまり、ある程度の簡単化は行われている。しかし、全てのパスは必ず root ノードを通るので、最上位の入力変数はどの積項にも存在し、取り除かれることがない。このことからもわかるように、パス列挙法で得られた積和論理式には、一般には、かなりの冗長性が残されている。

本稿では、BDDから直接、非冗長な積和論理式を生成する手法を提案する。

## 3 非冗長積和論理

非冗長積和論理について説明し、その性質について述べる。

### 3.1 積和論理式

以下の本文では、論理積 (AND) 演算を $x \cdot y$、論理和 (OR) 演算を $x + y$、否定 (NOT) 演算を $\bar{x}$ のように表記する。「$\cdot$」は通常は省略する。変数、およびその否定をリテラルと呼ぶ。特に、否定のないものを正リテラル、否定のついたものを負リテラルと呼ぶ。

本稿で扱う積和論理とは、AND-OR 2段（正確には NOT-AND-OR 3段）の形、すなわち、積項（キューブ）の和で表された論理式を言う。これは、キューブ集合、2段論理、ＰＬＡ形式と呼ばれる場合もある。積和論理の規模は、通常、積項数やリテラル数で評価される。

一般に、積和論理では、ある1つの論理関数に対して様々な形をとることができ、積項数やリテラル数も大きく変化する。そのため、冗長な積項やリテラルを取り除く最小化・簡単化処理が必要となる。積和論理の簡単化・最小化については、Quine-McCluskey[12] に始まり、MINI[13]、ESPRESSO[6]、McBOOLE[14] など、古くから多くのアルゴリズムが開発されており、現在では、実用的な規模の論理式（積項数1000程度）の簡単化が可能となっている。しかし、これらの手法はいずれも、真理値表や積項集合の操作により解を求めるものであり、BDD の操作に単純に置き換えられるものではない。

### 3.2 非冗長積和論理

非冗長積和論理 (ISOP: Irredundant Sum-of-Products Forms) とは、次のような条件を満たす積和論理式のことをいう。

- 各積項がすべて素項 (prime implicant) であること。すなわち、どのリテラルを取り除いても、異なる論理関数となる。
- 冗長な積項が存在しないこと。つまり、どの1つの積項を取り除いても、異なる論理関数となる。

例えば、$xyz + \bar{x}\bar{y}$ という式は、非冗長である。しかし、$xyz + x\bar{y}$ は非冗長ではない。なぜならば、積項 $xyz$ からリテラル $y$ を取り除いて

も論理が変わらないからである。

非冗長積和論理は、非常にコンパクトな形であるが、積項数やリテラル数が最小になるとは限らない。例えば、次の3つの式は、すべて同じ論理関数を表し、しかもどれも非冗長である。

$$x\bar{y} + xz + \bar{x}y + \bar{x}\bar{z}$$

$$x\bar{y} + \bar{x}y + yz + \bar{y}\bar{z}$$

$$x\bar{y} + \bar{x}\bar{z} + yz$$

この例で明らかなように、非冗長積和論理は1つの論理関数に対して1通りとは限らず、積項・リテラル数も異なる場合がある。ただし、積項・リテラル数が最小となる積和論理式は、必ず非冗長な形になっている。

テスト容易性の観点からみると、非冗長積和論理は、全ての単一縮退故障が検出可能 (fully single stack-at fault testable) であるという、好ましい性質を持つ。なぜならば、ORゲート入力やANDゲート入出力のstack-at-0故障は、積項を1つ取り除くことと等価であり、ANDゲート入力のstack-at-1故障は、リテラルを1つ取り除くことと等価である。非冗長積和論理は、これらの故障により必ず論理が変化するから、その故障を検出する入力の組合せが存在する。

## 4 BDDからの非冗長積和論理の生成アルゴリズム

BDDから非冗長積和論理を生成するアルゴリズムと、その実現方法について述べる。

### 4.1 Morrealeのアルゴリズム

本手法は、Morrealeが提案した *recursive operator* [5] のアルゴリズムをもとにしている。Morrealeのアルゴリズムは、与えられた積和論理式から冗長な項・リテラルを削除していくもので、その基本的アイディアは、次の式で説明される。

$$isop = v \cdot isop_1 + \bar{v} \cdot isop_0 + isop_*$$

ただし、$isop$ は非冗長積和論理を表す。$v$ は入力変数中の任意の1個である。このとき、以下のような考え方で処理を行う。

1. 非冗長積和論理は、$v$ を含む積項集合と $\bar{v}$ を含む積項集合、および $v, \bar{v}$ を含まない積項集合に分けられる。これらの積項集合からリテラル $v$ を取り除いた $isop_1, isop_0, isop_*$ は、それぞれがやはり非冗長積和論理となるはずである。

2. $isop_1$ と $isop_0$ に共通部分があれば、それは $isop_*$ によりカバーできるはずなので、その共通部分は *don't care* として、$isop_1$ と $isop_0$ を再帰的に求める。

3. その結果として、$isop_1$ と $isop_0$ に共通部分が残っている場合は、もはや $isop_*$ でカバーする必要がないので、その部分を *don't care* として $isop_*$ を再帰的に求める。

なお、展開していく変数の順序を固定すれば、1つの論理関数に対して結果は一意に定まる。

Morrealeのアルゴリズムは、計算中の部分関数や *Don't care* 関数をすべて積和論理式で表し、積項集合の操作によって論理演算を行なっていたため、否定演算や恒真性判定などに手間がかかり、処理効率が悪かった。しかし基本的には、変数に値を代入して展開していく再帰的アルゴリズムであるため、BDDの操作との整合性がよい。本稿の手法は、このアルゴリズムをBDD向けに改良・整理したものである。

### 4.2 BDD向けアルゴリズム

本手法では、積項集合からではなく、BDDから直接、非冗長積和論理を生成する。図5にそのアルゴリズムを示す。アルゴリズムの動作を図6の実行例について説明する。

(a) まず、BDDで最上位にある入力変数 $v$ に0,1を代入して、論理関数 $f$ を2つの部分関数 $f_0, f_1$ に分ける。

(b) $f_0, f_1$ の共有部分を *don't care* とした $f_0', f_1'$ を作る。$f_0', f_1'$ は、$\bar{v}, v$ を含む積項でなければカバーできない部分を表している。それらの非冗長積和論理 $isop_0, isop_1$ を再帰的に求める。

(c) $f_0, f_1$ のうち、すでに $isop_0, isop_1$ にカバーされた部分を *don't care* とした $f_0'', f_1''$ を作る。

(d) $f_0'', f_1''$ の共有部分（$\bar{v}, v$ を含まない積項でカバーされる部分）を表す $f_*$ を作り、その非冗長積和論理 $isop_*$ を再帰的に求める。

以上で求めた $\bar{v} \cdot isop_0, v \cdot isop_1, isop_*$ の総和が $f$ の非冗長積和論理となる。

図6では、わかりやすくするためカルノー図で示しているが、実際にはBDDで表現し、操作している。

本アルゴリズムは、以下の点で、BDDの操作に非常によく適合している。

- BDDで最上位の変数に0,1を代入することにより、グラフのエッジを1回たどるだけで容易に部分関数が得られる。
- BDDでは冗長なノードは削除されているので、論理式に現れない変数に対する不要な展開は自動的に省かれる。
- 再帰的に展開していく途中に、恒真・恒偽関数が現れた場合には、即座に検出して展開をやめ、結果を返すことができる。これに対して積項集合を用いた場合では、論理式を見ただけでは恒真・恒偽性が判定できず、さらに代入・展開を行わなくてはならない場合がある。

本アルゴリズムの計算時間は、厳密に評価することは難しいが、5章に示す実験結果によれば、おおよそ、BDDのサイズと、計算結果のリテラル数の積に比例すると考えられる。

### 4.3 アルゴリズムの効率的な実現手法

図5のアルゴリズムでは、理解しやすいように、$isop_0, isop_1, isop_*$ は論理式として記述した。しかし、これをそのまま計算機上に実現すると、$isop$ を求めるために論理式の操作が必要となる。我々の実現では、論理式をプログラム内部で操作するのではなく、結果を直接、ファイルに出力する方法をとっている。すなわち、リテラルを記憶するスタックを1つ用意しておき、再帰呼び出しごとに、着目しているリテラルをスタックに積んでおく。そして恒真関数を検出したときに、その時点のスタックの内容を、積項としてファイルに（追加して）出力して行く。このようにすれば、計算結果の積和論理式が大規模であっても、主記憶上ではBDDのデータを扱うだけですみ、記憶効率がよい。

また、本アルゴリズムでは、*don't care* を含む3値の論理関数を扱う。BDD処理系で *don't care* を扱うには、次の2つの方法がある [7]。

- BDDの終端ノードを、「0」,「1」,「*」の3値とした処理系を用いる。
- 3値を2ビットに符号化し、2つの2値BDDの組として表す。

```
ISOP(f(x)) {
/* 入力 f(x): {0,1}^n → {0,1,*} */
/* 出力 isop: 非冗長積和論理式 */
  if ( ∀x ∈ {0,1}^n; f(x) ≠ 1 ) { isop ← 0; }
  else if ( ∀x ∈ {0,1}^n; f(x) ≠ 0 ) { isop ← 1; }
  else {
    v ← one of x ; /*v は BDD で最上位の変数 */
    f0 ← f(x|v=0) ; /*v = 0 のときの subfunction */
    f1 ← f(x|v=1) ; /*v = 1 のときの subfunction */
    次の演算規則により f0', f1' を求める。;
```

| $f_0'$: $f_1$ \ $f_0$ | 0 | 1 | * |
|---|---|---|---|
| 0 | 0 | 1 | * |
| 1 | 0 | * | * |
| * | 0 | * | * |

| $f_1'$: $f_1$ \ $f_0$ | 0 | 1 | * |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 1 | 1 | * | * |
| * | * | * | * |

```
    isop0 ← ISOP(f0') ; /* v̄ を含む項を再帰的に求める */
    isop1 ← ISOP(f1') ; /* v を含む項を再帰的に求める */
    isop0, isop1 が表す論理関数を g0, g1 とする。;
    次の演算規則により f0'', f1'' を求める。;
```

| $f_0''$: $g_0$ \ $f_0$ | 0 | 1 | * |
|---|---|---|---|
| 0 | 0 | 1 | * |
| 1 | − | * | * |

| $f_1''$: $g_1$ \ $f_1$ | 0 | 1 | * |
|---|---|---|---|
| 0 | 0 | 1 | * |
| 1 | − | * | * |

```
    次の演算規則により f* を求める。;
```

| $f_*$: $f_1''$ \ $f_0''$ | 0 | 1 | * |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 |
| * | 0 | 1 | * |

```
    isop* ← ISOP(f*) ;
      /* v̄, v を含まない項を再帰的に求める */
    isop ← v̄ · isop0 + v · isop1 + isop* ;
  }
  return isop ;
}
```

図5：非冗長積和論理生成アルゴリズム

上記2つの方法のいずれが効率がよいかは、扱う論理関数の性質にも依存し、一概には決められない。ただし、3値の処理系を使用した場合、図5に記述されているような特殊な3値論理演算を行うためには、処理系内部で演算規則を定義しなければならない。我々の実現では、処理系内部の変更を避けて、2値BDDの組で表現する方法をとった。2ビットの符号化は、次のように $(\lfloor f \rfloor, \lceil f \rceil)$ に分解して処理している。

| $f$ | $\lfloor f \rfloor$ | $\lceil f \rceil$ |
|---|---|---|
| 0 | 0 | 0 |
| 1 | 1 | 1 |
| * | 0 | 1 |

この符号化によれば、*don't care* を含む恒真判定は $\lceil f \rceil$ の恒真判定で、*don't care* を含む恒偽判定は $\lfloor f \rfloor$ の恒偽判定で、それぞれ即座に判定できる。また、特殊な3値論理演算は、$\lfloor f \rfloor, \lceil f \rceil$ に関する2値の論理演算に分解して処理することができる。例えば、次のような演算規則：

| $f_0'$: $f_1$ \ $f_0$ | 0 | 1 | * |
|---|---|---|---|
| 0 | 0 | 1 | * |
| 1 | 0 | * | * |
| * | 0 | * | * |

を符号化すると、

$$(\lfloor f_0' \rfloor, \lceil f_0' \rceil) = (\lfloor f_0 \rfloor \cdot \overline{\lceil f_1 \rceil}, \lceil f_0 \rceil)$$

のように、2値の論理演算の組合せで実現できる。

### 4.4 多出力関数の処理

以上で説明したアルゴリズムは、1出力の論理関数を対象としたものであったが、これを多出力に拡張することができる。多出力の場合は、

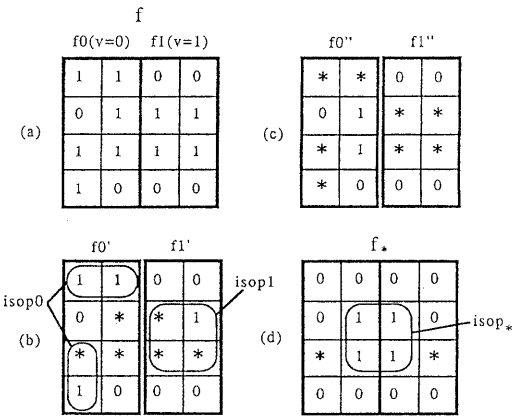



図6：アルゴリズム実行例

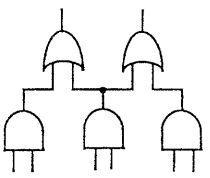


図7：多出力積和論理

複数の出力に同時に含まれる積項が存在するので、1出力ずつ逐次処理した場合よりもコンパクトに表現できる（図7）。

我々の実現では、逐次処理したのちに共通積項をマージするのではなく、各出力の非冗長積和論理を同時に生成するという方法をとっている。つまり、図5のアルゴリズム中の $f$ を多出力化して、各出力関数を表すBDDの配列として保持し、その各要素について1出力の場合と同様に処理を行う。0,1を代入する変数は、全出力のBDDの中で最上位のものを1つ選び、その変数で全出力を並行して展開していく。展開を再帰的に繰返していくと、最終的に恒真・恒偽の混在する多出力定数となる。そのうち恒真となっている出力が、その項を含むということになるので、前節で述べたスタックに積まれている積項を、該当する出力のOR入力に接続する。

## 5 実験と評価

以上で述べたアルゴリズムを実現し、実験と評価を行った。計算機は、SPARC station 2 (SunOS 4.1.1, 32MByte) を用いた。プログラムはCおよびC++で記述した。

表1：ESPRESSO との比較

| 関数 | 入力 | 出力 | 本手法 | | | ESPRESSO | | |
|---|---|---|---|---|---|---|---|---|
| | | | 積項 | リテラル | (秒) | 積項 | リテラル | (秒) |
| dec8 | 12 | 2 | 17 | 90 | 0.3 | 17 | 90 | 0.2 |
| enc8 | 9 | 4 | 17 | 56 | 0.2 | 15 | 51 | 0.3 |
| add4 | 9 | 5 | 135 | 819 | 0.7 | 135 | 819 | 1.9 |
| add8 | 17 | 9 | 2519 | 24211 | 13.3 | 2519 | 24211 | 443.1 |
| mult4 | 8 | 8 | 145 | 945 | 1.4 | 130 | 874 | 5.0 |
| mult6 | 12 | 12 | 2284 | 22274 | 26.7 | 1893 | 19340 | 1126.2 |
| achil8p | 24 | 1 | 8 | 32 | 0.2 | 8 | 32 | 2.0 |
| achil8n | 24 | 1 | 6561 | 59049 | 8.7 | 6561 | 59049 | 3512.7 |
| 5xp1 | 7 | 10 | 72 | 366 | 0.8 | 65 | 347 | 1.5 |
| 9sym | 9 | 1 | 148 | 1036 | 0.9 | 87 | 609 | 10.7 |
| alupla | 25 | 5 | 2155 | 26734 | 20.5 | 2144 | 26632 | 257.3 |
| bw | 5 | 28 | 68 | 374 | 1.1 | 22 | 429 | 1.4 |
| duke2 | 22 | 29 | 126 | 1296 | 3.2 | 87 | 1036 | 28.8 |
| rd53 | 5 | 3 | 35 | 192 | 0.3 | 31 | 175 | 0.5 |
| rd73 | 7 | 3 | 147 | 1024 | 1.2 | 127 | 903 | 4.2 |
| sao2 | 10 | 4 | 76 | 575 | 1.1 | 58 | 495 | 2.4 |
| vg2 | 25 | 8 | 110 | 914 | 1.9 | 110 | 914 | 42.8 |
| c432 | 36 | 7 | 84235 | 969037 | 1744.8 | × | × | >36k |
| c880 | 60 | 26 | 114299 | 1986014 | 1096.6 | × | × | >36k |

表2：変数順序の影響

| 関数 | 動的重みづけ法 | | | | 乱数順序 | | | |
|---|---|---|---|---|---|---|---|---|
| | ノード | 積項 | リテラル | (秒) | ノード | 積項 | リテラル | (秒) |
| dec8 | 16 | 17 | 90 | 0.3 | 41 | 17 | 90 | 0.3 |
| enc8 | 21 | 17 | 56 | 0.2 | 25 | 17 | 56 | 0.2 |
| add8 | 41 | 2519 | 24211 | 13.3 | 383 | 2519 | 24211 | 24.3 |
| mult6 | 1274 | 2284 | 22274 | 26.7 | 1897 | 2354 | 22963 | 30.2 |
| achil8n | 24 | 6561 | 59049 | 8.7 | 771 | 6561 | 59049 | 30.9 |
| 5xp1 | 43 | 72 | 366 | 0.8 | 60 | 72 | 364 | 0.9 |
| alupla | 1376 | 2155 | 26734 | 20.4 | 4309 | 2155 | 26730 | 43.1 |
| bw | 85 | 68 | 374 | 1.1 | 90 | 64 | 353 | 1.1 |
| duke2 | 396 | 126 | 1296 | 3.2 | 609 | 125 | 1280 | 3.7 |
| sao2 | 143 | 76 | 575 | 1.1 | 133 | 76 | 571 | 1.0 |
| vg2 | 108 | 110 | 914 | 1.9 | 1037 | 110 | 914 | 2.7 |

### 5.1 ESPRESSO との比較

本手法の性能を評価するため、実用的な組合せ回路の出力関数（一般に多段・多出力）を BDD で表現し、これから非冗長積和論理を生成したときの積項・リテラル数、および計算時間を調べた。BDD の入力変数の順序づけは動的重みづけ法 [7] を用いている。計算時間は、BDD の順序づけ・生成にかかった時間も含んでいる。

一方、比較対象として、misII[15] 上で多段回路を 2 段化したのち、ESPRESSO により簡単化を行った結果を調べた。（計算時間は 2 段化にかかった時間も含む。）

実験結果を表 1 に示す。実験に用いた回路は、8-bit データセレクタ (dec8)、8-bit エンコーダ (enc8)、4+4 bit 加算器 (add4)、8+8 bit 加算器 (add8)、2×2 bit 乗算器 (mult4)、3×3 bit 乗算器 (mult6)、24 入力アキレス腱関数 [6](achil8p) とその否定 (achil8n)、そしてその他は MCNC の論理合成用ベンチマークの中から抜粋したものである。

実験の結果をみると、本手法は、実行速度については ESPRESSO に比べて圧倒的に高速であり、大規模な例になるほど顕著である。特に C432, C880 の例では、積項数 10 万、リテラル数 100 万を超える大規模な積和論理式を、現実的時間内に生成することができた。一方、ESPRESSO を用いた場合では、10 時間経過しても 2 段化が完了しないため、簡単化を適用することができなかった。また「アキレス腱関数」は否定をとると積項数が著しく増加する性質があるため、ESPRESSO では極端に時間がかかっている。これに対して、BDD を用いると否定形でも同じ記憶量で表現できるので、積和論理式の生成も非常に効率よく実行できることがわかる。

表3：入力数による変化（出力数 =1）

| 入力 | ノード | 積項 | リテラル | リテラル / 積項 |
|---|---|---|---|---|
| 1 | 0.58 | 0.77 | 1.35 | 1.75 |
| 2 | 1.41 | 1.25 | 2.84 | 2.27 |
| 3 | 3.22 | 2.30 | 7.17 | 3.12 |
| 4 | 6.39 | 4.20 | 16.05 | 3.82 |
| 5 | 11.71 | 7.85 | 36.39 | 4.64 |
| 6 | 20.51 | 14.88 | 82.18 | 5.52 |
| 7 | 36.24 | 27.09 | 172.06 | 6.35 |
| 8 | 64.59 | 52.27 | 377.41 | 7.22 |
| 9 | 118.17 | 99.31 | 808.09 | 8.14 |
| 10 | 210.12 | 192.26 | 1738.89 | 9.04 |
| 11 | 365.04 | 370.90 | 3693.49 | 9.96 |
| 12 | 633.97 | 722.11 | 7865.91 | 10.89 |
| 13 | 1144.12 | 1406.31 | 16635.79 | 11.83 |
| 14 | 2154.49 | 2752.53 | 35154.84 | 12.77 |
| 15 | 4151.45 | 5393.25 | 73980.57 | 13.72 |

表4：出力数による変化（入力数 =10）

| 出力 | ノード | 積項 | リテラル | リテラル / 積項 |
|---|---|---|---|---|
| 1 | 209.80 | 192.13 | 1737.84 | 9.05 |
| 2 | 364.44 | 381.69 | 3452.20 | 9.04 |
| 3 | 500.86 | 568.10 | 5145.01 | 9.06 |
| 4 | 630.93 | 754.88 | 6842.25 | 9.06 |
| 5 | 758.33 | 933.86 | 8468.70 | 9.07 |
| 6 | 884.87 | 1120.83 | 10166.36 | 9.07 |
| 7 | 1011.08 | 1294.84 | 11750.90 | 9.08 |
| 8 | 1136.94 | 1471.63 | 13355.59 | 9.08 |
| 9 | 1262.29 | 1649.47 | 14978.33 | 9.08 |
| 10 | 1388.76 | 1815.44 | 16493.02 | 9.08 |
| 11 | 1513.15 | 1987.56 | 18078.64 | 9.10 |

生成結果の積項・リテラル数を ESPRESSO の場合と比較すると、関数によっては一致するものもあるが、一般にはやや多くなる。ただし、「非冗長」という性質はかなり強い制約を与えているらしく、ESPRESSO との差は多くの場合、0 ～ 20% 程度に留まっている。これまでの実験の限りでは、さほど大きな差（リテラル数が 2 倍以上）を生じる例は見つかっていない。

### 5.2 入力変数の展開順序の影響

次に、変数の順序づけによる影響を調べた。一般に BDD のサイズは、変数の順序によって大きく変化する。そこで、BDD のサイズが小さくなるような順序づけ（動的重みづけ法 [7]）を行った場合と、BDD のサイズが大きくなるような順序（乱数順）を与えた場合とで、本手法による計算結果の積項・リテラル数、および計算時間を比較した。その

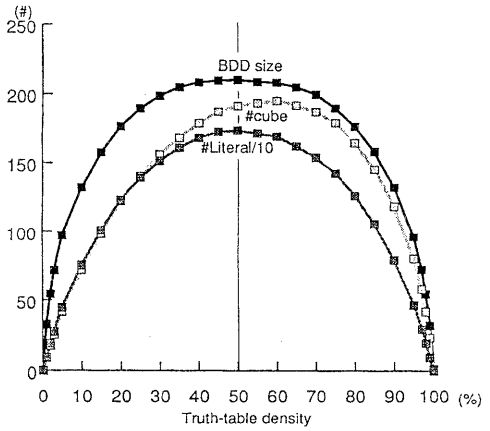



図8：真理値表濃度による変化（入力数＝10, 出力数＝1）

結果、表2に示すように、変数の順序によって、BDDのサイズは著しく変化するのに対し、積項・リテラル数はほとんど変化しないことがわかる。「非冗長」という性質があるため、変数の展開順序によらず、安定したサイズの論理式が得られている。ただし、計算時間はBDDのサイズに左右されるので、変数の順序づけはやはり重要である。

それから、表には示していないが、同じ関数でも変数の順序によって非冗長積和論理の形がいくつか存在する場合、BDDで上位の変数のリテラルが論理式中に少なく、下位のリテラルがやや多く現れる、という傾向が観察された。

## 6 非冗長積和論理の統計的性質

本手法の高速性を活用して、非冗長積和論理の統計的な性質を明らかにする実験を行った。実験は、任意の入力パタンに対してランダムな値を返す（つまり真理値表が乱数表となる）論理関数を100通り生成して、それぞれに本手法を適用した。そして各関数について、BDDのノード数、非冗長積和論理の積項・リテラル数を調べ、100回の平均値を求めた。なお、乱数はCの標準ライブラリ関数を用いて発生させた。

### 6.1 入力・出力数に関する性質

入力・出力数に対するデータ量の増加傾向を調べるため、

- 出力数を1に固定し、入力数を動かしたときの変化（表3）
- 入力数を10に固定し、出力数を動かしたときの変化（表4）

の2つの実験を行なった。

表3をみると、入力数が大きくなるにしたがって、BDDノード数、積項数ともに指数関数的に増大している。10変数まではBDDノード数よりも積項数の方が少ないが、それ以降はBDDノード数の方が少ない。BDDのノード数については、入力数$n$に対して最悪$O(2^n/n)$となることが知られている[1]が、乱数関数に対する統計的な実験でも、これに添った値が得られている。一方、積項数については、実験結果からみて、ほぼ$O(2^n)$となっているようである。積項数とリテラル数の関係では、積項1個あたりのリテラル数が、入力数$n$にほぼ比例していることが観測できる。

表4では、入力数固定で、出力数を増やした場合の様子がわかる。BDD、非冗長積和論理ともに、各出力間でデータの共有を行なっているので、その効果が現れて、1出力のときの記憶量の$m$倍（mは出力数）よりも小さい値となっている。ただし、本実験で用いた乱数関数では、各出力の間に全く相関関係がない。実際にLSI設計で使われる論理回路では、なんらかの相関がある場合が多いので、本実験結果よりも出力数に対する記憶量の増大は少ないと思われる。なお、入力数が固定なので、積項1個あたりのリテラル数はほとんど一定となっている。

### 6.2 真理値表濃度に関する性質

次に、真理値表濃度（真理値表中の1の割合）を変化させたときの様子を調べた。図8は、10入力1出力の論理関数について、重みつき乱数を発生させて真理値表濃度を0%から100%まで動かしたときの各データ量の変化を示したものである。BDDのサイズは、50%を中心に左右対称なグラフとなり、情報理論で用いられる「情報のエントロピー」と似た形となる。これは、BDDが論理関数を効率よく表現していることを示している。

次に積項数をみると、左右対称とはならず、60%付近で最大となっている。ここで、同じ実験を和積形で行なった場合を考えると、元の論理関数の否定の関数の積和形を求めることと等価であるから、真理値表の0,1の濃度が逆転し、積和形のグラフを50%の線を境に左右反転したグラフとなる。すなわち、和積形では40%付近で和項数最大となる。これは、統計的には、真理値表濃度50%以上のとき、積和形よりも和積形の方がコンパクトとなる可能性が高いことを示している。ただし、あくまで平均の話であり、個別の論理関数については、必ずしも当てはまらない。例えば、

$$ab + cd + ef$$

という積和論理は、真理値表濃度が約58%であるが、和積形に直すと、

$$(a+c+e)(a+c+f)(a+d+e)(a+d+f)$$
$$\cdot(b+c+e)(b+c+f)(b+d+e)(b+d+f)$$

となり、積和形よりも項数が多くなる。

一方、リテラル数はほとんど左右対称となっており、50%付近で最大となっている。つまり、積和形でも和積形でも、統計的にはリテラル数に差はない。一般に、多段論理合成系では、積和論理の複雑度評価の目安としては、積項数よりもリテラル数の方がよいといわれている。本実験結果から、統計的には、リテラル数が論理関数の情報量を比較的よく反映していると言える。

## 7 おわりに

本稿では、BDDから直接、非冗長積和論理を生成する手法について述べた。実験の結果、本手法により、コンパクトな積和論理式を高速に生成できることが示された。特に、これまで生成できなかった非常に大規模な積和論理式を、非冗長な形で得ることができた。あまりに大規模な積和論理式を生成しても、その後の処理ができないという問題もあるが、少なくとも、今まで不明だった積和論理式の規模を、現実に見積もることができるという意義は大きい。

現在のVLSI CAD/DAシステムでは、積和論理を直接・間接的に扱うアルゴリズムが多く見られる。積和論理の高速生成手法、あるいはBDDから積和論理へのデータ変換手法として、本手法の応用範囲は広いと考えられる。今後は、本手法の論理合成系への適用を考えたい。

## 謝辞

本稿を執筆するにあたり、御指導頂いたNTT LSI研究所 安達主幹研究員、遠藤主任研究員に感謝致します。本研究は、著者が京都大学在学中の研究を発展させたものであり、京都大学の矢島脩三教授、大阪大学の石浦菜岐佐講師を始め、御助言・御討論頂いた諸氏に感謝致します。

## 参考文献

[1] S. B. Akers: "Binary Decision Diagrams", *IEEE Trans. Comput.*, pp. 509-516 (1978).

[2] R. E. Bryant: "Graph-Based Algorithms for Boolean Function Manipulation", *IEEE Trans. Comput.*, pp. 677-691 (1986).

[3] 松永, 藤田: "順序付き2分決定グラフと許容関数を用いた多段論理回路簡単化手法", 信学論, Vol. J74-A, No. 2, pp. 196-205, (1991).

[4] S. Minato, N. Ishiura and S. Yajima: "Fast Tautology Checking Using Shared Binary Decision Diagram - Benchmark Results -", *Proc. IFIP International Workshop on Applied Formal Methods for Correct VLSI Design*, pp. 580-584, (1989).

[5] E. Morreale: "Recursive Operators for Prime Implicant and Irredundant Normal Form Determination" *IEEE Trans. Comput.*, pp. 504-509, (1970).

[6] Brayton, McMullen, Hachtel, and S.-Vincentelli: "Logic Minimization Algorithms for VLSI Synthesis", *Kluwer Academic Publishers USA*, (1984).

[7] 湊, 石浦, 矢島: "論理関数の共有二分決定グラフによる表現とその効率的処理手法", 情処論, Vol. 32, No. 1, pp. 77-85 (1991).

[8] 藤田, 藤沢, 松永, 角田: "2分決定グラフのための変数順決定アルゴリズムとその評価", 情処論, Vol. 31, No. 4, pp. 532-541, (1990).

[9] 湊 真一: "共有二分決定グラフの「幅」に着目した変数の順序づけ手法", 第42回情処全大, pp. 6.158-159, (1991).

[10] 石浦菜岐佐: "二分決定グラフからの組合せ論理回路の合成" 第43回情処全大, pp. 1-105, 106, (1991).

[11] S. Yajima, N. Ishiura: "A Class of Logic Functions Expressible by a Polynomial-Size Binary Decision Diagrams", *Proc. of Synthesis and Simulation Meeting and Int. Interchange (SASIMI'90)*, (1990).

[12] E. J. McCluskey: "Minimization of Boolean Functions", *The Bell System Technical Jounal, Vol. 35, pp. 1417-1444*, (1956).

[13] S. J. Hong, R. G. Cain and D. L. Ostapko: "MINI: A Heuristic Approach for Logic Minimization", *IBM Jounal of Res. and Dev., Vol. 18, No. 5, pp. 443-458*, (1974).

[14] M. R. Dagenais, V. K. Agrawal and N. C. Rumin: "A New Procedure for Exact Logic Minimization" *IEEE Trans. on CAD, Vol. CAD-5, pp. 229-238*, (1986).

[15] R. K. Brayton, R. Rudell, A. S.-Vincentelli and A. R. Wang: "MIS: A Multiple-Level Logic Optimization System" *IEEE Trans. on CAD, Vol. CAD-6, No. 6, pp. 1062-1081*, (1987).