写像枝を用いた系列二分決定グラフの効率化

青木 洋士[1,a)]　湊 真一[1,3,b)]　山下 茂[2,c)]

概要：二分決定グラフ (Binary Decision Diagram, BDD) は，論理関数を効率良く処理することができるグラフである．BDD 処理系のメモリ効率は，節点数に依存する．BDD の節点数を削減する手法として，BDD に属性枝を付与する手法が提案されている．BDD に属性枝を付与すると，同じ部分グラフの数が増加し，同一の節点が効率良く共有される．一方，系列二分決定グラフ (Sequence Binary Decision Diagram, SeqBDD) は，文字列集合を表現するデータ構造である．SeqBDD は，変数の出現順序や構築ルールが BDD と異なる．このため，SeqBDD は，BDD とは違った節点の共有手法を適用できる可能性がある．本稿では，属性枝を付与した新しい SeqBDD の節点の共有手法を提案する．実験により，SeqBDD に対して提案した属性枝の手法を適用すると，節点数を削減することができ，さらに，特定の SeqBDD の演算を高速化できることを確認した．

キーワード：系列二分決定グラフ，属性枝，写像

An Efficient Sequence Binary Decision Diagrams with Mapping Edges

Aoki Hiroshi[1,a)]　Minato Shin-ichi[1,3,b)]　Yamashita Shigeru[2,c)]

Abstract: A Binary Decision Diagram (BDD) gives efficient boolean function manipulations. The memory efficiency of BDDs depends on the number of their nodes. We can reduce the number of nodes of BDDs by adding attributed edges: nodes are shared efficiently since the number of the same sub-graphs increases when attributed edges are used. Sequence Binary Decision Diagram (SeqBDD) is a data structure that represents a set of strings. SeqBDDs are different from BDDs with respect to the variable ordering and the construction rules. Therefore, there may be another approach to share nodes of SeqBDDs. In this paper, we propose a new technique for sharing nodes of SeqBDDs with attributed edges. Our experimental results show that our attributed edge reduce the number of nodes of SeqBDDs and make some oprations of a SeqBDD fast.

Keywords: SeqBDD, attributed edges, mapping

1. はじめに

二分決定グラフ (Binary Decision Diagram, BDD) [1] は，論理関数を表現するデータ構造であり，論理関数をコンパクトかつ一意なグラフで表現することができる．さらに，BDD を用いた効率の良い論理関数の合成アルゴリズム *Apply* [2] が実現されている．これらの性質から，BDD は，VLSI 設計における論理合成に代表される様々な分野で応用されている．

BDD を拡張した決定グラフのデータ構造を用いて，論理関数以外のデータ処理に BDD の効率の良い機能を利用する研究が盛んに行われている．ゼロサプレス型二分決定グラフ (Zero-suppressed Binary Decision Diagram, ZDD) [3] と系列二分決定グラフ (Sequence Binary Decision Dia-

[1] 北海道大学情報科学研究科
Graduate School of Information Science and Technology, Hokkaido University, Japan
[2] 立命館大学情報理工学研究科
Graduate Schoool of Information Science and Engineering, Ritsumeikan University, Japan
[3] JST ERATO 湊離散構造処理系プロジェクト
ERATO MINATO Discrete Structure Manipulation System Project, JST, Japan
a) kioa@ist.hokudai.ac.jp
b) minato@ist.hokudai.ac.jp
c) ger@cs.ritsumei.ac.jp

ⓒ 2013 Information Processing Society of Japan

gram, SeqBDD) [4] は，それぞれ組合せ集合と系列集合を表現できるように BDD を拡張したデータ構造であり，これらの集合に対して効率の良いデータ操作を行うことができる．πDD [5] は，順列集合を表現し，パズル問題の解答を高速に全列挙することができる．さらに，論理関数だけでなく，量子回路を表現するデータ構造として，QMDD [6] や DDMF [7] といったデータ構造が提案されている．これらのデータ構造は，いずれも BDD の性質を引き継いでいるため，対象を一意にかつコンパクトに表現できる．そのため，*Apply* と同様の手法で効率よくデータ同士の合成を行うこともできる．

BDD 処理系は，属性枝の手法を導入することで処理性能を向上させることができることが知られている．属性枝の手法は，BDD の部分グラフがもつ情報の一部を属性として枝に抽出することで，その枝より下の部分グラフを他の部分グラフと共有しやすくする手法である．この手法を導入することで，BDD の節点数を削減することができる．これによって，メモリ使用量を削減し，*Apply* の計算速度を向上させることができる．否定枝 [2] は，論理関数の否定演算を表す属性枝であり，枝に 1 ビットの否定演算の有無を表す情報を付与することで実現される．否定枝を用いると，互いに否定関係にある論理関数を同じ部分グラフとして共有することができる．否定枝は，その性質上，論理関数を表現できる決定グラフにしか効率よく動作しない．また，Variable Shifter [8] は枝に接続する 2 つの変数ラベルの値の差を表す属性枝である．この枝を用いると，変数ラベルが異なっていたとしても同じ変数ラベルの値の差をもつグラフを共有することができる．この手法は，変数ラベルの出現順序が一意に定まった決定グラフでは効果的であるが，変数ラベルの出現順序に自由度のある SeqBDD では効果的に利用できない．このように，属性枝の手法は，特定の決定グラフに対してしか効率よく利用できない．

本稿では，あらゆる決定グラフで利用できる属性枝として，写像枝を提案する．写像枝は，変数ラベルの読み替えを表す属性を写像として付与した枝である．写像枝には，変数ラベルを読み替える任意の写像を付与できるため，Variable Shifter よりも高い自由度で変数ラベルを読み替えることができる．複数種類の写像を用いた貪欲な写像枝と，1 種類の写像のみしか用いない高速な処理が可能な写像枝の 2 つの写像枝の手法を示す．本稿では，これらの 2 つの写像枝の手法について，SeqBDD を用いた実装と実験を行った．貪欲な写像枝を用いた手法は，多量の節点を削減できるが，写像演算のオーバーヘッドが大きく，構築や *Apply* の演算に通常の SeqBDD よりも多くの時間がかかってしまう．一方，高速な処理が可能な写像の手法は，共有できる節点の数が貪欲な写像枝の手法よりも小さいが，写像演算のオーバーヘッドが小さく，通常の SeqBDD と同程度の時間で *Apply* の演算を行うことができる．さらに，写像を用いた性質を活用したグラフからの特定の文字の数え上げ演算では，通常の SeqBDD よりも高速に動作する．

以下，本稿の 2 章では，BDD とそれを拡張したデータ構造である ZDD，SeqBDD について説明する．次に，3 章では，BDD における既存の属性枝の手法について述べる．さらに，4 章では，新しい属性枝として，写像枝を提案する．そして，5 章では，計算機実験による評価方法とその結果を述べる．最後に，6 章では，本稿のまとめと今後の課題を述べる．

2. 準備

2.1 系列

本稿で扱うアルファベットは全順序集合であるものとし，その集合を Σ とする．2 つの文字 $c_1, c_2 \in \Sigma$ の順序関係を $c_1 < c_2$ のように記号 $<$ で表す．$<$ が定義されているとき，$c_1 < c_2 \Leftrightarrow c_2 > c_1$，$\neg(c_1 < c_2) \wedge \neg(c_1 > c_2) \Leftrightarrow c_1 = c_2$ として，記号 $>$ と記号 $=$ を定義する．任意の c_1, c_2 について，$c_1 < c_2$，$c_1 > c_2$，$c_1 = c_2$ のいずれかが常に満たされるものとする．系列全体の集合は，Σ^* で表される．系列 $s \in \Sigma^*$ の長さは，$|s|$ と表記する．長さが 0 の系列は，空列と呼び ε で表す．2 つの系列 $a, b \in \Sigma^*$ を連結した系列は，ab で表す．系列 s について $s = xyz$，$x, y, z \in \Sigma^*$ であるとき，x, y, z をそれぞれ s の接頭辞，部分系列，接尾辞と呼ぶ．系列 s の i 番目の文字を $s[i]$ で表し，$s[i...j]$ は s の i 番目から j 番目までの部分系列を表し，$i > j$ のときは，$s[i...j] = \varepsilon$ と表記する．s^R は系列 s の逆列を表し，$s^R = s[|s|]...s[1]$ のことである．

2.2 写像

本稿で扱う写像は，文字から文字への写像である．$A, B \subset \Sigma$ とする．写像 $f : A \to B$ が定義されているとき，文字 $c \in A$ を f で写像させて得られる文字を $f(c)$ で表す．また，任意の文字 $c \in \Sigma$ について，$f(c) = c$ となるような f を恒等写像と呼び，I で表すものとする．また，文字列集合 P が与えられたとき，その文字列集合に含まれる全ての文字列の文字を f で置換した文字列集合を $f(P)$ で表す．写像を具体的に表記する際は，恒等写像との違いだけを表記するものとする．例えば，$f = \{1 \to a, 3 \to c\}$ は，$f(1) = a$，$f(2) = 2$, $f(3) = c$ である．

2.3 二分決定グラフ

BDD は論理関数をコンパクトかつ一意に表現することができる閉路のないグラフである [2]. BDD の節点は，論理関数の各変数と対応する．変数の値が 1 ならば，1-枝に遷移し，0 ならば，0-枝に遷移する．遷移を繰り返し，最終的にたどり着いた終端節点（0-終端節点か 1-終端節点）の値が，その論理関数に各変数の値を代入したときの結果となる．

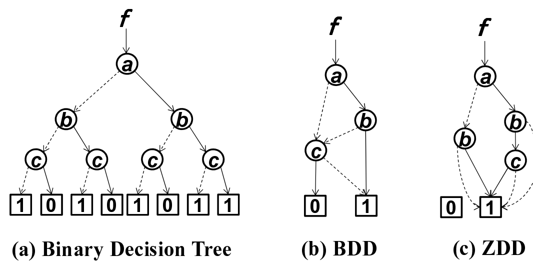

図 1　Binary Decision Tree, BDD, ZDD

論理関数の変数の出現順序を固定して，下の 2 つの簡約化ルールを可能な限り適用したものを Reduced Ordered BDD (ROBDD) と呼ぶが，一般的に ROBDD を BDD と呼ぶため，本稿での BDD もこれに準ずるものとする．

- 冗長な節点を削除する
- 等価な節点を共有する

図 1 (a) の Binary Decision Tree を簡約化すると，図 1 (b) のような BDD になる．図 1 (a), (b) のグラフを論理関数とみなした場合，各変数に値を代入することで得られた結果は，論理関数の結果に対応する．例えば，a=0, b=0, c=0 という入力が与えられた場合，a の 0-枝，b の 0-枝，c-の 0-枝，と順にグラフをたどり，最終的に行きついた終端節点の示す値の 1 が結果として得られる．図 1 (b) の BDD の場合，グラフをたどる途中で，b に対応する節点をたどらないが，これは，b の値が 0 であっても 1 であっても結果が変わらないということを表している．

BDD は上の簡約化ルールにより，同じ論理関数を一意に表現することができ，別の論理関数を表す BDD との間でも共通な部分節点を共有することができる．

2.4　ゼロサプレス型二分決定グラフ

ZDD [3] は組合せ集合を効率よく表現することができるデータ構造である．ZDD は，図 1 (a) に示されるような Binary Decision Tree で表現した組合せ集合を以下の簡約化ルールを可能な限り適用して縮約したものである．

- 1-枝が 0-終端節点を指している節点を削除する．
- 等価な節点を共有する．

ZDD のそれぞれの経路は，1 つの組合せを表す．ZDD の節点に対応する変数の値は，その節点の文字が組合せ集合に含まれるか否かに対応する．ZDD の根から 1-終端節点までの経路で出現する 1-枝をもつ節点の文字の集合が 1 つの組合せ集合に相当する．各経路では，同じ文字は高々 1 回出現し，それらの出現順序は固定されている．このため，ZDD は，文字の出現順序が異なる組合せ集合を表現できない．

図 1 (c) に ZDD の例を示す．図 1 (c) のグラフを組合せ集合とみなした場合，$\{\{\varepsilon\}, \{a\}, \{b\}, \{a,b\}, \{a,b,c\}\}$ を表す．各変数に値を代入することで得られた結果は，1 ならば集合に含まれ，0 ならば集合に含まれないことを表す．a=1, b=1, c=0 という入力が与えられた場合，a の 1-枝，b の 1-枝，c の 0-枝，と順にグラフをたどり，最終的に行きついた終端節点の示す値の 1 が結果として得られ，組合せの a, b が集合に含まれることが分かる．図 1 (c) の ZDD をたどる途中では，a=0, b=0 のように，文字 c に対応する節点をたどらない経路があるが，これは，a と b を含まない組合せには，文字 c が出現しないことを意味する．

ZDD は，同じ組合せ集合を一意に表現することができ，異なる組合せ集合を現す ZDD 同士で共通な部分グラフを共有できる．ZDD 同士の集合演算には，Apply [2] と呼ばれる効率良く 2 つの組合せ集合同士の演算を行うことができるアルゴリズムを用いる．Apply は，2 つの節点についての再帰的な関数呼び出しの結果をキャッシュしておき，同じ演算がもう一度実行されたときにキャッシュされている結果を返すことで，高速な演算を実現する．これらの機能は，ZDD とは簡約化ルールが異なる Binary Decision Diagram (BDD) [1] で利用されている手法を応用したものである．

2.5　系列二分決定グラフ

SeqBDD [4] は ZDD では効率的に扱うことができない系列集合を効率良く扱うことができるデータ構造であり，頻出部分系列マイニングや部分系列集合を効率良く表現する SuffixDD [9] のデータ構造として利用されている．

SeqBDD は，ZDD の節点に対応する文字の順序付けルールを 1-枝側に適用しないため，1 つの経路に任意の順番で何度も同じ文字の出現が可能である．これにより，系列の集合を効率的に表現することができる．

SeqBDD のそれぞれの経路は，1 つの系列を表し，その系列は，根から 1-終端節点までの経路で出現する 1-枝をもつ節点の文字を順につなげたものとなる．SeqBDD の節点を $N = node(x, N_1, N_0)$ と表すと，N は，文字 x に対応し，1-枝に N_1，0-枝に N_0 がつながっている節点を表す．各節点は，系列の集合に対応し，N は，N_1 が表す各系列の接頭辞に x を付加した系列集合と N_0 が表す系列集合の和集合を表す．このため，SeqBDD を構成するそれぞれの節点が，系列集合を表現する．SeqBDD における 1-終端節点は，空列からなる集合 $\{\varepsilon\}$ を表し，0-終端節点は空集合 ε を表す．

SeqBDD はトライや DAWG と同じように共通な接頭辞を共有して 1 つのパスで表すグラフ表現である．しかし，二分グラフであるため，0-枝側に節点を連ねることで多分岐構造を表現する．

SeqBDD の例を図 2 に示す．図 2 の SeqBDD P のルート節点は，$\{abb, ac, bc, dc\}$ の系列集合を現している．P の部分グラフの各節点もそれぞれが系列集合を表している．

SeqBDD は，0-枝では変数順序を必ず満たすようにする

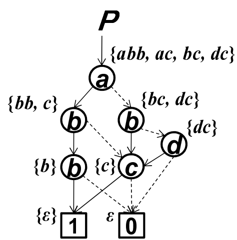

図 2　SeqBDD

表 1　SeqBDD 処理系の基本演算

P.top	系列集合 P の先頭文字の中で最上位の文字を返す．
P.onset(x)	系列集合 P の部分集合のうち，x から始まる系列集合から先頭文字を取り除いた系列集合を返す．
P.offset(x)	系列集合 P のうち，x から始まらない系列集合を返す．
P.push(x)	系列集合 P の全要素の先頭に文字 x を付加した系列集合を返す．
$P \cup Q$	P と Q の和集合を返す．
$P \cap Q$	P と Q の積集合を返す．
$P \setminus Q$	P と Q の差集合を返す．
$\|P\|$	P を含む P の子孫節点の数を返す．
SeqBDD(x, P, Q)	P.push(x) $\cup$ Q に相当する系列集合を返す．

が，1-枝では変数順序の制約を外すことによって，ZDD を拡張している．このため，SeqBDD は，ZDD と同じ簡約化規則と節点の共有機能を用い，集合演算を行うことができる．

SeqBDD 処理系に用意されている基本的な演算を表 1 に示す．これらの演算の多くは，Loekito らの論文 [4] や伝住らの論文 [9] で用いられている．これらの演算のうち，和集合を求める集合演算 $\cup$ のアルゴリズムを図 3 に示す．集合の積集合を求める演算 $\cap$ や差集合を求める演算 $\setminus$ は，和集合を求める演算と同じ再帰的なアルゴリズムで実現できる．

これらの集合演算は，BDD/ZDD の演算と同様なアルゴリズムである Apply で実現されている．Apply では，演算キャッシュが上書きされない限り，演算キャッシュの利用により同じ節点のペアでの再帰呼び出しが 2 回目以降は行われない．Apply の計算時間は最悪の場合には $O(|P||Q|)$ となるが，多くの実用的な例題では，入力と出力の SeqBDD サイズの和の線形時間で抑えられることが経験的に知られている [2]．

3.　既存の属性枝

BDD 処理系は，属性枝の手法を用いることで処理性能を向上させることができることが知られている．属性枝の手法は，BDD の部分グラフがもつ情報の一部を属性として枝に抽出することで，その枝が示す部分グラフを他の部分グラフと共有しやすくする手法である．この手法を導入することで，BDD の節点数を削減することができる．これによって，メモリ使用量を削減することができる．

PROCEDURE $P \cup Q$
Require: P, Q: SeqBDD
1: **if** P = 0-終端節点 **then**
2: 　**return** Q
3: **else if** Q = 0-終端節点 **then**
4: 　**return** P
5: **else if** $P = Q$ **then**
6: 　**return** P
7: **end if**
8: **if** 演算キャッシュに $P \cup Q$ の結果がある **then**
9: 　**return** 演算キャッシュの $P \cup Q$ の結果
10: **end if**
11: $p \leftarrow P$.top; $q \leftarrow Q$.top
12: $P_1 \leftarrow P$.onset(p); $P_0 \leftarrow P$.offset(p)
13: $Q_1 \leftarrow Q$.onset(q); $Q_0 \leftarrow Q$.offset(q)
14: **if** $p = q$ **then**
15: 　$r \leftarrow p$
16: 　$R_1 \leftarrow P_1 \cup Q_1$; $R_0 \leftarrow P_0 \cup Q_0$
17: **end if**
18: **if** $p < q$ **then**
19: 　$r \leftarrow p$
20: 　$R_1 \leftarrow P_1$; $R_0 \leftarrow P_0 \cup Q$
21: **end if**
22: **if** $p > q$ **then**
23: 　$r \leftarrow q$
24: 　$R_1 \leftarrow Q_1$; $R_0 \leftarrow P \cup Q_0$
25: **end if**
26: $R \leftarrow$ SeqBDD(r, R_1, R_0)
27: 演算キャッシュに $P \cup Q$ の結果として，R を保存する
28: **return** R

図 3　2 つの SeqBDD の和集合 $\cup$ の演算アルゴリズム

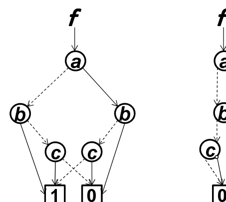

図 4　否定枝を用いない BDD と否定枝を用いた BDD

3.1　否定枝

否定枝 [2] は，論理関数の否定演算を表す属性枝であり，枝に 1 ビットの否定演算の有無を表す情報を付与することで実現される．否定枝を用いると，互いに否定関係にある論理関数を同じ部分グラフとして共有することができる．

図 4 の左の BDD に対して否定枝を適用すると右のようなグラフで表現される．右側のグラフで小さな丸印が付与された枝が否定枝である．否定枝が示すグラフは，グラフが表す関数を否定した関数を意味する．これにより，互いに否定関係にある論理関数を同じグラフで表現することができ，節点が削減される．

否定枝を用いた場合にも，BDD を一意な形で表現でき

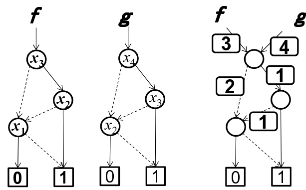

図 5 Variable Shifter を用いない BDD と Variable Shifter を用いた BDD

ないと効率の良い BDD の操作ができなくなるおそれがある．そこで，BDD に否定枝を利用する際には，次の 2 つのルールを用いる．

- 0-枝には否定枝を用いない
- 1-終端節点を用いない

これらの 2 つのルールを適用することで否定枝を用いた BDD は一意に定まる．

否定枝を導入することにより，BDD に対する否定演算を効率よく実現することができる．これは，BDD のルート節点に否定枝を付与するか取り除くかだけで，否定演算が実現できるからである．

さらに，否定演算だけでなく，2 つの BDD から $Apply$ により新たな BDD を取得する演算も効率よく実現できる．例えば，f と g を論理関数としたとき，ドモルガンの公式により，$f \vee g = \neg(\neg f \wedge \neg g)$ の等式が得られる．このとき，$f \vee g$ と $\neg f \wedge \neg g$ は互いに否定関係にあるため，いずれか一方の演算結果を BDD として表現できれば，もう一方もその BDD に対する一度の否定演算によって容易に取得することができる．よって，否定枝を用いることで，異なる演算間で演算結果を共有することができ，演算処理を高速化することが可能となる．

3.2 Variable Shifter

2.1 節で示したように，BDD はグラフの各パスでの変数ラベルの出現順序が固定されている．この変数ラベルの出現順序のレベルの差を利用して，グラフの構造を共有する属性枝として，Variable Shifter [8] が知られている．図 5 の左の 2 つの BDD に対して，Variable Shifter を適用すると右のようなグラフで表現される．Variable Shifter では，グラフのルートの枝にルート節点の変数のレベルの値を付与し，それ以降の枝には，枝が接続する 2 つの節点のレベルの差を付与する．これにより，互いに異なる変数ラベルをもつ BDD 同士でも，終端節点から見て同じ変数差をもつ部分グラフがあれば，部分グラフを共有できるようになる．Variable Shifter も，否定枝と同様に BDD の形を一意に定めることができ，既存の BDD の演算を高速に実現することができる．

これらの否定枝と Variable Shifter の属性枝は，論理関数を表す BDD に対して適用することが想定されている．このため，論理の否定関係を有効に利用できない ZDD では，否定枝は効率よく動作しないし，変数ラベルの順序関係が単調に変化せず，同じ変数ラベルが一つのパスに何度も出現する SeqBDD では Variable Shifter を有効に利用できない．

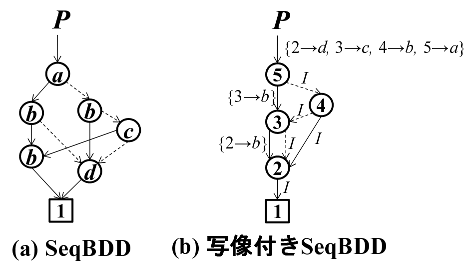

図 6 同じ文字列集合を表す SeqBDD と写像付き SeqBDD

4. 写像枝

4.1 写像枝の定義

SeqBDD の各枝に写像情報を付与し，SeqBDD の節点の共有効率を向上させるデータ構造として，写像付き SeqBDD を提案する．写像枝は，決定グラフへ導入する属性枝の属性として，写像を用いたものである．このデータ構造では，節点を用いて表現されていた情報を写像という形で枝へ抽出する．これにより，通常の SeqBDD よりも多くの節点の共有が可能である．

写像枝に付与する写像は，定義域 Σ から値域 Σ への写像である．Σ は決定グラフ中で扱う変数のラベル文字のアルファベット集合である．この写像によって，変数ラベルを読み替える．写像付き SeqBDD を構成する節点は，通常の SeqBDD とほぼ同じであるが，各枝に写像が付与されている点が異なる．写像は，写像付き SeqBDD の各節点から伸びる枝と根節点を指す枝に付与される．さらに，写像 f と文字列集合 S について，$f(S)$ とすると，文字列集合 S が表す文字列中の全ての文字を写像 f によって，置き換えた文字列集合を表すものとする．

定義 1 写像付き SeqBDD の節点 $N = (c, N_1, N_0, f_1, f_0)$ について，N に対応付けらた文字が c，N の 1-枝及び 0-枝につながる節点を N_1，N_0，N の 1-枝及び 0-枝に付与された写像を f_1，f_0 とする．節点 N の表現する文字列集合を N.strs で表すものとする．このとき，写像付き SeqBDD の節点 N は次のように再帰的に文字列集合を表現する．

$$N.\mathrm{strs} = f_1(N_1.\mathrm{strs.push}(c)) \cup f_0(N_0.\mathrm{strs})$$

また，根の節点 N を指す枝に対応付けられた写像を f とすると，N と f で表される写像付き SeqBDD (N, f) は，$f(N.\mathrm{strs})$ という文字列集合として表される．

図 6 (a) は SeqBDD を表し，図 6 (b) は写像付き SeqBDD を表している．いずれのグラフも同じ文字列集合 $\{abb, ad, bd, cb, d\}$ を表している．図 6 (b) の文字 2 が対応付けられた節点は，文字列集合 $\{2\}$ を表す．文字 3 が対応

ⓒ 2013 Information Processing Society of Japan

付けられた節点は，文字列集合 $\{3b, 2\}$ を表す．この節点のもつ 1-枝，0-枝は，ともに同じ文字 2 に対応付けられた節点を指しているが，付与された写像がそれぞれ異なる．このため，文字 2 が対応付けられた節点は，1-枝から辿った場合は $\{b\}$，0-枝から辿った場合は $\{2\}$ と，異なる文字列集合として解釈される．このようにして，本来なら異なる節点で表される部分構造が同じ節点で共有される．

写像付き SeqBDD を実現するためには，$P = \mathrm{getSeqBDD}(c,\ P_1,\ P_0)$ の基本演算を定義することができればよい．表 1 の各演算中での節点の生成は，getSeqBDD 演算を用いて行う．よって，getSeqBDD 演算を定義できれば，表 1 の演算を行うことができる．次節以降では，2 つの写像枝の手法を示し，それぞれの getSeqBDD を定義する．

4.2 貪欲な写像枝

本節では，できるだけ多くの節点を共有できるように，貪欲に写像を決定する手法を示す．本節で示す写像枝を付与した写像付き SeqBDD は，グラフの終端側の文字情報をできるだけ多く写像として上位の枝に抽出する方法である．

まず，写像付き SeqBDD が扱う文字集合を次のようにして表すものとする．$\alpha, \beta \subset \Sigma$ は互いに素な文字集合である．写像付き SeqBDD が表現することができる文字列集合 P は，$P \subset \beta^*$ を満たすものとする．α は，写像枝を付与することによって，SeqBDD が表す文字を置きかえる際に利用される文字集合である．

次に，2 つの写像 f, g から新しく写像を生成する演算を定義する．

定義 2 $h = f \overset{+}{\leftarrow} g$ としたとき，h は次の写像を表す．

(1) $f(x) \neq x$ または，$\exists y(g(x) = f(y) \in \beta)$ のとき，

$$h(x) = f(x)$$

(2) (1) 以外のとき，

$$h(x) = g(x)$$

この演算は，f に g を足し合わせるような演算である．例えば，$f = \{1 \to a, 2 \to b\}, g = \{1 \to b, 3 \to c\}$ とすると，$f \overset{+}{\leftarrow} g = \{1 \to a, 2 \to b, 3 \to c\}$ となる．

定義 3 $h = f \overset{-}{\leftarrow} g$ としたとき，h は次の写像を表す．

(1) $f(x) = g(x)$ のとき，

$$h(x) = x$$

(2) (1) 以外のとき，

$$h(x) = f(x)$$

この演算は，f から g を引くような演算である．例えば，$f = \{1 \to a, 2 \to b, 3 \to c\}, g = \{1 \to b, 3 \to c\}$ とすると，$f \overset{-}{\leftarrow} g = \{1 \to a, 2 \to b\}$ となる．

次に，3 つの写像付き SeqBDD を $P = (N,\ f)$，$P_1 = (N_1,\ f_1)$, $P_0 = (N_0,\ f_0)$ で表す．このとき，$c \in \alpha$，P_1，P_0 に対しての getSeqBDD 演算を次の規則を適用することで実現する．

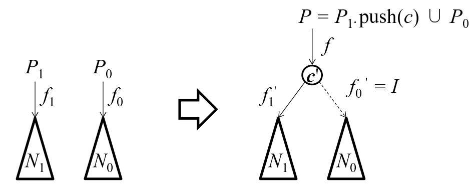

図 7 写像付き SeqBDD における $\mathrm{getSeqBDD}(c, P_1, P_0)$

(1) $f_0(x) = c$ となる x が存在するとき

$$\begin{cases} c' &= x \\ f &= f_0 \overset{+}{\leftarrow} f_1 \\ f'_1 &= f_1 \overset{-}{\leftarrow} f \\ f'_0 &= I \end{cases}$$

(2) (1) を満たさず，$f_1(x) = c$ となる x が存在し，かつ，$N_0.\mathrm{strs.onset}(c) = \varepsilon$ であるとき (N_0 が c から始まる文字列を含まないとき)

$$\begin{cases} c' &= x \\ f &= \{c' \to c\} \overset{+}{\leftarrow} f_0 \overset{+}{\leftarrow} f_1 \\ f'_1 &= f_1 \overset{-}{\leftarrow} f \\ f'_0 &= I \end{cases}$$

(3) (1), (2) を満たさないとき，

$$\begin{cases} c' &= min\{x | f_0(x) = x, x \in \alpha\} \\ f &= \{c' \to c\} \overset{+}{\leftarrow} f_0 \overset{+}{\leftarrow} f_1 \\ f'_1 &= f_1 \overset{-}{\leftarrow} f \\ f'_0 &= I \end{cases}$$

getSeqBDD の様子を図 7 に示す．c' は文字 c を置き換えて得られる文字列であり，f'_1 および f'_0 は上の規則によって，定められた getSeqBDD によって生成された節点から接続する 1-枝と 0-枝に付与される写像である．上の規則の 1 および 2 は既に使用されている変数ラベルを上位の枝に抽出する操作を表す．規則の 3 では，c を置き換える c' として，まだ使用していない文字を α から選び出している．

この演算から得られる写像付き SeqBDD の P は，文字列集合を一意に表現す．何故なら，P_1 と P_0 を表す写像枝付き SeqBDD の表現が一意に決まっているならば，getSeqBDD で得られる節点の文字と各枝に付く写像が一意に決まるため，図 7 の P の表現も一意に決まる．グラフの終端を表す空文字列 1 つからなる集合と空集合に対応する SeqBDD は，それぞれ，(1-終端節点，I)，(0-終端節点，I) として一意に決まる．よって，上の規則によって生成される写像付き SeqBDD は一意に表現できる．

図 6 (b) の写像付き SeqBDD は，上の規則を守って生成した写像付き SeqBDD であり，$\alpha = \{2, 3, 4, 5\}, \beta = \{a, b, c, d\}$ である．全ての節点を α の文字で置き換え，枝に写像を付与することで，一意に節点が決定されている．

さらに，SeqBDD の構築は，図 3 の *Apply* に代表される表 1 の各アルゴリズムによって構築される．これらのアル

ゴリズムは，getSeqBDD 演算を定義することによって，写像付き SeqBDD にも同様に適用できる．これらのアルゴリズムは，接頭辞が一致する部分を同じグラフで表現するものであるため，写像付き SeqBDD は，通常の SeqBDD と同様に接頭辞の共有を行う．

SeqBDD の接尾辞の共有は，同じ文字列集合が一意に表現される性質から，冗長な節点を共有することで実現されている．写像付き SeqBDD でも，同じ文字列集合は一意に表現できるため，SeqBDD と同等以上の共有ができる．

以上より，写像枝を付与した SeqBDD には，少なくとも通常の SeqBDD の接頭辞共有，接尾辞の節点の共有を行うことができる．よって，写像枝を付与した SeqBDD の節点数は，通常の SeqBDD と比べて増加することはない．

4.3 写像を限定した写像枝

前節で示した貪欲な写像枝の手法は節点の削減という点では，優れているが，節点の文字情報を写像として抽出したことによって，写像を表現するために多くのメモリを必要としてしまう．このため，貪欲な写像枝の手法を用いたとしても，常にメモリを削減できるとは限らない．さらに，*Apply* 等の演算操作を行う都度，写像操作によって節点の文字ラベルを読み替えなければならないため，演算の処理速度の効率では既存の SeqBDD に劣る．本節では，使用する写像を限定し，写像情報を貪欲な写像枝の手法よりも小さくし，写像枝付与によるメモリ使用量や処理速度のオーバーヘッドを小さくする高速な処理が可能な写像枝の手法を示す．

高速な処理が可能な写像枝の手法によって作られた写像付き SeqBDD は，SeqBDD に付与することができる写像の種類を 2 種類に限定する．これらの写像のうちの 1 つは，恒等写像 I であり，もう 1 つは，写像 f である．f^{-1} が f の逆写像を表すものとすると，写像 f は全単射かつ，$f = f^{-1}$ である．このような f と I の写像だけを用いると，写像の情報は全体で 1 つだけ記憶しておくだけでよくなり，写像 f を適用するグラフを指す枝の写像枝にのみ，写像を表すマークを付与することで，写像 f が付与されている枝なのか，恒等写像が付与されている枝なのかを区別することができる．よって，通常の SeqBDD と比べて，写像を限定した写像枝をもつ SeqBDD は，枝 1 つあたり 1 ビットのメモリ増加だけで実現できる．

この手法は，BDD の否定枝による *Apply* 演算の高速化手法と同様に，通常の SeqBDD では異なる 2 つの演算を写像操作によって変換し，*Apply* の演算キャッシュの共有を行うことができる．

5. 実験結果と考察

5.1 実験環境

提案した 2 つの写像枝の手法を適用した SeqBDD の実装を行い，3 つの実験を行った．実験環境のコンピュータは，CPU が Intel Core i7-920 @2660MHz，主記憶が DDR3-1333 24GB，OS は Ubuntu8.04 64bit であった．貪欲な写像枝の手法については，Java 言語で実装した (実験 1)．写像を限定した写像枝の手法は，C/C++言語でコンパイルし，gcc/g++の version 4.3.4 で-O3 オプションを用いてコンパイルした (実験 2, 3)．

表 2　実験 1: 貪欲な写像枝を用いた SeqBDD の節点数とメモリ使用量

入力データ		SeqBDD		貪欲な写像枝を用いた SeqBDD	
データ種別	(n, L, Σ)	節点数	メモリ使用量 (byte)	節点数	メモリ使用量 (byte)
RANDOM	(100, 10, 4)	377	4147	268	4293
RANDOM	(100, 10, 26)	486	5346	304	4870
RANDOM	(10000, 10, 4)	6951	76461	6864	109830
RANDOM	(10000, 10, 26)	11842	130262	9510	152167

表 3　実験 2: 写像を限定した写像枝を用いた SeqBDD に対する節点の数え上げ操作の結果

入力データ		SeqBDD		写像を限定した写像枝を用いた SeqBDD	
データ種別	(n, L, Σ)	節点数	実行時間 (秒)	節点数	実行時間 (秒)
RANDOM	(10,000, 20, 4)	83308	0.002843	78292	0.002719
RANDOM	(10,000, 20, 26)	1415222	0.088893	1396895	0.087394
RANDOM	(100,000, 20, 4)	502450	0.047507	453796	0.043435

5.2 実験結果と考察

5.2.1 実験 1

系列集合を表現するために必要な節点数とメモリ使用量を貪欲な写像枝を用いた SeqBDD と通常の SeqBDD とで比較した．結果を表 2 に示す．入力データは，$L = 10$ でランダムに生成した系列の集合である．アルファベットサイズは，4 と 26 のものについて実験を行っており，それぞれゲノムの系列集合と英単語の集合を入力として想定している．貪欲な写像枝を用いた SeqBDD の処理時間のオーバーヘッドを考えると，既存の SeqBDD よりも高速に構築できることはないため，構築時間は計測していない．

表 2 から，いずれの入力データに対しても，貪欲な写像枝を用いた SeqBDD を用いると節点数を削減することができていることがわかる．しかし，$L = 10, \Sigma = 26$ のデータを除き，貪欲な写像枝を用いた SeqBDD を用いるとメモリ使用量が増加している．このことから，節点数の削減が必ずしもメモリ使用量の削減につながっておらず，節点数を削減するための写像情報の表現に多くのメモリが必要であることが分かる．

5.2.2 実験 2

写像を限定した写像枝を用いた SeqBDD について，節点の数え上げ操作を行った．実験結果を表 3 に示す．実験に用いたデータは，いずれもランダムに生成したものである．

表 3 から，いずれのデータでも提案手法の写像枝を用いた SeqBDD が効率の良い節点数，実行時間で動作している．実験では，節点に対する数え上げ操作を行ったが，

表 4　実験 3: 写像を限定した写像枝を用いた SeqBDD に対する 2 つの和集合演算の結果

入力データ		SeqBDD		写像を限定した写像枝を用いた SeqBDD	
データ種別	(n, L, Σ)	節点数	実行時間 (秒)	節点数	実行時間 (秒)
RANDOM	(10,000, 20, 4)	83434	0.011063	78360	0.012137
RANDOM	(100,000, 30, 4)	1500281	0.132392	1449988	0.140176
MAPPING	(100,000, 30, 4)	236712	0.164000	236712	0.101529

数え上げ操作のみならず，文字の数え上げなどの 1 つの SeqBDD に対する演算も同様に効率よく実現可能であることが予想される．

5.2.3　実験 3

写像を限定した写像枝を用いた SeqBDD の節点数と実行時間を通常の SeqBDD と比較した．結果を表 4 に示す．入力データは，4 つの文字列集合であり，2 つの文字列集合に対する *Apply* による和集合演算を最初の 2 つの文字列集合と後の 2 つの文字列集合に対して適用した．表 4 中の各データ種別の詳細を示す．

- RANDOM: 全ての文字がランダムに出現するデータ．
- MAPPING: 1 つ目の文字列集合と 3 つ目の文字列集合が写像操作により互いに一致し，かつ，2 つ目の文字列集合と 4 つ目の文字列集合が写像操作により互いに一致するようなデータ．

表 4 から，写像を限定した写像枝を用いた手法は，RANDOM のような単純な入力に対しては通常の SeqBDD よりも効率よく動作するとは限らないことが分かる．しかしながら，MAPPING のケースでは，写像を限定した写像枝を用いた手法が，高速に動作している．これは，最初の和集合演算の結果を 2 度目の和集合演算で再利用でき，2 度目の和集合演算の処理時間を短縮できているからだと考えられる．

6.　おわりに

本稿では，BDD や BDD から派生したデータ構造に対する属性枝の手法として新しく写像枝の手法を提案した．本手法は，変数ラベルを読み替える写像枝という枝を付与することで BDD の節点を通常の BDD よりも効率よく共有できる．この共有により，メモリ使用量の削減を行うことができる．さらに，演算キャッシュの効率化によって，*Apply* と呼ばれる BDD 同士の演算の効率化が可能である．

実験では，系列集合を表現する SeqBDD に対して，写像枝を用いた 2 つの手法についてメモリ使用量や処理時間を計測した．貪欲な写像枝を用いた手法では，写像枝を構成するために多くのメモリが必要になり，メモリ使用量をほとんど削減することができなかった．一方，写像を限定した写像枝を用いた手法では，写像枝 1 つあたりのメモリ増加量をほぼ 0 にして確実にメモリ使用量を削減しながらも，一部の演算では，通常の SeqBDD 処理系よりも効率よく高速に動作することを確認した．

今後の課題は，BDD の属性枝として，変数ラベルを読み替えるだけでなく，変数の順序を入れ替える操作を属性として付与することで節点を削減する手法を実現していくことである．

参考文献

[1] Akers, S. B.: Binary Decision Diagrams, *IEEE Trans. Computers.*, Vol. 27, No. 6, pp. 509–516 (online), DOI: http://dx.doi.org/10.1109/TC.1978.1675141 (1978).
[2] Bryant, R. E.: Graph-Based Algorithms for Boolean Function Manipulation, *IEEE Trans. Computers*, Vol. 35, pp. 677–691 (1986).
[3] Minato, S.: Zero-suppressed BDDs for set manipulation in combinatorial problems, *DAC '93: Proceedings of the 30th international Design Automation Conference*, New York, NY, USA, ACM, pp. 272–277 (online), DOI: http://doi.acm.org/10.1145/157485.164890 (1993).
[4] Loekito, E., Bailey, J. and Pei, J.: A Binary Decision Diagram Based Approach for Mining Frequent Subsequences, *Knowledge and Information Systems: An International Journal*, Vol. 24, No. 2, pp. 235–268 (2010).
[5] Minato, S.: π DD: A New Decision Diagram for Efficient Problem Solving in Permutation Space, *Theory and Applications of Satisfiability Testing - SAT 2011* (Sakallah, K. and Simon, L., eds.), Lecture Notes in Computer Science, Vol. 6695, Springer Berlin Heidelberg, pp. 90–104 (2011).
[6] Miller, D. M. and Thornton, M. A.: QMDD: A Decision Diagram Structure for Reversible and Quantum Circuits, *Proceedings of the 36th International Symposium on Multiple-Valued Logic*, ISMVL '06, Washington, DC, USA, IEEE Computer Society, pp. 30–35 (online), DOI: 10.1109/ISMVL.2006.35 (2006).
[7] Yamashita, S., Minato, S. and Michael, M. D.: DDMF: An Efficient Decision Diagram Structure for Design Verification of Quantum Circuits under a Practical Restriction, *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, Vol. E91-A, No. 12, pp. 3793–3802 (2008).
[8] Minato, S., Ishiura, N. and Yajima, S.: Shared binary decision diagram with attributed edges for efficient Boolean function manipulation, *Design Automation Conference, 1990. Proceedings., 27th ACM/IEEE*, pp. 52 –57 (online), DOI: 10.1109/DAC.1990.114828 (1990).
[9] Denzumi, S., Arimura, H. and Minato, S.: Construction of Substring Indices Using Sequence BDDs, The Second Forum on Data Engineering and Information Management (DEIM 2010), E3-4 (2010).

ⓒ 2013 Information Processing Society of Japan