変数シフトの導入による項分岐決定図の省空間化

Variable Shift SDD: A More Succinct and Canonical Sentential Decision Diagram

中村　健吾† 　伝住　周平†† 　西野　正彬†

† NTT コミュニケーション科学基礎研究所　〒 619–0237 京都府相楽郡精華町光台 2-4
†† 東京大学大学院情報理工学系研究科　〒 113–8656 東京都文京区本郷 7-3-1
E-mail: †{kengo.nakamura.dx,masaaki.nishino.uh}@hco.ntt.co.jp, ††denzumi@mist.i.u-tokyo.ac.jp

あらまし　論理関数や組合せ集合の圧縮索引として，二分決定図 (BDD) が広く用いられているが，近年その一般化である項分岐決定図 (SDD) が提案され，圧縮性能の良さと索引としての機能性の高さから注目を集めている．本稿では，SDD をより圧縮して保持しつつ，索引としてのクエリ処理の機能を SDD と同様に併せもつ，変数シフト SDD(VS-SDD) というデータ構造を提案する．VS-SDD は対称性のある論理関数や組合せ集合に対して SDD と比べ圧縮性能が良くなることを理論と実験の両面で確かめた．

キーワード　決定図，論理関数，組合せ集合，集合族，変数シフト，対称性

1 はじめに

アイテム集合が与えられたときに，そのアイテムの選び方の集合，すなわち部分集合の集合を取り扱う場面は数多い．例えば図 1(a) のように，論理回路は出力を *true* にするような入力変数の組合せを全て集めることで，論理関数としてだけでなく部分集合の集合としても表せる．また，図 1(b) のように，所与のグラフに対して特定の条件，例えばマッチングやパス，連結グラフなど，を満たす部分グラフを列挙する問題を考えると，各辺をアイテムだと思えば，辺の組合せのうち条件を満たすもの全てをもつことになるので，やはり集合の集合を扱う．**論理関数**や集合の集合 (**組合せ集合**，**集合族**) の中身を仮に列挙できれば，例えば前者の例では論理回路の設計や動作の検証など，後者の例では配電網の損失最小化やネットワーク信頼性の計算などに応用できる．しかし，論理関数や集合族を扱う際には，しばしば**組合せ爆発**ともよばれる，扱う集合の数が問題サイズに対して指数的に増加する現象に見舞われる．例えば，図 1(b) のようなグリッドグラフにおける対角間パスの列挙では，11×11 グリッドでもパスの数は 10^{24} 個に達する．従って，ただ列挙するだけでは表現が大きくなりすぎるので，このような論理関数や集合族をコンパクトかつ扱いやすい「索引」として保持するための研究が古くから行われている．

その中でも，**二分決定図** (Binary Decision Diagram, *BDD*) [6] は特に有名で，論理関数 [1]の表現形として広く用いられている．BDD は論理関数を有向非巡回グラフ (Directed Acyclic Graph, DAG) で表現する．BDD が広く用いられる理由として，(1) 応用上現れる様々な論理関数や集合族が理論的・経験的に小さく表現できること，(2) 一定の条件下で，同じ論理関数を表現すると全く同じ BDD になること (**表現の一意性**)，

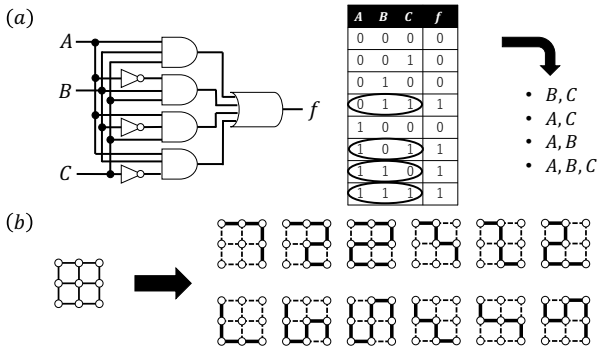

図 1　組合せ集合を扱う例．(a) 論理回路，(b) 部分グラフ構造列挙．

(3) 様々な有用なクエリや操作をサポートすること，が挙げられる．表現の一意性により，各々BDD で表現された 2 つの論理関数が等価であるかのチェックが構造の同一性のチェックで行えて，これは CUDD [26] などの BDD パッケージでは定数時間で行える．また，一度論理関数を BDD で表現すれば，出力を *true* にするような入力変数の組合せの数え上げ (**モデルカウント**) や，他の BDD で表現された論理関数との AND や OR などの演算が，BDD の DAG 構造をほどくことなく行える．近年 BDD は，ネットワークの影響拡散の厳密計算 [17] や，ネットワーク信頼性の近似計算 [25]，厳密最大化 [20] など，ネットワークにまつわる計算と最適化によく使われている．

近年，BDD の一般化として，**項分岐決定図** (Sentential Decision Diagram, *SDD*) [10] が提案され，注目を集めている [14], [23]．SDD は論理関数を表現する際のサイズについて BDD よりもタイトな上限をもち [10]，さらに SDD が BDD よりも指数倍小さくなるような論理関数も存在することが知られている [4]．また，SDD は BDD と同様に一定の条件下で表現の一意性を有し，さらに文献 [11] にあるような標準的なクエリや操作をサポートする．従って，SDD は BDD より小さな表現を実現し，かつ索引としての機能性を BDD と同様に備える．

1：図 1(a) のように，論理関数から等価な集合族に変換できて，逆もまた然りなので，以後は特に断りのない限り「論理関数」と「集合族」を同一視する．

本稿では，SDD をより小さく圧縮して表現する，**変数シフト** *SDD*(Variable Shift SDD, *VS-SDD*) というデータ構造を提案する．VS-SDD は BDD や SDD と同様に一定の条件下で表現の一意性をもち，また文献 [11] に挙げられたクエリや操作のうち SDD がサポートするものは同様にサポートする．また，VS-SDD は同じ論理関数を表す SDD より大きくなることはない．さらに，VS-SDD で表現する方が SDD で表現するよりもサイズが指数倍小さくなるような論理関数が存在することも示した．すなわち，VS-SDD は SDD より小さな表現を実現し，なおかつ索引としての機能性を SDD と同様に備える．VS-SDD は，対称性を備える論理関数や集合族を特に小さく表現することができ，いくつかの対称性があるベンチマーク論理関数を，SDD よりも小さく表現できることを実験的に示した．

類似研究として，より一般的な決定図である FBDD や DDG を論理関数の対称性を取り入れて小さく表現する Sym-FBDD と Sym-DDG [2] がある．これらは変数の置換も扱えるため，変数シフト SDD よりも多くの対称性を扱える．しかし，Sym-FBDD/DDG のもととなる FBDD/DDG には SDD には不可欠な vtree のような構造が無いので，このアイデアを SDD へと直接導入することはできない．また，Sym-FBDD/DDG は conditioning や `Apply` といったいくつかの重要なクエリや操作をサポートしない．

2 項分岐決定図 (SDD)

まず，SDD [10] の構造について導入する．SDD は BDD と同様に，論理関数を DAG として表現するデータ構造である．ここで以後の説明のため，表記のルールを導入する．まず，大文字アルファベットで 1 つの変数，小文字アルファベットでその中身の値 (*true* もしくは *false*) を表す．また，太字の大文字アルファベットで複数変数の集合，小文字アルファベットでその中身の値たち (インスタンス化) を表す．

論理関数 f が引数にとる変数たちを $\mathbf{Z}$，その分割を $\mathbf{X}, \mathbf{Y}$ とする．すると，f は $\mathbf{X}$ の変数のみからなる論理関数 $p_1, \ldots, p_n$ と，$\mathbf{Y}$ の変数のみからなる論理関数 $s_1, \ldots, s_n$ を用いて，$f(\mathbf{Z}) = \bigvee_{i=1}^{n} [p_i(\mathbf{X}) \wedge s_i(\mathbf{Y})]$ と分解できる．これを f の $(\mathbf{X}, \mathbf{Y})$-**分解**とよぶ．このとき，n をこの分解の**サイズ**とよぶ．特に，$(\mathbf{X}, \mathbf{Y})$-分解であって，任意の $i \neq j$ に対し $p_i \wedge p_j = \mathit{false}$ かつ，$\bigvee_{i=1}^{n} p_i = \mathit{true}$ かつ，任意の i に対し $p_i \neq \mathit{false}$ (**互いに素**) であるものを，$\mathbf{X}$**-分割**とよぶ．以降，本稿では $\mathbf{X}$-分割を $f = \{(p_1, s_1), \ldots, (p_n, s_n)\}$ のように表現する．このとき，$p_1, \ldots, p_n$ (左側) を**プライム**，$s_1, \ldots, s_n$ (右側) を**サブ**とよぶ．もし $\mathbf{X}$-分割が任意の $i \neq j$ に対し $s_i \neq s_j$ を満たすならば，この分割は *compressed* である，という．

SDD は論理関数を，$\mathbf{X}$-分割を繰り返し適用して分解してゆくことにより表現するデータ構造である．このとき，どのように変数を ($\mathbf{X}$ と $\mathbf{Y}$ に) 再帰的に分割してゆくかを決めるのが，*vtree* とよばれる二分木である．vtree の葉のそれぞれは論理関数に用いられる変数と 1 対 1 対応している．各内部ノードは，左側の部分木にある変数たちを $\mathbf{X}$，右側の部分木にある変数

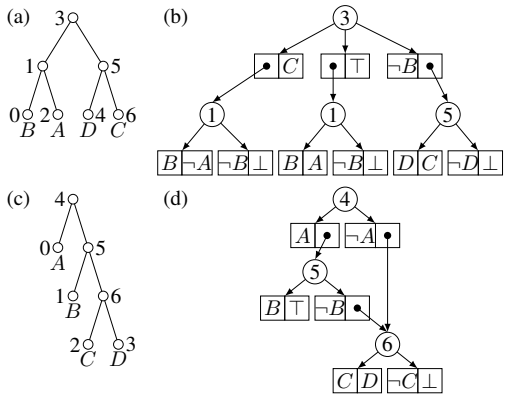

図 2 (a) vtree の例．(b) 関数 $f = (A \wedge B) \vee (B \wedge C) \vee (C \wedge D)$ を表す，(a) の vtree に従う SDD の例．(c) vtree の例．(d) 関数 $f = (A \wedge B) \vee (C \wedge D)$ を表す，(c) の vtree に従う SDD の例．

たちを $\mathbf{Y}$ とする $\mathbf{X}$-分割を表現している．vtree の例を図 2(a) に示す．この vtree は，まず 4 つの変数 $\{A, B, C, D\}$ に対し $\{A, B\}$-分割を適用し，次いで左側の $\{A, B\}$ にさらに $\{B\}$-分割を，右側の $\{C, D\}$ にさらに $\{D\}$-分割を適用することを表す．vtree の各ノードには，図 2(a) のように番号をつけておく．

vtree を用いると，SDD とそれが表す論理関数は次のように再帰的に定義できる．ここで，SDD α に対し，α が表す論理関数を $\langle \alpha \rangle$ と表す．

定義 1 vtree v に従う SDD は，次のいずれかである．

- $\top$ や $\bot$ は，それ自体が SDD である．ここで，$\langle \top \rangle = \mathit{true}$，すなわち恒真関数，$\langle \bot \rangle = \mathit{false}$，すなわち恒偽関数である．これらの SDD を**定数**とよぶ．
- 変数 X に対し，v が変数 X に対応する葉を含むならば，X や $\neg X$ は v に従う SDD である．$\langle X \rangle = X$，$\langle \neg X \rangle = \neg X$ である．これらの SDD を**リテラル**とよぶ．
- $p_1, \ldots, p_n$ は v の左の子ノード以下に従う SDD で，$\langle p_1 \rangle, \ldots, \langle p_n \rangle$ は互いに素，$s_1, \ldots, s_n$ は v の右の子ノード以下に従う SDD とする．このとき，これらを各々ペアにした $\{(p_1, s_1), \ldots, (p_n, s_n)\}$ は v に従う SDD である．この SDD は $[\langle p_1 \rangle \wedge \langle s_1 \rangle] \vee \cdots \vee [\langle p_n \rangle \wedge \langle s_n \rangle]$ という論理関数を表す．このような SDD を**分解ノード**とよぶ．

SDD α の**サイズ**は，α 中で用いられる全ての分割のサイズの和として定義し，$|\alpha|$ で表す．

図 2(b) は，関数 $f = (A \wedge B) \vee (B \wedge C) \vee (C \wedge D)$ を表す，図 2(a) の vtree に従うサイズ 9 の SDD を表す．図中で，丸いノードはそれぞれ分解ノードを表し，中に書いてある数字は，そのノードがどの vtree ノードに従っているかを番号で表している．例えば，図 2(b) の SDD の一番上のノードでは，f は $((\neg A \wedge B) \wedge C) \vee (A \wedge B) \vee (\neg B \wedge (C \wedge D))$ と分解されている．

SDD は次のような操作ができる．論理関数 f に対して，f を *true* にするような，変数への *true* と *false* の割り当て方の総数を f の**モデルカウント**とよぶ．SDD α に対し，α が表す論理関数のモデルカウントを計算するのは $O(|\alpha|)$ 時間でできる．モデルカウントの応用については後述する．次に，論理関

数 f と g を論理和 $\vee$ や論理積 $\wedge$ などの演算で組み合わせること (**Apply 演算**) も，SDD のままで可能である．より具体的には，関数 f を表す SDD α と関数 g を表す SDD β が与えられれば，α と β が同じ vtree に従う限り，$f \vee g$ や $f \wedge g$ などの関数を表す SDD を $O(|\alpha||\beta|)$ 時間で作れる．この操作は論理関数から SDD を作る際に直接使えるほか，集積回路などのシステムの論理設計や検証でも頻繁に用いられる．

また，SDD は次のような制限を加えることで表現の一意性を保証できる．ある SDD のクラス (集合) が**カノニカル**であるとは，vtree が所与であるときに，任意の論理関数 f に対して，f を表す SDD がそのクラス中に唯一つ存在するということである．SDD では 2 つのカノニカルなクラスを考えられる．

定義 2 SDD α は，α 中で用いられる全ての分割が compressed であるとき，*compressed* であるという．α は，$\{(\top, \beta)\}$ や $\{(\beta, \top), (\neg\beta, \bot)\}$ の形の分割をもたないとき，*trimmed* であるといい，一方で $\{(\top, \top)\}$ や $\{(\top, \bot)\}$ の形の分割をもたないときは *lightly trimmed* であるという．α は，全ての分割において，$p_1, \ldots, p_n$ が左の vtree 子ノード，$s_1, \ldots, s_n$ が右の vtree 子ノードにちょうど従うとき，*normalized* であるという．

定理 3 ([10]) *Compressed* かつ *trimmed* な *SDD* はカノニカルである．また，*compressed* かつ *lightly trimmed* かつ *normalized* な *SDD* もカノニカルである．

なお，SDD α が与えられたときに α を trimmed にするのは，$\{(\top, \beta)\}$ や $\{(\beta, \top), (\neg\beta, \bot)\}$ の形の分割を β に置き換える操作をボトムアップに行えばできる．同様にして α を lightly trimmed や normalized にすることもできる．

SDD が小さな表現を実現する理由の一つに，もし分解の途中で全く同じ論理関数が部分関数 (プライムやサブ) として複数出てきたとしても，その論理関数を表す SDD は一つだけ作ればよい，という点が挙げられる．これを**部分構造の共有**とよぶ．図 2(d) の SDD では，番号 4 がついた分解ノードは $C \wedge D$ という論理関数を表しているが，このノードは 2 箇所から参照されている．これは，論理関数を分解する途中で $C \wedge D$ という関数が 2 回出てきたことを意味するが，この場合でも，$C \wedge D$ を表現する分解ノードは一つあればよい．

3 変数シフト項分岐決定図 (VS-SDD)

ここから，本稿で提案する変数シフト項分岐決定図 (VS-SDD) について説明する．VS-SDD のアイデアは，SDD のように全く同じ論理関数が出てきた場合に部分構造の共有を起こすだけでなく，ある特定の種類の変数の置き換えによって同じ論理関数を表現するようになる場合にも部分構造の共有を起こすようにすることである．もしこれが可能ならば，従来の SDD と比べて部分構造の共有を起こせる場所が増えるため，SDD より小さな論理関数の表現を得られる．これを達成するために，従う vtree ノード番号 (図 2(b) や (d) でいうと丸ノードの中に書かれた数字) を差分的に管理する方法を提案する．

まず,「特定の種類の変数の置き換え」を例を用いて述べた後に，正式に定義する．例えば，図 2(b) の SDD では，下側中央の番号 1 がついたノードは $A \wedge B$ という論理関数を，下側右の番号 5 がついたノードは $C \wedge D$ という論理関数を表す．また，図 2(a) の vtree で，1 番の部分木と 5 番の部分木は変数以外は全く同じ構造をしており，B を D，A を C に置き換えることによって互いに行き来できる．ここで，同様の変数の置き換えを $A \wedge B$ という論理関数に対して行うと，$C \wedge D$ となる．さらに，図 2(b) の SDD で下側中央のノード以下の構造に同様の変数の置き換えを行うと，下側右のノード以下の構造そのものが得られる．このような分解ノードの組を一つにまとめることを目指す．ここで正式な定義のためにいくつかの記号を導入する．SDD α とその分解ノード q に対し，$\alpha(q)$ で α の q 以下の部分構造 (SDD) を表す．また，vtree v のノード v_1, v_2 に対し，$v_1 \sim v_2$ で，各々を根とする部分木が葉に付随する変数以外構造が同一であるということを表す．このとき，h_{v_1,v_2} を，v_1 の変数から v_2 の変数への写像であって，v_1 の葉の各変数 X を $h_{v_1,v_2}(X)$ に置換することで v_2 が得られるようなものとする．なお，$\sim$ は vtree ノード間の同値関係になる．

定義 4 vtree v に従う SDD α について，α 中の 2 つの分解ノード q, r は次の条件を満たすとき**合同**であるという．(1) v_q を q が従う vtree，v_r を r が従う vtree とすると $v_q \sim v_r$．(2) $\alpha(r)$ は $\alpha(q)$ 中の各リテラル $X, \neg X$ を $h_{v_q,v_r}(X), \neg h_{v_q,v_r}(X)$ で置き換え，各分解ノードが従う vtree ノード番号も書き換えることにより得られる．

定義 5 vtree v が与えられたとき，使う変数がそれぞれ $B(f), B(g)$ である 2 つの論理関数 f, g は次の条件を満たすとき v **の下で同型**であるという．(1) v_f を，葉の変数が $B(f)$ と一致する v の部分木とする．同様に v_g を定める．すると $v_f \sim v_g$．(2) g は f 中の各変数 X を $h_{v_f,v_g}(X)$ に置き換えることにより得られる．

図 2(b) の SDD で，下側中央のノードと下側右のノードは合同であり，これらが表す論理関数 $A \wedge B$ と $C \wedge D$ は，図 2(a) の vtree の下で同型である．

さて，合同なノードを一つにまとめるためにはどうすればよいだろうか．もとの SDD では，各分解ノードは従う vtree ノード番号を明示的にもち，各リテラルはそのまま保持される．しかし，合同な分解ノードは異なる vtree ノードに従い，それ以下の部分構造中のリテラルも異なるので，もし合同な分解ノードを一つにまとめるのであれば，これらの情報は明示的に保持してはならない．そこで，VS-SDD では (i) vtree ノード番号のつけ直しと (ii) 従う vtree ノード番号の差分管理という 2 段階の方法でこれを解決し，合同なノードを一つにまとめる．

まず (i) について考える．もとの SDD では，vtree ノード番号は任意につけてよいものなので，番号づけをするルールを固定してしまっても SDD の応用範囲が制限されることはない．そこで，ここでは vtree のノードを一定の規則で辿って，その辿った順番に従い 1 から順番に番号をつけることを考える．ここで「一定の規則」とは，次の 3 つの操作を各ノードで決められた順番で行うことをいう．

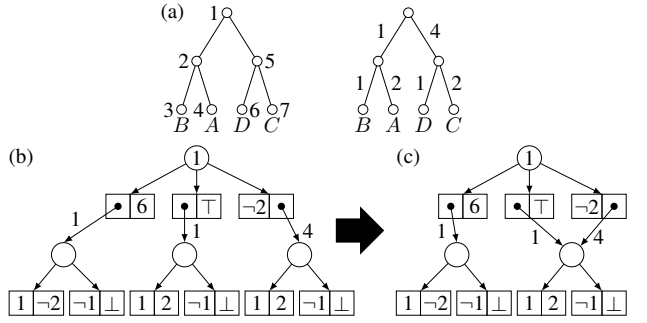

図 3　(a) 図 2(a) の vtree の行きがけ順番号づけと，その差分番号づけ．(b) 図 2(b) の SDD で，vtree ノード番号を差分的に考えた構造．(c) (b) で同一な部分構造を共有したもの．

- (L) 左の子ノードの部分木を再帰的に辿る．
- (R) 右の子ノードの部分木を再帰的に辿る．
- (N) 自分自身のノードを辿る．

特に，各ノードで (N)→(L)→(R) という順番で操作を行って得られる順番を**行きがけ順**とよぶ．例として，図 2(a) の vtree の行きがけ順による番号づけを図 3(a) に示す．以降の説明は行きがけ順を用いて行うが，行きがけ順以外でも「一定の規則」の下で順番づけする限りはどの方法でもよい．この番号づけのポイントは以下の補題にまとめられる．

補題 6 T を n ノードの根付き順序木として，T のノードの「一定の規則」による 1 から n までの番号づけを考える．もし，T の v を根とする部分木と w を根とする部分木がノード番号以外順序木として同一ならば，これらの差分番号づけも同一になる．ここで差分番号づけとは，木の各辺でその両端のノード番号の差をもたせたものである．

同一な構造の部分木は一定の規則の下では同じように辿られるので，この補題が成立する．例えば，図 2(a) の右側は，行きがけ順のノード番号に対して差分番号づけを描いたものであるが，行きがけ順番号 2 の部分木と 5 の部分木では，その差分番号づけは全く同じになる．

続いて (ii) について考える．まず，各分解ノードで従う vtree ノード番号を明示的に保持する代わりに，分解ノードを指す各辺に，指す先の分解ノードが従う vtree ノード番号と，指す元のノードが従う vtree ノード番号の差をもたせる．次に，各リテラルをそのまま保持する代わりに，そのリテラルの変数に対応する葉の vtree ノードの番号と，そのリテラルを分解の一部としてもつ分解ノードが従う vtree ノード番号の差をもたせる．ここで使用する vtree ノード番号は，行きがけ順番号づけによるものである．図 2(b) の SDD に，図 3(a) にある vtree の行きがけ順番号づけを考えた上で，vtree 番号の差分化を行なった構造を図 3(b) に示す．すると，次の補題が成立する．

補題 7 *vtree* ノードが一定の規則により番号づけされているならば，上記の操作により，合同なノードは全く同一な部分構造に変換される．

この補題は合同性の定義 (定義 4) と補題 6 より示せる．例えば，図 3(b) において，下側中央と下側右の分解ノードは元々合同であったが，変換後にはそれ以下の構造も含めて全く同一になっている．こうして生ずる同一な部分構造を共有することによって，ノード数の削減を達成できる．図 3(b) で生じた同一な部分構造を共有することにより得られる構造を図 3(c) に示す．このように，vtree ノード番号の差を考える変換を行って，それにより生じた同一な部分構造を共有することによって得られる構造を VS-SDD とよぶ．なお,「変数シフト」の呼び名は，文献 [19] の *variable shifter* を元にしている．Variable shifter は複数の BDD をまとめて表現する際のサイズを小さくするために考案されたものである．しかし，variable shifter は変数の順番 (番号) に注目するものだが，本稿での変数シフトはより一般に vtree ノード番号に注目している．

VS-SDD を構築するには，もとの SDD から変換する方法と，表したい論理関数から直接構築する方法がある．ここでは前者について説明する．もとの SDD α を VS-SDD へと変換するアルゴリズムを Algorithm 1 に示す．ここで，SDD α に対して，α が従う vtree ノードを $v(\alpha)$ と表し，vtree ノード v に対してその行きがけ順による番号を $\mathtt{num}(v)$ とする．また，vtree ノード w に対し，$e(w)$ を，v のノードで部分木の構造が w の部分木と一致するもののうちノード番号が最も小さいものの番号とする．変数の個数を m とすれば，全ての $e(\cdot)$ の値を $O(m)$ 時間で求められるので，この値は前もって計算しておく．Algorithm 1 の中で，リテラルを変換する処理は 7,8 行目，分解ノードを変換する処理は 9 行目にある．アルゴリズムの内部では `ConvTable` と `UniqTable` という 2 つの連想配列を用いる．`ConvTable` は，入力の SDD α のノードと，出力の VS-SDD のノードとの対応関係を保持する．`UniqTable` は，もしすでに合同な部分構造が処理されていたらその処理結果の VS-SDD を返すような連想配列である．ここで e が，従う vtree ノードの部分木の構造が同一であるか判定するために用いられている．もしまだ合同な部分構造が処理されていなければ (12 行目)，13 行目で `CreateNewNode` で新たな分解ノードの VS-SDD を作る．SDD α に対し，$\mathtt{VS}(\alpha)$ を α の**変数シフト形**とよぶことにする．

命題 8 *SDD* α が与えられると，その変数シフト形は $O(|\alpha|)$ 時間で計算できる．

最後に VS-SDD の基本的な性質をいくつか示す．まず一つは表現の一意性である．実は，Algorithm 1 の `VS` は SDD と VS-SDD の間の全単射の写像であることが示せるので，SDD と VS-SDD の間には一対一関係がある．従って，VS-SDD に対しても，compressed, trimmed, lightly trimmed, normalized の 4 つの性質を，対応する SDD がその性質をもつかどうかで定義できる．すると，カノニカルな SDD のクラスからカノニカルな VS-SDD のクラスを導ける．

定理 9 *Compressed* かつ *trimmed* な *VS-SDD* はカノニカルである．また，*compressed* かつ *lightly trimmed* かつ *normalized* な *VS-SDD* もカノニカルである．

次に VS-SDD のサイズについて，Algorithm 1 の `VS` では，

Algorithm 1 SDD α を VS-SDD に変換する手続き $\texttt{VS}(\alpha)$.

入力：分解ノードの SDD α.
出力：α を VS-SDD へと変換した構造.

```
1: if ConvTable(α) ≠ nil then
2:    return ConvTable(α)
3: else
4:    γ ← {}
5:    for all α 中の要素 (p_i, s_i) do
6:       if p_i が定数 then p ← p_i
7:       else if p_i がリテラル then p ← num(v(p_i)) − num(v(α))
8:       else if p_i が否定リテラル then p ← ¬(num(v(p_i)) − num(v(α)))
9:       else p ← (num(v(p_i)) − num(v(α)), VS(p_i))
10:      (s と s_i に対しても同様の処理を行う)
11:      要素 (p, s) を γ に加える
12:   if UniqTable(e(v(α)), γ) = nil then
13:      UniqTable(e(v(α)), γ) ← CreateNewNode(γ)
14:   return ConvTable(α) ← UniqTable(e(v(α)), γ)
```

もとの SDD で合同なノードをまとめるだけで，サイズが増加することは無いので，$|\texttt{VS}(\alpha)| \leqq |\alpha|$ が成立する．

命題 10 任意の *SDD* α に対して，$|\texttt{VS}(\alpha)| \leqq |\alpha|$.

3.1 指数圧縮

次に注目するのは，変数シフト形に変換することでサイズがどの程度圧縮できるのか，ということである．変数の個数を m とすると，vtree の中で同じ形の部分木は最大でも m 個なので，サイズの比 $|\texttt{VS}(\alpha)|/|\alpha|$ の下限は $1/m$，すなわち最大でも m 倍の圧縮が限界である．一方で，この圧縮率を漸近的にほぼ達成する論理関数を実際に構成できる．

定理 11 論理関数の列 $f_1, f_2, \ldots$ であって，f_k は $O(2^k)$ 変数の論理関数で，f_k を表す *compressed SDD* のサイズは任意の *vtree* に対して $\Omega(2^k)$ であるが，f_k を表す *compressed VS-SDD* のサイズはある *vtree* に対して $O(k)$ になるものが存在する．

本定理における圧縮率は $O(k/2^k) = O(\log m/m)$ である．定理 11 は，SDD に対してあらゆる vtree を考慮しているため，冒頭の議論よりも強い主張になっている．また，定理 11 は，VS-SDD が決定図のサイズを線形サイズから対数サイズに指数的に圧縮するケースがある，ということも表している．

詳しい過程は省略するが，定理 11 の証明では，

$$f_k(\mathbf{X}) = (\neg X_1 \vee \neg X_2) \wedge \bigwedge_{j=1}^{2^k-2} \big((\neg X_j \vee \neg X_{2j+1}) \wedge (\neg X_j \vee \neg X_{2j+2}) \wedge (\neg X_{2j+1} \vee \neg X_{2j+2}) \big),$$

という論理関数列が実際に条件を満たすことを示す．f_k は，実は深さ k の完全二分木の部分グラフであってマッチングを成すような辺集合を考えることに相当する．図 4(a) のような完全二分木を考えると，$f(\mathbf{x}) = \mathit{true}$ となるのは，*true* を割り当てた変数たちに対応する辺集合がマッチングを成すとき，かつそのときに限る．定理 11 の前半は容易に示せる．一方で後半に関しては，図 4(b) にあるような vtree $v_k(\mathbf{X})$ に従う compressed VS-SDD のサイズが実際に $O(k)$ になることを示す．ここで $\mathbf{Y}$ は二分木で X_1 以下の辺に対応する変数 $(X_3, X_4, \ldots)$，$\mathbf{Z}$ は X_2 以下の辺に対応する変数 $(X_5, X_6, \ldots)$ である．このようにして

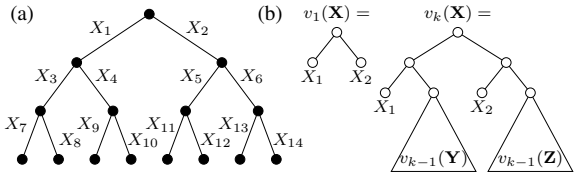

図 4 (a) 各辺に変数が付随した深さ 3 の完全二分木．(b) 証明に用いる vtree $v_k(\mathbf{X})$ の再帰的構造．

作った vtree に対して，その compressed SDD を考えると，その中に指数的に多くの合同な分解ノードのペアが存在することがわかる．VS-SDD にすることでそれらのノードをまとめられるので，全体としてサイズ $O(k)$ を達成できる．

4 データ構造

VS-SDD が空間効率の面でも良くなるためには，理論的性質のみならず実装にも言及しなければならない．なぜなら，理論的な VS-SDD サイズが SDD サイズよりも小さくなったとしても，vtree ノード番号を各分解ノードではなく各辺に差分的にもつことでメモリ使用量がむしろ増加してしまう可能性があるからである．残念ながら，VS-SDD がその通り実装される限りは，VS-SDD の方がメモリ使用量が大きくなる例は存在する．

しかし，多少の変更でこの問題は回避できる．まず，normalized VS-SDD に対しては，辺に付随させる vtree ノード番号の差を無視すればよい．Normalized ならばプライムが左の子ノード，サブが右の子ノードにちょうど従うので，上から辿るだけで従う vtree ノード番号を復元できるためである．一方で，一般の VS-SDD に対してはもとの SDD の再利用が考えられる．変数シフト形に変換する際に行うのは，合同なノードを一つにまとめることである．その代わりに，例えば従う vtree ノード番号が最も小さいものなど，合同なノードのうち代表の 1 つのみを残すという方法も考える．このとき，他の合同なノードは分解 $\{(p_1, s_1), \ldots, (p_n, s_n)\}$ をもつ代わりに，代表のノードへのポインタを張る．分解を直接もつと，たとえ分解のサイズが 1 でも 2 つのポインタを使うため，これを 1 つのポインタに置き換えてもメモリ使用量が増加することはない．なお，このような構造でも VS-SDD の性質が損なわれることはない．なぜならこの構造でも，vtree ノード番号のオフセットを保存しておき，上から順番にノードを辿り，合同なノード間のポインタを辿るときにノード番号の差をオフセットに足せば，従う vtree ノード番号を復元できるからである．

5 クエリと操作

VS-SDD の重要な性質は，合同なノードを解いて SDD へと戻すことなく，SDD ができるのと同様なクエリや操作を実現できるということである．ここでは，特に文献 [11] に記された，論理関数に対する基本的なクエリや操作に焦点を当てる．これらのクエリや操作の基礎となるのは，`Apply` とよばれる操作と，(重みなし) モデルカウントである．

2 章でも述べた，論理関数のモデルカウントを求める問題は，

#SAT ともよばれ，ネットワーク信頼性の推定 [12] など幅広い分野に応用をもつ．SDD が表す論理関数のモデルカウントを求めるためには，各分解ノードでそれ以下の部分構造が表す論理関数のモデルカウントの値をボトムアップに求める動的計画法を行う．ここで，同型な論理関数は同じモデルカウントをもつので，SDD 中で合同な分解ノードが表す部分論理関数もやはり同じモデルカウントをもつ．例えば，図 2 中の合同なノードはどちらもモデルカウントが 1 である．従って，VS-SDD で同様な動的計画法を行う際に，合同なノードが何個まとめられたかに関わらずカウントの値は 1 つだけもてばよい．各カウント値は動的計画法により各々の分解サイズに比例する時間で求まり，全体でも VS-SDD サイズに比例する時間で求まる．

命題 12 *VS-SDD* α が表す論理関数のモデルカウントの値は $O(|\alpha|)$ 時間で求まる．

なお，確率推論 [11], [24] などのいくつかの応用先では，各変数に重みがついた重み付きモデルカウントが必要になる．VS-SDD α に対し重み付きモデルカウントの計算は必ずしも $O(|\alpha|)$ 時間ではできないが，各分解ノードに対してまとめられた合同なノードの個数分だけカウンタを用意することで少なくとも元の SDD を用いるのと同じ時間でできる．さらに，いくつかの変数で重みが同じであれば，カウンタのうちいくつかは共有できて，SDD を用いるより計算が速くなる可能性がある．

次に `Apply` とは，2 つの決定図 (この場合は VS-SDD) α, β と，$\vee$ (論理和) や $\wedge$ (論理積) のような二項演算 $\circ$ を受け取り，$\langle\alpha\rangle \circ \langle\beta\rangle$ を表す決定図を出力する操作である．この操作は，論理関数から直接決定図を構築するために重要であり，2 章で述べた通り，SDD では $O(|\alpha||\beta|)$ 時間でこの操作を実行できる．

VS-SDD に対する `Apply` を考える際に重要なのは，合同性と同型性を使うことにより計算を省略できる場合がある，ということである．例えば $f = A \wedge B$, $f' = C \wedge D$, $g = \neg A$, $g' = \neg C$ に対し，f と f'，g と g' は同型だが，このとき $f \vee g = \neg A \vee B$ と $f' \vee g' = \neg C \vee D$ も同型になる．このように，用いる変数が共通する論理関数の組 $(f, g), (f', g')$ に対し，f と f'，g と g' が同型なら，$f \circ g$ と $f' \circ g'$ や $\neg f$ と $\neg f'$ も各々同型になる．この事実を用いると，証明は少し込み入るが，VS-SDD に対しても $O(|\alpha||\beta|)$ 時間で `Apply` ができることを示せる．

命題 13 同じ *vtree* に従う *2* つの *VS-SDD* α, β が与えられたとき，$\langle\alpha\rangle \vee \langle\beta\rangle$ や $\langle\alpha\rangle \wedge \langle\beta\rangle$ を表す *VS-SDD* を $O(|\alpha||\beta|)$ 時間で構築できる．

重要なのは，VS-SDD における `Apply` の計算は，SDD における `Apply` の計算のうち，同型性を用いることで省略できる部分を省略するような形になるので，SDD の `Apply` より高速にできる，ということである．命題 13 により，CNF で表された論理関数から，$\vee$ や $\wedge$ を順番に組み合わせることで直接 VS-SDD を構築できる．

なお，命題 13 のアルゴリズムでは，例え入力の α, β が compressed であったとしても，出力が compressed である，つまり各分解でサブとして同じものが出てこないとは限らない．出力が compressed であることを強制するためには，計算中に同じサブが出現したら対応するプライムの OR を取って一つにまとめるという操作を加えればよい．しかし，SDD でこのような操作を加えると理論上は `Apply` の計算が $O(|\alpha||\beta|)$ 時間で終わらなくなってしまうことがあり [5]，VS-SDD でも同様の問題が起こる．一方で実用上は，SDD の `Apply` でこのような操作を挟むことでむしろ計算が高速化したり出力サイズが小さくなったりすることが多く [5]，VS-SDD でも同様である．

表 1　SDD (S), compressed SDD (S(C)), VS-SDD (V), compressed VS-SDD (V(C)) の，多項式時間で計算できる (a) クエリと (b) 操作の一覧．✓ は多項式時間アルゴリズムが存在することを，● は多項式時間アルゴリズムは存在し得ないことを示している．各クエリや操作の説明については文献 [11] を参照．

	(a) クエリ	S	V	S(C)	V(C)
CO	consistency	✓	✓	✓	✓
VA	validity	✓	✓	✓	✓
CE	clausal entailment	✓	✓	✓	✓
IM	implicant check	✓	✓	✓	✓
EQ	equivalence check	✓	✓	✓	✓
CT	model counting	✓	✓	✓	✓
SE	sentential entailment	✓	✓	✓	✓
ME	model enumeration	✓	✓	✓	✓
	(b) 操作	S	V	S(C)	V(C)
$\wedge$**C**	conjunction	●	●	●	●
$\wedge$**BC**	bounded conjunction	✓	✓	●	●
$\vee$**C**	disjunction	●	●	●	●
$\vee$**BC**	bounded disjunction	✓	✓	●	●
$\neg$**C**	negation	✓	✓	✓	✓
CD	conditioning	✓	✓	●	●
FO	forgetting	●	●	●	●
SFO	singleton forgetting	✓	✓	●	●

以上のモデルカウントと `Apply` に加えて，いくつかのクエリや操作に対して固有のアルゴリズムを考えることで，表 1 の結果が示される．ポイントは，文献 [11] に示された基本的なクエリや操作で，SDD が多項式時間でサポートするものは全て VS-SDD でも多項式時間でサポートする，ということである．

6 実　　験

最後に，変数シフトを導入することで決定図のサイズがどれほど小さくなるかを，実験で確かめる．VS-SDD のアイデアは同型な関数を表すノードを一つにまとめることなので，ここでは対称性のある論理関数に焦点を当てる．本稿では以下の実験 A 及び B を行った．

実験 A では，CNF の形で与えられた論理関数を，動的 vtree 探索 [8] を用いて SDD へ変換し，SDD 自体のサイズとその変数シフト形 (VS-SDD) のサイズを比較した．CNF を SDD へと組み上げる際には，SDD package version 2.0 [9] を利用し，動的 vtree 探索の際の初期 vtree は平衡二分木とする．

ここではまず，同じく変数の置換により同じになる関数を表現するノードをまとめるというアイデアに基づいている SymDDG [2] の実験でも使われた，計画問題に関する CNF のデータセットを用いる．決定的な計画問題は，状態変数の集合，初期状態，目標状態の集合，行動の集合の 4 つ組で表され，1 回行動を行うとそれに従って状態変数の中身が (決定的に) 変化する．期間 T が与えられたとき，この計画問題に対する計画とは，初期状態を目標状態へと変化させる，$t = 0, \ldots, T-1$ に対する行動の列である．計画問題を論理関数で表現する方法につい

表 2　実験 A の結果.

問題及び変数個数		S	V	V/S (%)
blocks-2_t3	248	8811	7057	80.1
blocks-2_t5	406	31861	28858	90.6
bomb-5-1_t3	348	3798	2278	60.0
bomb-5-1_t5	564	6327	3960	62.6
bomb-5-1_t7	780	11212	7287	65.0
bomb-5-1_t10	1104	16514	10426	63.1
comm-5-2_t3	488	20584	18033	87.6
emptyroom-4_t3	116	1822	1146	62.9
emptyroom-4_t5	188	3090	1885	61.0
emptyroom-4_t7	260	5073	3001	59.2
emptyroom-4_t10	368	106737	103417	96.8
emptyroom-8_t3	244	10511	8549	81.3
safe-5_t3	54	567	441	77.8
safe-5_t5	86	898	640	71.2
safe-5_t7	118	1710	1314	76.8
safe-5_t10	166	2506	1756	70.1
safe-30_t3	304	5476	4067	74.3
safe-30_t5	486	8710	6328	72.7
safe-30_t7	668	14449	10371	71.8
safe-30_t10	941	23469	17421	74.2
8-Queens	64	2222	1624	73.1
9-Queens	81	5559	4767	85.8
10-Queens	100	10351	9159	88.5
11-Queens	121	30611	28876	94.3
Matching-6x6	60	13091	12671	96.8
Matching-8x8	112	98200	97103	98.8
Matching-6x18	192	36228	34241	94.5

表 3　実験 B の結果.

問題及び変数個数		B	S	V	V/S (%)	+Implicit Partitioning B	S	V	V/S (%)
8-Queens	64	4898	2647	979	37.0	2540	712	251	35.3
9-Queens	81	19110	5787	2591	44.8	9906	1538	692	45.0
10-Queens	100	51886	12260	5884	48.0	26666	3918	1984	50.6
11-Queens	121	189640	35703	23682	66.3	97489	13076	8636	66.0
12-Queens	144	870336	110407	67469	61.1	449026	37480	24837	66.3
13-Queens	169	4088784	425125	288914	68.0	2114867	159184	112703	70.8
14-Queens	196	19144832	2179800	1525922	70.0	9914781	1049888	761602	72.5
Matching-6x6	60	3692	4914	3986	81.1	2561	2587	1978	76.5
Matching-8x8	112	22832	29842	24082	80.7	15781	15691	11850	75.5
Matching-10x10	180	123956	160274	129042	80.5	85545	84235	63242	75.1
Matching-6x18	192	20588	30289	21007	69.4	14273	14346	7960	55.5
Matching-8x24	352	145712	214289	147471	68.8	100773	100874	54856	54.4
Matching-10x30	560	902196	1328145	909327	68.5	623145	622602	334088	53.7
Path-6x6	60	6980	10685	7273	68.1	4573	4751	2556	53.8
Path-8x8	112	91438	165977	123319	74.3	60840	70254	42132	60.0
Path-10x10	180	1073156	2386085	1897815	79.5	719989	995343	670069	67.3
Path-6x18	192	29924	54911	40029	72.9	19885	21622	11772	54.4
Path-8x24	352	374830	822133	637779	77.6	251368	321357	191971	61.6
Path-10x30	560	4249716	11359107	9282557	81.7	2865909	4541423	3143189	69.2
Cardinality-10	2046	44792	77410	2064	2.7	42756	33396	1263	3.8
Cardinality-20	2046	85092	131370	5360	4.1	83066	57309	3469	6.1
Cardinality-50	2046	203592	272548	21090	7.7	201596	122196	14439	11.8
Cardinality-100	2046	393092	476762	60984	12.8	391146	222529	43281	19.4
Knapsack-(300,10)	300	31912	14214	7257	51.1	30613	7977	5111	64.1
Knapsack-(300,20)	300	104120	35545	29523	83.1	100938	27870	25365	91.0
Knapsack-(600,10)	600	64414	28065	11021	39.3	61849	14026	7123	50.8
Knapsack-(600,20)	600	206472	51916	36826	70.9	199891	36346	30152	83.0

ては，文献 [2] を参照のこと．本実験では，Sym-DDG の実験でも用いられた，“blocks-2”, “bomb-5-1”, “emptyroom-4/8”, “safe-5/30” の各計画問題を使用する．

もう一つは，より明らかな対称性がある論理関数のデータセットを用いる．そのうちの一つが N-クイーン問題である．これは，$N \times N$ のチェスボードに、N 個のクイーンを互いに干渉しないよう配置する問題である．この問題に対し，ボードの各マスに変数を割り当て，*true* のマスにクイーンを置くと N-クイーン問題の解となるときに *true* となる論理関数を考える．この論理関数は，ZDD などの決定図のベンチマークとして用いられる [7], [18]．もう一つは図 1(b) のようなグリッドグラフの部分グラフ索引化である．文献 [15], [22] にあるように，グラフに対してある条件を満たす部分グラフを索引化することには多くの応用がある．ここで，グリッドグラフは，その構造の単純さと比べ部分グラフの索引化が難しいため，ベンチマークとしてよく用いられる [13]．ここでは特にマッチングを成す部分グラフの索引化を考える．ここで，チェスボードにもグリッドグラフにも，線対称性や点対称性があることが観察できる．

表 2 が実験 A の結果である．“S” の列は SDD サイズ，“V” の列は VS-SDD サイズ，“V/S” の列は SDD サイズと比べた VS-SDD サイズの比を表す．ここで，計画問題の末尾の “_tn” は，期間が $T = n$ であることを表し，マッチング問題の末尾はグリッドサイズを表す．まず，多くの計画問題のデータに対し，変数シフト形のサイズは元の SDD のサイズの 60〜80%程度に圧縮できていることがわかる．V/S の列に示した圧縮率は，DDG のサイズと比べた Sym-DDG [2] の圧縮率と比べて，同等か，一部のデータでは勝っている．一方で，N-クイーンのデータでは $N = 11$ を除き相変わらず良い圧縮率が得られているが，マッチングのデータでは変数シフトの効果は薄かった．動的 vtree 探索では，強い依存関係をもつ変数を近くに集めることで小さな SDD を達成する傾向がある．計画問題のデータでは，近い t に関係した変数が近くに集まったことで，対称性をうまく捉えられたと思われる．

実験 B では，明らかな対称性がある論理関数に注目し，各論理関数をトップダウンコンパイラ [22] を用いて SDD に組み上げ，その変数シフト形とサイズを比較した．ここで，vtree としては，その論理関数の対称性を取り入れたものを使用する．詳細は省くが，これらの vtree は，例えばグリッドの線対称性など明らかな対称性しか使わずに簡単に作れるものである．また，ここで用いる論理関数は BDD の文献でよく現れるものなので，良い変数順を用いた BDD ともサイズを比較した．ここで BDD サイズは SDD のように，節点数ではなく辺の数で評価した．さらに，トップダウンコンパイラ [22] は，通常 Implicit Partitioning (IP) [21] というサイズ削減のテクニックを適用した SDD を出力するので，本実験では IP を適用した BDD, SDD, VS-SDD サイズも測定した．

使うデータセットは，実験 A でも用いた N-クイーンとマッチングに加え，次の 3 種類である．まず “Path-mxn” は，$m \times n$ グリッドグラフの対角コーナー間の単純パスを索引化する問題である．他の 2 種類のデータセットは，各変数を $\{0, 1\}$-変数とみなしたときのナップサック制約 $\sum_i w_i X_i \leqq \theta$, $w_i, \theta \in \mathbb{R}_+$ を満たす組合せを索引化する問題である．ここで各 w_i は重み，θ は閾値とよばれる．このような索引化は組合せ最適化問題を解くのに実用的に重要である [3]．ここで，もし重みとして現れる値の種類が少なければ，SDD を作る際に多くの同型な制約，すなわち変数は異なるが重みや閾値は同じ制約が現れることが期待される．“Cardinality-θ” は，変数の個数が 2046，閾値が θ で，全ての重みが 1 であるナップサック制約の索引化問題である．“Knapsack-(m, k)” は，m 変数で，重みの値が k 種類で，閾値が 300 であるナップサック制約の索引化問題である．ここで，各重みは $\{8 + \lfloor \frac{37i}{k-1} \rfloor \mid i = 0, \ldots, k-1\}$，つまり区間 $[8, 45]$ から等間隔にとった k 個の整数から一様ランダムに選んだ．

表 3 が実験 B の結果である．“B” の列は BDD サイズを表す．まず N-クイーン問題では，VS-SDD サイズは BDD や SDD のサイズより非常に小さく，圧縮率は全ての N で 73%以下になっている．これは VS-SDD が問題の対称性をうまく捉えられて

いることを表す．さらに，実験 B における SDD や VS-SDD サイズは実験 A よりも小さくなっている．一方で，N が大きくなると圧縮率が少し悪くなる．これは，プライム側は互いに素でなければならないが，サブ側はそうではない，という非対称性に起因すると思われる．次にナップサック制約についても，よく対称性が捉えられていることが観察できる．特に，変数の数が多く，重みの種類数が少ないほどよく圧縮が効くことがわかる．次にマッチングの問題については，IP を適用すると BDD と SDD のサイズはほぼ同程度だが，VS-SDD サイズはそれの 50〜75%程度になることがわかる．最後にパスの問題については，SDD サイズが BDD サイズより大きくなっており，これは SDD で用いた vtree が最適ではないことを示している．それでも，IP を適用すると VS-SDD サイズは BDD サイズより小さくなっており，SDD にとっては最適ではない vtree を使っても VS-SDD では小さな表現が得られる可能性があることを示唆している．

7 関連研究と結論

ここで関連研究について述べる．BDD に対しては，追加のデータ構造を用いてより小さく表現する研究は数多くあり，その中で最も有名なのは否定枝および，variable shifter を含む attributed edges [16], [19] である．最近は，データ構造の付加により BDD と ZDD の良いとこ取りを目指す研究もあり，chain-reduced ZDD/BDD [7] や tagged BDD [27] が提案されている．VS-SDD によく似た構造として，変数の番号をずらす手法を提案した↑ΔBDD [1] がある．このアイデアは本稿の変数シフトに似ているが，そのまま SDD の vtree に適用すると上手くいかないので，VS-SDD では行きがけ順の変数順序を導入している．また，合同性の定義を行うことで，↑ΔBDD ではできない $O(|\alpha||\beta|)$ 時間での `Apply` を可能にした．

本稿では，変数シフト SDD という，SDD をより小さく表現するデータ構造を提案した．VS-SDD は，表現の一意性や，クエリや操作のサポートといった SDD の重要な性質を受け継ぐ．また，VS-SDD のサイズは SDD より大きくなることはなく，サイズが指数倍小さくなる例も存在する．さらに，実験にて，いくつかの論理関数に対し，VS-SDD は関数の対称性を捉えてサイズが小さくなることを確かめた．

文　献

[1] Anuchit Anuchitanukul, Zohar Manna, and Tomás E. Uribe. Differential BDDs. In *Computer Science Today*, pp. 218–233, 1995.

[2] Anicet Bart, Frédéric Koriche, Jean-Marie Lagniez, and Pierre Marquis. Symmetry-driven decision diagrams for knowledge compilation. In *ECAI*, pp. 51–56, 2014.

[3] David Bergman, Andre A. Cire, Willem-Jan van Hoeve, and John Hooker. *Decision Diagrams for Optimization*. Springer International Publishing, 2016.

[4] Simone Bova. SDDs are exponentially more succinct than OBDDs. In *AAAI*, pp. 929–935, 2016.

[5] Guy Van den Broeck and Adnan Darwiche. On the role of canonicity in knowledge compilation. In *AAAI*, pp. 1641–1648, 2015.

[6] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Comput.*, Vol. C-35, pp. 677–691, 1986.

[7] Randal E. Bryant. Chain reduction for binary and zero-suppressed decision diagrams. In *TACAS*, pp. 81–98, 2018.

[8] Arthur Choi and Adnan Darwiche. Dynamic minimization of sentential decision diagrams. In *AAAI*, pp. 187–194, 2013.

[9] Arthur Choi and Adnan Darwiche. The SDD package: version 2.0. `http://reasoning.cs.ucla.edu/sdd/`, 2018.

[10] Adnan Darwiche. SDD: a new canonical representation of propositional knowledge bases. In *AAAI*, pp. 819–826, 2011.

[11] Adnan Darwiche and Pierre Marquis. A knowledge compilation map. *J. Artif. Intell. Res.*, Vol. 17, pp. 229–264, 2002.

[12] Leonardo Duenas-Osorio, Kuldeep S. Meel, Roger Paredes, and Moshe Y. Vardi. Counting-based reliability estimation for power-transmission. In *AAAI*, pp. 4488–4494, 2017.

[13] Hiroaki Iwashita, Yoshio Nakazawa, Jun Kawahara, Takeaki Uno, and Shin-ichi Minato. Efficient computation of the number of paths in a grid graph with minimal perfect hash functions. Technical Report TCS-TR-A-13-64, Division of Computer Science, Hokkaido University, 2013.

[14] Doga Kisa, Guy Van den Broeck, Arthur Choi, and Adnan Darwiche. Probabilistic sentential decision diagrams. In *AAAI*, pp. 558–567, 2014.

[15] Donald E. Knuth. *The Art of Computer Programming*, Vol. 4A: Combinatorial Algorithms, Part I. Addison-Wesley, 2011.

[16] Jean-Christophe Madre and Jean-Paul Billon. Proving circuit correctness using formal comparison between expected and extracted behaviour. In *DAC*, pp. 205–210, 1988.

[17] Takanori Maehara, Hirofumi Suzuki, and Masakazu Ishihata. Exact computation of influence spread by binary decision diagrams. In *WWW*, pp. 947–956, 2017.

[18] Shin-ichi Minato. Zero-suppressed BDDs for set manipulation in combinatorial problems. In *DAC*, pp. 272–277, 1993.

[19] Shin-ichi Minato, Nagisa Ishiura, and Shuzo Yajima. Shared binary decision diagram with attributed edges for efficient boolean function manipulation. In *DAC*, pp. 52–57, 1990.

[20] Masaaki Nishino, Takeru Inoue, Norihito Yasuda, Shin-ichi Minato, and Masaaki Nagata. Optimizing network reliability via best-first search over decision diagrams. In *IEEE INFOCOM*, pp. 1817–1825, 2018.

[21] Masaaki Nishino, Norihito Yasuda, Shin-ichi Minato, and Masaaki Nagata. Zero-suppressed sentential decision diagrams. In *AAAI*, pp. 1058–1066, 2016.

[22] Masaaki Nishino, Norihito Yasuda, Shin-ichi Minato, and Masaaki Nagata. Compiling graph substructures into sentential decision diagrams. In *AAAI*, pp. 1213–1221, 2017.

[23] Umut Oztok and Adnan Darwiche. A top-down compiler for sentential decision diagrams. In *IJCAI*, pp. 3141–3148, 2015.

[24] Tian Sang, Paul Beame, and Henry Kautz. Performing Bayesian inference by weighted model counting. In *AAAI*, pp. 475–481, 2005.

[25] Yuya Sasaki, Yasuhiro Fujiwara, and Makoto Onizuka. Efficient network reliability computation in uncertain graphs. In *EDBT*, 2019. to appear.

[26] Fabio Somenzi. Efficient manipulation of decision diagrams. *Int. J. Softw. Tools Technol. Transf.*, Vol. 3, pp. 171–181, 2001.

[27] Tom van Dijk, Robert Wille, and Robert Meolic. Tagged BDDs: combining reduction rules from different decision diagram types. In *FMCAD*, pp. 108–115, 2017.