TCS-TR-B-**08-3**

TCS Technical Report

Master Thesis: Studies on Variable Ordering of Zero-suppressed Binary Decision Diagrams for Database Analysis

by

Haruya Iwasaki

Division of Computer Science

Report Series B

March 22, 2008

Hokkaido University
Graduate School of
Information Science and Technology

Email: iwsk@ist.hokudai.ac.jp

Phone: +81-011-706-7681
Fax: +81-011-706-7681

修士論文：データベース解析のためのゼロサプレス型二分決定グラフの変数順序付けに関する研究

岩崎 玄弥
北海道大学 大学院情報科学研究科
コンピュータサイエンス専攻
札幌市北区北 14 条西 9 丁目

2008 年 3 月 22 日

(あらすじ) 近年，パソコンの周辺機器の発達や高速インターネットの普及などにより，我々は自分たちの周りに情報が溢れかえる，いわゆる情報爆発の状態にさらされている．インターネット上の情報は膨大なものとなり，今や世界最大の知の集積体となっている．こうした背景から，膨大な量の情報を適切に処理し，利用者にとって有用な情報を得る，データマイニングの技術が求められるようになってきている．

我々は，これまで大規模なトランザクションデータベースを効率よく処理し，有用な情報を得るための技術について研究を行ってきた．トランザクションデータベースとは，例えば小売店を例に考えると，一人の客が買った品物を一つの行（レコード）で表し，それらを複数組み合わせてデータベースとしたものである．他にも様々な分野で利用されているデータ形式である．トランザクションデータを解析することにより，従来知られていなかった性質や規則を見つけ出せる可能性がある．

これを実現するために，我々は二分決定木をもとにしたデータ構造である「ゼロサプレス型二分決定グラフ」(ZBDD: Zero-suppressed Binary Decision Diagrams) を用いた手法の研究を行ってきた．ZBDD は大規模な組合せ集合データを非明示的に列挙し，効率よく演算処理を行うことができる．ZBDD では変数の順序付けによってサイズに大きな違いが生まれるので，データを効率よく扱うためにはよい変数順序付けを行うことが必須である．

我々は ZBDD の変数の順序付け法として「動的重み付け法」を開発した．これは，データベースの構造情報を用いて，生成される ZBDD のサイズを小さくする変数順序を求める手法である．この手法を用いて種々の実験を行い，よい結果が得られることを確認した．頻度をもとにした簡単な変数順序付けと比較しても，安定的によい結果を得られることが確認できた．また，動的重み付け法にかかる処理時間の改善のために，動的重み付け法自体の改良を行い，場合によっては従来手法の 100 分の 1 以下の時間で変数の順序付けを行えることを確認した．さらに，データベースをサンプリングする手法を提案し実験を行い，こちらも大幅な変数順序付け速度の向上が確認できた．

これらのことにより，今まで扱うことが困難であったパタン（レコードに含まれるアイテムの組み合わせ）をより簡単に扱うことが可能になり，トランザクションデータの解析の範囲を広げることが可能となった．

1 はじめに

本章では，本研究が行われた背景，目的，概要，論文の構成について述べる．

1.1 研究の背景

近年，高速ネットワークと大容量記憶装置の急速な発展によって，データマイニングの研究が盛んになっている．データマイニングとは，大量に蓄積されたデータの中から有用な規則を発見する技術のことである．例えば，小売店の販売データやクレジットカードの利用履歴など様々なデータを解析することによって，例えばスーパーで客が購入した物の傾向

から「ビールを買う客は一緒に紙オムツを買うことが多い」ということがわかったり，クレジットカードにおいては，不正使用時に現れやすい規則を発見することによって不正使用を防止できるようになるかもしれない．これはあくまでも一例であるが，データマイニングの技術の進歩によって，広い範囲で便利で効率のよいサービスが生まれてくるかもしれない．このような理由から，データマイニングの技術に対する需要が高まってきていると思われる．

1.2 研究の目的

こうした背景から，本研究では，大規模なトランザクションデータベースにおいて，パターンの発見から解析にいたる多様な演算を効率よく実現する手法を発見することを目的とする．トランザクションデータベースとは，ある特定の規則を持ったアイテムの組み合わせを 1 つの行（レコード）とし，それを複数束ねてデータベースにしたものである．具体的な例としては，きのこの色や形，においなどを要素としたものや，Web の閲覧履歴をまとめたものなどがある．これらを効率的に解析することで，従来は知られていなかった規則性を見つけ出すことができる可能性がある．これにより，我々の日常生活においても様々な面で利便性の向上などが見込まれる．

1.3 研究の概要

この目的を実現するために，VLSI CAD の分野で大規模論理関数データの表現方法として広く用いられている二分決定グラフ (BDD: Binary Decision Diagrams)[4]，その中でも「ゼロサプレス型 BDD」(ZBDD: Zero-suppressed BDD) [15, 18] と呼ばれるデータ構造を用いる．ZBDD は大規模な組合せ集合データを非明示的に列挙し，効率よく演算処理を行うことができる．また，数式により組合せ集合の演算を記述し，これを ZBDD を用いて計算するプログラム「VSOP」[19] を用いて，ZBDD の変数順序付け法の適用などによりデータを効率よく解析することを目指す．

1.4 論文の構成

本論文では，前節の概要を次章以降で以下の通り詳述する．

第 2 章では，本研究に関するデータマイニングについて述べる．

第 3 章では，本研究で使用する BDD，ZBDD および VSOP の基本的な性質，特徴，仕様などについて述べる．

第 4 章では，本研究の目的の実現のために行う ZBDD の簡単化の手法，具体的には，ZBDD における変数の順序付け法として動的重み付け法を提案し，本手法を用いて行った各種実験結果と考察を述べる．さらに動的重み付け法がどのようなデータベースに対して有効なのか実験した結果と考察を述べる．

第 5 章では，前章で述べた動的重み付け法の処理時間の改善のために行った動的重み付け法の改良点について述べ，旧手法と改善手法の処理時間の比較を行った実験の結果を示し考察を述べる．

第 6 章では，データベースのサンプリングを用いた動的重み付け法の手法を提案し，行った実験の結果を示し考察を述べる．

第 7 章では，本実験を通しての結論と，今後の課題について述べる．

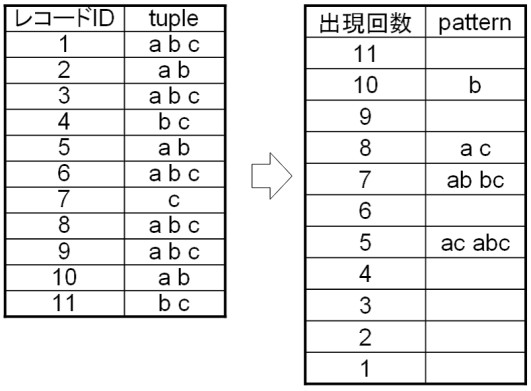
2 データマイニング

本章では，データマイニングについて簡単に触れる．特に頻出パタン抽出問題について，いくつかのアルゴリズムを簡単に説明する．

2.1 頻出パタン抽出問題

頻出パタンマイニング (Frequent Pattern Mining) とは，データベースからアイテムセットの頻度を数える問題で，最も基本的なデータマイニング問題であり，Agrawal 等 [1] による Apriori アルゴリズムの研究に始まり，現在までに様々なアルゴリズムが提案されている [7, 9, 25].

頻出パタンマイニングは，図 1 を例にとって考えると，出現回数が 8 回以上のものを取り出すとすると，その答えは $\{a, b, c\}$ という集合である．このように，元からあるデータというのは統一性がなく，いろいろなものが乱雑に並んでいるのが普通であるが，



レコードID	tuple
1	a b c
2	a b
3	a b c
4	b c
5	a b
6	a b c
7	c
8	a b c
9	a b c
10	a b
11	b c

出現回数	pattern
11	
10	b
9	
8	a c
7	ab bc
6	
5	ac abc
4	
3	
2	
1	

図 1: データベース例

その中から自分がほしいと思っているものを，今回の頻出パタン抽出問題の場合は出現回数であるが，データの中を探索して取り出すのがデータマイニングである．Apriori アルゴリズムは，動的計画法を用いた幅優先探索で頻出パタンを効率よく抽出するアルゴリズムである．Apriori は多くの拡張手法のベースとなっている．また，Eclat や，FP-growth といった手法も開発されている．Eclat は深さ優先探索で頻出パタン抽出を行うアルゴリズムで，メモリ消費が少なく，場合によっては Apriori よりも早くパタンを抽出できることもある．FP-growth はグラフに基づいたデータ構造をしていて，こちらも深さ優先探索を採用している．

しかし，大容量記憶装置の発展によりデータベースが大規模化し，行う処理の回数も膨大になってきており，より効率のよいデータマイニングの手法が求められている．

2.2 データマイニングと頻度表

データベースの解析によって得られた結果，本研究ではアイテムセットの出現頻度について考える．アイテムセットの出現頻度を調べることによって，アイテムの相関関係が見えてくることがあるかもしれない．今回は，タプル頻度表とパタン頻度表を用いる．タプル頻度表とパタン頻度表の説明は 3.4 節で述べる．

3 BDD,ZBDD と VSOP

本章は，本研究で用いるデータ構造やプログラムを紹介するために，文献 [19, 20] を説明したものである．

3.1 BDD

BDD は，図 2(a) に示すような論理関数のグラフによる表現である．一般に，論理関数のそれぞれの変数について，0,1 の値を代入した結果を，二分岐の枝（0-枝/1-枝）で場合分けし，最終的に得られる論理関数の値を 2 値の定数節点（0-終端節点/1-終端節点）で表現すると，図 2(b) のような二分木状のグラフになる．このとき，場合分けする変数の順序を固定し，

- 冗長な節点を削除する （図 3(a)）
- 等価な節点を共有する （図 3(b)）

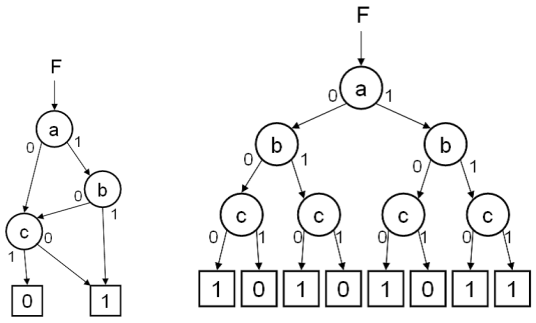
という 2 つの縮約規則を可能な限り適用することにより，図 2(a) のような「既約」な形が得られ，論理関数をコンパクトかつ一意に表せることが知られている [2, 11]．さらに，複数の論理関数を表す BDD の間においても，変数順序を固定すれ互いにグラフを共有することが可能であり（図 4），1 つのメモリ空間の中で多数の論理関数データを扱うことができる．

BDD は，パリティ関数や加算器等を含む多くの実用的な論理関数を比較的少ない記憶量で一意に表現することができる．また，2 つの BDD を入力とし，それらの二項論理演算（AND，OR 等）の結果を表す BDD を直接生成するアルゴリズム [4] が考案されている．このアルゴリズムはハッシュテーブルを巧みに用いることで，データが計算機の主記憶に収まる限りは，その記憶量にほぼ比例する時間内で論理演算を効率よく実行できるという特徴を持っている．(詳細は文献 [5] を参照)

3.2 ZBDD

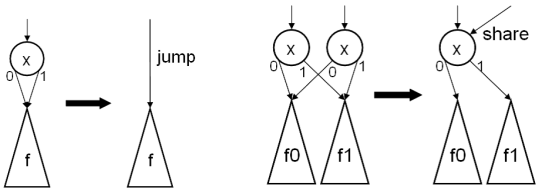
BDD は元々は論理関数を表現するために考案されたものだが，これを用いて組み合わせ集合データを表現・操作することもできる．

組合せ集合とは，「n 個のアイテムから任意個を選ぶ組合せ」を要素とする集合である．n 個のアイテムから任意個を選ぶ組合せには 2^n 通り存在するので，組合せ集合としては 2^{2^n} 通りの取り方がある．例えば，a, b, c, d, e という 5 つのアイテムに関し



(a) BDD (b) 二分決定木

図 2: BDD と二分決定木: $F = (a \wedge b) \vee \bar{c}$



(a) 冗長節点の削除 (b) 等価な節点の共有

図 3: BDD の簡約化規則

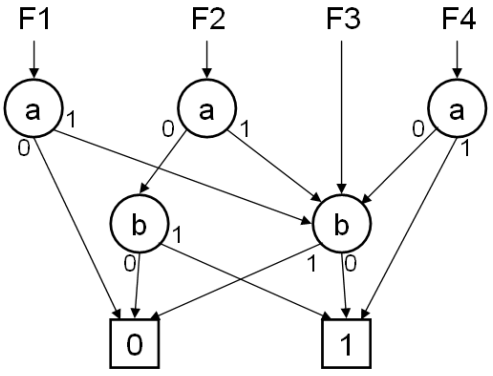


図 4: 複数の BDD の共有化

a	b	c	F	
0	0	0	0	
0	0	1	1	→ c
0	1	0	0	
0	1	1	1	→ bc
1	0	0	0	
1	0	1	0	
1	1	0	1	→ ab
1	1	1	1	→ abc

論理関数として見た場合:
$F = ab + \bar{a}c$
組合せ集合として見た場合:
$F = \{ab, abc, bc, c\}$

図 5: 組合せ集合と論理関数の対応

て，$\{ab, e\}, \{abc, cde, bd, acde, e\}, \{1, cd\}, \emptyset$ は，いずれも組合せ集合の一例である．(本研究では “1” は空の組合せ要素を表し，“$\emptyset$” は空の集合を表すこととする.）組合せ集合は，組合せ問題の解集合を表現する基本的・汎用的なデータ構造であり，実問題においては，スイッチの ON/OFF の組合せ等，様々な局面で現れる．

ある 1 つの組合せ集合は，1 つの論理関数（特殊関数と呼ばれる）に対応づけることができる．例えば，図 5 は $ab + \bar{a}c$ という論理関数を表す真理値表であるが，見方を変えると，$\{ab, abc, bc, c\}$ という組合せ集合も表現している．特殊関数を BDD で表すことにより，組合せ集合を非明示的に表現することができる．このとき，論理関数の AND/OR 演算は，集合の交わり (intersection)/結び (union) の演算にそのまま対応するので，多数の要素を含む集合同士の演算を，BDD 処理系でまとめて実行することが可能となる．たとえ要素数が非常に多くても，類似する組合せが多ければ，部分的に共通する組合せがグラフ上で共有されて，記憶量や計算時間が大幅に（ときには指数関数的に）削減される場合がある．さらに，組合せ集合に特化した「ゼロサプレス型 BDD」(ZBDD)[15] を用いると，より簡潔な表現が得られ，一層効率よく扱うことができる．

ZBDD では，冗長な節点を削除する簡約化規則が通常の BDD と異なる．通常の BDD では，図 3(a) のように，0-枝と 1-枝が同じ節点を指しているときに，この節点を冗長として取り除くのに対し，ZBDD ではそのような節点を取り除くことはせず，その代わりに，図 6 のように 1-枝が 0-終端節点を直接指している節点を取り除く，という規則になっている．

ZBDD では，0-枝はその節点のアイテム x を含まない部分集合（コアファクタ）を，1-枝はアイテム x を含む部分集合（コアファクタ）を表している．もしも 1-枝が 0-終端節点を直接指しているとすると，そ

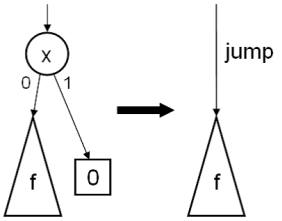


図 6: ZBDD の簡約化規則

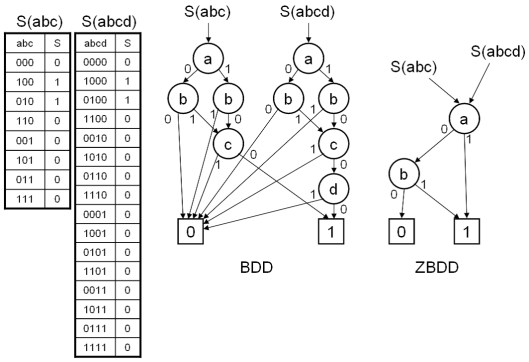


図 7: ZBDD の縮約効果

の組合せ集合には，アイテム x は一度も出現しないということになる．そのような関係ないアイテムに関する節点は冗長とみなして削除し，0-枝側に直結させるというのが，ZBDD の考え方である．この簡約化規則により，組合せ集合に影響を与えない（一度も選ばれることのない）アイテムに関する節点が自動的に削除されることになり，通常の BDD よりも効率よく組合せ集合を表現・操作することができる．

以下に ZBDD の特徴を列挙する．

- ZBDD を用いると，組合せ集合に無関係な（一度も選ばれることのない）アイテムに関する節点が自動的に取り除かれる．通常の BDD ではそのような節点は残ってしまい，せっかくの BDD の共有化のメリットが損なわれる場合がある．図 7 に例を示す．組合せ集合 $S(abc)$ と $S(abcd)$ は，想定するアイテム変数のドメインの違いを除けば等価な組合せ集合である．このとき，アイテム c, d は，この組合せ集合には関与していないにも関わらず，通常の BDD 表現では c, d に関する節点が残ってしまう．一方，ZBDD では，c, d に関する節点は自動的に取り除かれ，$S(abc)$ と $S(abcd)$ は同じ形の ZBDD となる．

- 特に疎な組合せからなる集合に対して ZBDD は顕著な効果がある．例えば 1,000 個のアイテムから数個を選ぶような組合せでは，ほとんどの入力変数は 0 である．このような場合にゼロサプレス簡約化規則が非常に効果的である．

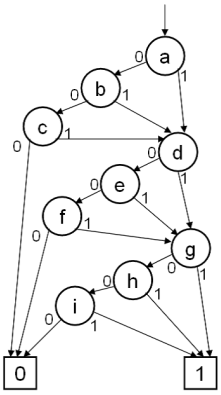


図 8: $(a+b+c)(d+e+f)(g+h+i)$ の展開形

- ZBDD を用いると，積和形の数式をコンパクトに表現することができる．例えば，$(a+b+c)(d+e+f)(g+h+i)\cdots$ を展開して得られる積和形は，アイテム変数の個数に対して指数の項数となるため，明示的に積和形データを保持することは従来困難であった．ZBDD ならばこれを図 8 のように変数の個数に比例する節点数で表現できる．このように，項数が百万を超えるような大規模な積和形データであっても，ZBDD で現実的に処理可能となる場合がある．

- グラフの最上位節点から，1 のラベルを持つ終端節点へ至る経路の数が，集合の要素数と完全に一致する．（通常の BDD では必ずしも一致いない．）

- ZBDD は，もしも等価な節点が存在せず共有が全く行われなかった場合，線形リスト表現でアイテム組合せを羅列したのとほぼ同じデータ構造となる．（図 9 に例を示す．最上位節点から 1 の終端節点に至る経路が 3 つあり，それぞれが集合の組合せ要素を線形リストで羅列表現したものになっている．）つまり，ZBDD の記憶量は最悪の場合でも，データを明示的に羅列した場合に要する記憶量（の定数倍）を超えることはない．（通常の BDD では，疎な組合せ集合を扱う場合にオーバーヘッドが大きいが，ZBDD ではオーバーヘッドがほとんどない．）

ZBDD 処理系では，表 1 に示すように，基本的な演算として，“0”, “1” の定数集合を返す演算，与

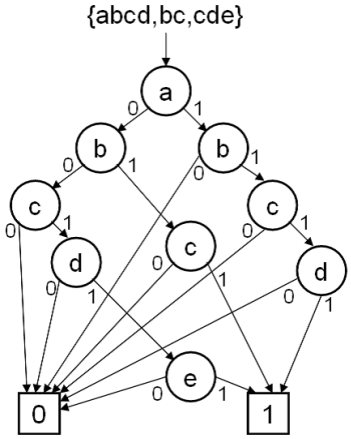


図 9: 節点共有ができない例

表 1: ZBDD 処理系の基本演算

“$\emptyset$”	空集合（0-定数節点）
“1”	空の組合せ項を持つ集合（1-定数節点）
P.top	P の最上位節点のアイテム番号を返す
P.offset(v)	アイテム v を含まない項集合
P.onset(v)	アイテム v を含む項集合から v を取り除いたもの
P.change(v)	各項のアイテム v の有無の反転
$P \cup Q$	P と Q の結び (union)
$P \cap Q$	P と Q の交わり (intersection)
$P - Q$	差集合（P にあって Q にないもの）
P.count	P の要素数を数える

えられた ZBDD の特定のアイテム変数に $0, 1$ を代入・操作する offset，onset，change の各演算，与えられた 2 つの ZBDD に関する組合せ集合の結び，交わり，差集合を表す ZBDD を新しく生成する二項集合演算，および要素数を数える演算などが用意されている．ZBDD を構築する場合，まず単一のアイテム変数からなる ZBDD を作っておき，それらの二項集合演算を繰り返し実行することにより，より複雑な組合せ集合を表す ZBDD を構築していく．二項集合演算は，BDD の論理演算アルゴリズムと同様の原理に基づいており，最上位の変数にそれぞれ $0, 1$ を代入して二分木状に展開して計算を行う再帰的なアルゴリズムである．単純に二分木展開を続けると変数の個数に対して指数関数的な時間がかかってしまうが，過去の演算結果を記憶・参照するハッシュテーブルを用いて冗長な計算を省く技法により，演算に関与する ZBDD の節点数にほぼ比例する計算時間と記憶量で，効率よく二項演算を実行できることが知られている．ZBDD の処理系に関する詳細は文献 [15, 18] などに記載されている．

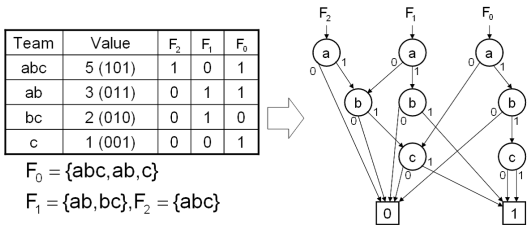


図 10: $(5abc + 3ab + 2bc + c)$ の 2 進ベクトルによる表現

3.3 VSOP

VSOP(Valued-Sum-Of-Products)[19] は，数式により組合せ集合の演算を記述し，これを ZBDD を用いて計算するプログラムである．VSOP は非常に多数のアイテム変数の組合せを要素とする積和集合を扱うことができ，また，単なる集合演算だけでなく，集合の各要素に整数値（係数あるいは重み）を定義し，加減乗除や大小比較などの算術演算を含む数式を処理できることを特徴とする．

ここで，ZBDD を用いて重み付き積和集合を非明示的に表現する方法について述べる．ZBDD は組合せ集合を表すことしかできないため，そのままでは重みを表現できない．これの解決法として，本論文では整数値を 2 進数に分解する方法を用いる．n 個の ZBDD$\{F_0, F_1, \ldots, F_{n-1}\}$ をベクトル状に並べると，最大 $(2^n - 1)$ までの重みを表現することができる．すなわち，整数値の重みを 2 進数で符号化し，最下位ビットが 1 になる組合せ集合を F_0，次のビットが 1 になるような組合せ集合を F_1，という形で F_{n-1} まで並べることにより，図 10 のように，各項の重みを非明示的に表すことができる．

ところで，整数値を 2 進数で符号化する場合，負数の表現方法を考慮する必要がある．代表的な方法として，2 の補数表現と絶対値表現がある．2 の補数表現では，絶対値の小さい負数を表すと上位ビットがすべて 1 となるため，ゼロサプレス型の簡約化規則が効果的に働かないという欠点がある．一方，絶対値表現を用いた場合，各項の正負・大小により加算と減算を分類して別々に処理する必要があり，計算手順が複雑になる．そこで我々は (-2) 進数で符号化する方法を用いている．すなわち，(-2) のべき乗 $(1, -2, 4, -8, 16, \ldots)$ の和で数値を表現する方法で，これでも整数を一意に表現できる．例えば -12 は $(-2)^5 + (-2)^4 + (-2)^2$ と分解して表現できる．(-2) 進数を用いることにより，正負に関わらず上位ビットが 0 になるので，ZBDD の簡約化規則が有効に働

き，意味のない上位ビットに関する節点は自動的に削除される．

さらに，我々の実装では，ZBDD のベクトルを 1 つの ZBDD に束ねるために，特別なアイテム変数を定義している．グラフの上位に 20 個の変数を確保することにより，最大 2^{20}（約 100 万）桁までの ZBDD ベクトルを束ねて 1 つの ZBDD として表すことができ，実質的に無制限の多倍長表現が可能になっている．

重み付き積和集合を計算するには，まず単一のアイテム変数を表す ZBDD，および定数を表す ZBDD を生成し，これらの間で加減乗除の算術演算を繰り返し適用して，より複雑な積和集合を生成していく．以下に文献 [19] に示されている各演算アルゴリズムについて簡単に述べる．

(**変数乗算**)　重み付き積和集合 F にアイテム変数 v を掛ける場合，F の各項に v を付加するだけであり，これは ZBDD の offset,onset,change 演算を適宜用いて簡単に実行できる．計算時間は v より下位にある節点の個数に比例する．

(**定数乗算**)　F に整数定数 c を掛ける場合，F の各項の重みを c 倍するという操作になる．c が (-2) のべき乗数であれば，F の各桁の ZBDD をシフトするだけなので，節点数には依存せず，桁数に比例する時間内で実行できる．一般の c に対しては，c をビットごとに分解してそれぞれについて F のシフトを計算し，次の算術加算を用いて合計することになる．

(**加減算**)　2 つの式の加算 $(F+G)$ は，F と G の対応する項同士の重みの和を計算し，それを重みとする新たな積和集合を生成する操作である．例えば $F = ab + 2bc - 3c, G = 3ac - 2bc + c$ とすると，$(F+G) = ab + 3ac - 2c$ となる．基本アルゴリズムを示す．まず，F と G の各桁ごとの共通項 $(F \cap G)$ を抽出し，もしこれが空集合（共通項がない）ならば，桁上がりの必要性がないということを表しているので，F と G の各桁ごとのマージ $(F \cup G)$ を計算すれば，それがそのまま算術加算の結果となる．一方，$(F \cap G)$ が空でないならば，それが F と G に共通して含まれる桁上がり項を表しているので，それを 2 倍して再度加えればよい．2 倍することにより新たに共通項が発生する可能性があるので，共通項がなくなるまでこの操作を繰り返す．

我々の実装では，(-2) 進数による符号化を用いているので，シフト演算は 2 倍ではなく (-2) 倍になる．したがって，2 回目の繰り返しは $S - (-2 \cdot C)$ となり，加算ではなく減算を呼び出すことになる．逆に引き算の桁借りは，(-2) 倍して足すことになる．つまり加算と減算を交互に呼び出しあう形で，両者を実現できる．

(**直積乗算**)　ここでいう直積 $(F \times G)$ とは，F の各項と G の各項から 1 つずつ選ぶ組合せすべてについて変数乗算・定数乗算を行い，その総和の式を求める演算である．ZBDD の最上位のアイテム変数を v とすると，F と G は v を含むか否かにより

$$F = v \cdot F_1 \cup F_0, G = v \cdot G_1 \cup G_0$$

と分解できる．このとき $(F \times G)$ は，

$$((F_1 \times G_1) + (F_1 \times G_0) + (F_0 \times G_1)) \times v + (F_0 \times G_0)$$

と展開でき，それぞれの部分乗算は再帰的に計算できる．これを繰り返し，最終的に自明な式まで分解されたところで結果が得られ，それらをまとめて全体の結果とする．このアルゴリズムでは，最悪の場合，変数の個数に対して指数回の再起呼び出しが必要となるが，計算結果を一時記憶するハッシュテーブルを用意して，部分乗算の計算結果をテーブルに登録することによって，アルゴリズムを高速化できる．再起呼び出しのたびにテーブルを検索し，過去に同じ計算が行われていれば，その結果をコピーして用いることにより，無駄な式展開を避けることができる．この技法によれば，計算時間は ZBDD の節点数によって決まり，式の項数には直接依存しない．

最後に VSOP の仕様を述べておく．VSOP は C-Shell と似たインタフェースを持ち，キー入力による対話的処理と，ファイルからのバッチ処理の両方が可能である．プログラムは，C，C++および yacc で記述されており，32 ビット，または 64 ビットの Linux マシンで動作する．

VSOP では，アイテム変数（小文字で始まる英数字列で表現）と，プログラム変数（大文字で始まる英数字列で表現）の 2 種類の変数を使用する．アイテム変数は，求めようとする重み付き積和集合を一時的に保存する記憶場所を表す．一方，プログラム変数は，計算結果を一時的に保存する記憶場所を表す．アイテム変数とプログラム変数を用いて，複数の式からなる多段の数式を記述することができる．計算結果は，プログラム変数を含まない，アイテム変数だけからなる数式として出力される．VSOP ではアイテム変数は最大 65510 個まで，プログラム変

```
***** VSOP calculator <v0. 95> *****
vsop> symbol a b c d e
vsop> F = (a + 2b)(c + d)
vsop> print F
  a c + a d + 2 b c + 2 b d
vsop> print /rmap F
  a b : c d
      |    00    01    11    10
   00 |     0     0     0     0
   01 |     0     2     0     2
   11 |     0     0     0     0
   10 |     0     1     0     1
vsop> G = (2 a - d)(c - e)
vsop> print G
  2 a c - 2 a e - c d + d e
vsop> H = F * G
vsop> print H
  4 a b c d - 4 a b c e + 4 a b c - 4 a b d e
   + a c d e - 2 a c e + 2 a c - a d e
   + 2 b c d e - 4 b c d + 2 b d e
vsop> print /count H
    11
vsop> print /size H
    24 (35)
vsop> quit
```

図 11: VSOP の実行例

数は無制限に使用できる．重みの記述は 10 進整数を用いる．扱える整数値の最大桁数に制限はない．

VSOP では，加減算，直積乗算，除算，大小比較演算，選択演算子「? :」等，種々の演算を実行できるが，if-then-else や while-do 等の分岐・繰り返し構造はサポートしていない．ただし，別のプログラミング言語（Shell や Perl 等）を用いて，制御構造を展開した VSOP スクリプトを生成し，これをパイプライン的に VSOP に入力することにより，より複雑な演算シーケンスを実行できる．

VSOP では，人手で読み書きする程度の数式に対しては，ほとんど瞬時に計算結果が得られる．ZBDD の節点数は主記憶サイズに応じて，数千万個まで処理することができ，かなり複雑な問題にも対応できる．計算結果は，積和形表示や整数値カルノー図等の形式で表示することができる．図 11 に簡単な実行例を示す．/size は節点数，/count は式に現れる組合せの個数を，また，/plot を実行することで BDD の形を図示することができる．

表 2: 代表的なデータマイニング例題の特徴

Data name	$\#I$	$\#T$	$total\|T\|$	$avg\|T\|$	$avg\|T\|/\#I$
T10I4D100K	870	100,000	1,010,228	10.1	1.16%
T40I10D100K	942	100,000	3,960,507	39.6	4.20%
chess	75	3,196	118,252	37.0	49.30%
connect	129	67,557	2,904,951	43.0	33.33%
mushroom	119	8,124	186,852	23.0	19.32%
pumsb	2,113	49,046	3,629,404	74.0	3.50%
pumsb_star	2,088	49,046	2,475,947	50.5	2.42%
BMS-POS	1,657	515,597	3,367,020	6.5	0.39%
BMS-WebView-1	497	59,602	149,639	2.5	0.51%
BMS-WebView-2	3,340	77,512	358,278	4.6	0.14%
accidents	468	340,183	11,500,870	33.8	7.22%

3.4 VSOP によるタプル頻度表，パタン頻度表の作り方

データベースの例として，国際会議 FIMI-2003 のベンチマークデータ [8] から抜粋したものを表 2 に示す．$\#I$ はアイテム数，$\#T$ はタプル数，$total|T|$ はタプル長（タプルに含まれるアイテムの数）の総和，$avg|T|$ は平均タプル長，$avg|T|/\#I$ はアイテムの平均出現頻度である．タプル頻度表とは，与えられたトランザクションデータの中に，同じタプル（アイテムの組合せ）が何回出現したかを，タプルごとに数えて表にしたものである．通常の組合せ集合では，組合せパターンの有無のみに着目し，重複は考慮しないが，実際のデータベースでは，同じタプルが複数回出現することがしばしばあり，その出現回数を数えることが必要な場合がある．例を挙げると，表 2 にある BMS-WebView-1 データに含まれる 59,602 個のレコード中に，最も頻度が高いタプルは 1,533 回出現しており，上位 10 個のタプルで総計 8,404 回（タプル全体の 14%）出現している．図 12 に，簡単なデータベースの例とそのタプル頻度表との関係を示す．このように，タプル頻度表を生成することによって，コンピュータにとっても人間にとってもデータベースが扱いやすくなる．

ZBDD を用いてタプル頻度表を非明示的に表現する方法について述べる．3.3 節で説明したことと重複してしまうが，もう一度説明する．ZBDD は組合せ集合（すなわちタプルの有無）を表すことしかできないため，単純な ZBDD では出現回数を記憶できない．そこで，図 10 のように，n 個の ZBDD$\{F_0, F_1, \ldots, F_{n-1}\}$ をベクトル状に並べ，最大 $2^n - 1$ までの出現回数を表現することとする．すなわち，出現回数を 2 進数で符号化し，最下位ビットが 1 になる（=奇数回出現する）タプルの集合を F_0，次のビットが 1 になるような出現回数のタプルの集合を F_1，という

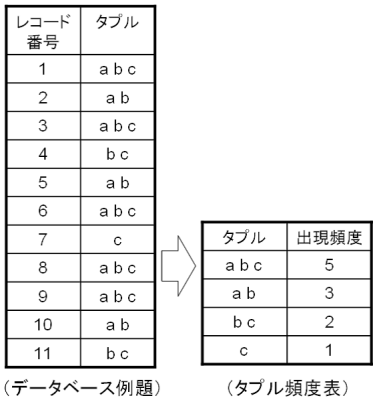


図 12: データベース例題とタプル頻度表

形で，F_{n-1} まで並べることにより，各タプルの出現回数を非明示的に表すことができる．例えば，図 10 では，$F_0 = \{abc, ab, c\}, F_1 = \{ab, bc\}, F_2 = \{abc\}$ と分解されている．

トランザクションデータのファイルを読み込んで ZBDD のベクトルを作るには，データベースからレコードを 1 個ずつ読み込み，そのタプル情報を頻度表に累算していけばよい．具体的には，読み込んだタプルのアイテム組合せのみを表す組合せ集合 T を表す ZBDD を作る．これは，まず “1”（アイテム 0 個のタプルを表す集合）を作り，これに Change 演算を繰り返し適用して必要なアイテムを付加することで作成できる．すなわち，タプルに出現するアイテム数に比例する時間で作成可能である．次に，F_0 と T を比較し，共通する部分がなければ T を F_0 に加えて終わり，もしも共通部分があれば F_0 から T を引き去り，次の F_1 に桁上げさせる．このようにして，桁上がりがなくなるまで上位の桁に波及していく．この頻度表 F へのタプル T の累算を F.add(T) で表す．

F.add(T) の計算時間を分析すると，各桁ごとの処理は定数回の二項集合演算で実現できるので，関与する ZBDD の節点数にほぼ比例する．そして桁上げ処理を繰り返す回数は，出現頻度の最大値の log で抑えられる．また，使用記憶量は，関与する ZBDD の節点数に比例する．

以上のタプル累算演算処理を，すべてのデータレコードについて繰り返し実行すれば，タプル頻度表が完成する．データベース D からタプル頻度表 F_T を作る手続きは次のようになる．

```
1 2 3 4 5
1 3 5
2 4 5
2
3 4
```

図 13: データの例

```
symbol x1 x2 x3 x4 x5

S = 0
S = S + x1 x2 x3 x4 x5
S = S + x1 x3 x5
S = S + x2 x4 x5
S = S + x2
S = S + x3 x4

print /size S
print /count S
```

図 14: VSOP スクリプトの例

$F_T = 0$
forall $T \in D$ do
  $F_T = F_T$.add(T)
return F_T

この処理を実際に VSOP を用いて行った例を示す．扱うデータは図 13 とする．

このデータを VSOP で処理するスクリプトに変換すると図 14 になる．

このスクリプトの各行では各タプルを S に加えている．それを繰り返すことにより，最終的な頻度表を生成する．

3.2 節で述べたように，ZBDD の節点数は，最悪の場合でも組合せ集合データを明示的に列挙したときの記憶量を超えない．したがって，ZBDD のベクトルでタプル頻度表を表した場合，データベースの総文字数と 2 進符号桁数の積に比例する上界が存在するため，ZBDD の大きさが指数爆発的に増大することはない．そして，部分的に共通のパターンをもつタプルが多ければ，ZBDD のグラフが共有され，コンパクトな表現が得られる．

いったん，ZBDD でタプル頻度表を作ってしまえば,「出現回数 α 以上のタプル集合を取り出せ」という問題でも，α を 2 進数で表し，これとの大小比較

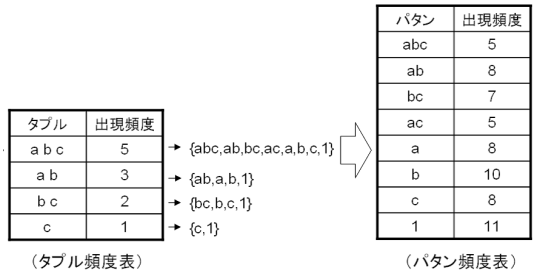


図 15: タプル頻度表とパタン頻度表

をビットごとの ZBDD の集合演算で記述すれば，グラフのサイズに比例する時間で容易に計算できる．

次に，パタン頻度表の生成について述べる．パタンとは，トランザクションデータの各タプルの中に出現するアイテムの部分集合のことを呼ぶ．パタン頻度表は，トランザクションデータに含まれるすべてのパタンについて，その出現回数を数え上げて表にしたものである．タプル頻度表とパタン頻度表との関係を図 15 に示す．

一般に，k 個のアイテムからなるタプルは 2^k 個のパタンを含んでいるので，パタン頻度表はタプル頻度表よりもはるかに大きなデータ量となり，小規模な例題を除けば，完全なパタン頻度表を作ることは困難である．そこで実用的な問題として，適当なしきい値 α より多く出現する頻出パタンの集合を，現実的な計算時間と記憶量で求めるアルゴリズムが盛んに研究されている．

ZBDD を用いる場合，多数の類似するパタンをグラフで共有してコンパクトに表現できるため，従来不可能だと思われていた完全なパタン頻度表を，ある程度の規模まで現実的に生成できる可能性がある．例えば，図 16 は，五つのアイテムからなるタプル $T = abcde$ と，それに含まれるすべてのパタン集合 $P = \{1, a, b, c, d, e, ab, bc, cd, abc, \ldots, abcde\}$ を，それぞれ ZBDD で表現したものである．これを見ると，2^k 個のパタン集合わずか k 個の節点で非明示的に表現できることがわかる．つまり，各タプルに含まれるパタンを列挙する ZBDD は，タプルに出現するアイテム数に比例する時間で生成可能である．そして，タプル頻度表を作るのと全く同様に，2 進数で符号化した ZBDD のベクトルに累算していけば，パタン頻度表を生成できる．

以上をまとめると，データベース D からパタン頻度表 F_P を作る手続きは次のようになる．

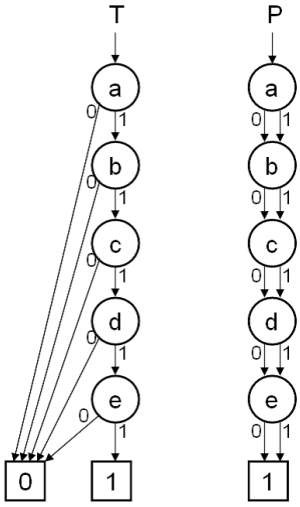


図 16: タプルとそれに含まれるパタン集合を表す ZBDD

```
symbol x1 x2 x3 x4 x5

S = 0
S = S + (1+x1)(1+x2)(1+x3)(1+x4)(1+x5)
S = S + (1+x1)(1+x3)(1+x5)
S = S + (1+x2)(1+x4)(1+x5)
S = S + (1+x2)
S = S + (1+x3)(1+x4)

print /size S
print /count S
```

図 17: VSOP スクリプトの例

$F_P = 0$
forall $T \in D$ do
  $P = T$
  forall $v \in T$ do
    $P = P \cup P.\mathrm{onset}(v)$
  $F_P = F_P.\mathrm{add}(P)$
return F_P

これも図 13 のデータを用いて，VSOP スクリプトの例を示す．パタン頻度表を生成するための VSOP スクリプトは図 17 になる．

ここで，`(1+x1)(1+x2)(1+x3)(1+x4)(1+x5)` というのは，$\{1, x1, x2, x3, x4, x5,\ x1x2, x1x3, \ldots, x1x2x3x4x5\}$ というパタン集合を表している．タ

プルの場合と同様に，これを累算していくことで最終的な結果を得る．

パタン頻度表を生成するための計算量は，タプル頻度表生成の場合と同様に，ZBDD の節点数に依存する．ただし，パタン頻度表の場合，データを明示的に列挙したときの記憶量が，アイテム数に対して指数関数的であるので，ZBDD の節点数を抑える上界が非常に大きい．そのため，パタン集合を累算していく過程で，タプル頻度表の場合よりも急速に ZBDD 節点数が増大する．したがって，ある程度の規模の問題までしか，この方法は適用できない．しかし，もし完全なパタン頻度表を生成できてしまえば非常に強力であり，その後は様々なデータマイニングの解析処理を，ZBDD 処理系の演算により効率よく行うことが可能となる．どの程度の規模まで実際にパタン頻度表を生成できるのかは，非常に興味深い問題である．

なお，参考までに，上記で述べたように，データベースのレコードごとにパタン集合を作って累算していく方法のほかに，いったん，タプル頻度表 F_T を完成させてから，タプル頻度表全体に ZBDD の演算処理を適用して，パタン頻度表 F_P を作り出す方法もある．

$F_P = F_T$
forall $v \in F_T$ do
  $F_P = F_P$.add(F_P.onset(v))
return F_P

どちらが早く計算できるかは今後の検討課題だが，いずれの方法でも最終的に得られる ZBDD の形や節点数は同じである．

3.5 ZBDD-growth 法による頻出パタン生成

パタン頻度表は生成できれば強力であるが，中規模以上のベンチマーク例題に対しては，ZBDD が巨大になり過ぎて，現実的な記憶量では表現することができない．しかし，ある一定の頻度以上の頻出パタン集合だけを抽出するのであれば，最小頻度のしきい値 α を大きくしていけばパタン数が単調減少するので，適度に大きい α を与えれば ZBDD を構築することが可能となる．そこで我々の研究グループでは，ZBDD のベクトルで表現されたタプル頻度表から，パタン頻度表を経由せずに，直接，頻出パタン集合を表す ZBDD を生成するアルゴリズム「ZBDD-growth 法」を開発した [21]．これにより，中規模以上のベンチマーク例題でも頻出パタン集合を表す ZBDD を生成できるようになった．

このアルゴリズムではまず，タプル頻度表で使われているアイテムの 1 つ v を選び出す．これは ZBDD 処理系では最上位のアイテムを取り出す操作となる．次に，入力として与えられたタプル頻度表を，v を含む部分と v を含まない部分を表す 2 つの頻度表に分解する．この演算は ZBDD 処理系では cofactor 演算と呼ばれ，v が最上位アイテムであれば節点の 0 枝と 1 枝を参照するだけで実行できる．ZBDD-growth 法は，頻度表をサブ頻度表に分解して，それぞれに対して再起呼び出しを行うアルゴリズムとなっている．再起呼び出しの構造は二分木状になるので呼び出し回数はアイテム数に対して指数関数的となる．しかし，ZBDD 演算の常套手段として，ハッシュテーブルを用いた演算キャッシュを用意しておいて，等価な ZBDD に対する演算が 2 度現れたら再起呼び出しを打ち切って，即座にキャッシュの内容を返す，という高速化手法を用いることによって，呼び出し回数は，ZBDD 節点数にほぼ比例する回数に抑えられる．

表 3 に文献 [21] の実験結果を示す．実験に用いた PC は，Pentium-4 2.8GHz，主記憶 1.5GByte，OS は SuSE Linux 9 を使用している．この PC では最大 2,000 万節点までの ZBDD を主記憶上に構築し，演算操作を実行することが可能である．例題は FIMI2003 ベンチマーク [8] から選んだものである．実験結果を見ると，“mushroom”では閾値 α が 1，つまり全てのパタンを含む ZBDD をわずか 1.8 秒で生成できていることがわかる．“T10I4D100K”の場合には閾値 α を下げていくにつれて処理時間が増加しているが，この “T10I4D100K”は機械的に乱数を使って合成したデータベースであることが知られており，タプル間にほとんど相関関係がない．このような場合，ZBDD はほとんど節点の共有効果が発揮できず，ZBDD を扱うためのオーバーヘッドのみが現れたものと考えられる．“BMS-WebView-1”に関しては，閾値 α が 30 のときに，35 兆個ものパタンを圧縮して主記憶上に表現することに成功している．このように，データ構造として ZBDD を，ZBDD の生成手法として ZBDD-growth 法を用いることで，これまで扱うことのできなかった頻出パタン集合を扱うことができる可能性がある．

しかし，ZBDD-growth 法を用いて ZBDD を生成する際に用いた変数の順序付けが，その ZBDD を表すのに適切なものでなければ，生成される ZBDD の節点数は最悪の場合，頻出パタンを全て羅列した

表 3: ZBDD-growth アルゴリズムの実験結果

Database name: 最小頻度 α	頻出パタン数	（出力データ） ZBDD 節点数	ZBDD-growth CPU 時間（秒）
mushroom: 5,000	41	11	1.2
1,000	123,277	1,417	3.7
200	18,094,821	12,340	9.7
16	1,176,182,553	53,804	7.7
4	3,786,792,695	59,970	4.3
1	5,574,930,437	40,557	1.8
T10I4D100K: 5,000	10	10	81.3
1,000	385	382	135.5
200	13,255	4,288	279.4
16	175,915	89,423	543.3
4	3,159,067	1,108,723	646.0
1	2,217,324,767	(mem. out)	-
BMS-WebView-1: 1,000	31	31	27.8
200	372	309	31.3
50	8,191	3,753	49.0
34	4,849,465	64,601	120.8
32	1,531,980,297	97,692	133.7
31	8,796,564,756,112	117,101	138.1
30	35,349,566,550,691	152,431	143.9

のと同じだけ必要になってしまう．さらに，ZBDD の節点数の増加に伴って，処理時間も膨大なものとなってしまい，頻出パタンを表す ZBDD を生成することが不可能になってしまう．よって，依然として大規模な ZBDD を生成する際には適切な変数順序付けを行うことが必要不可欠である．

3.6 課題

これまで述べてきたように，ZBDD と VSOP を用いて重み付き積和集合を扱うことで，大規模なデータベースを解析し処理することができるが，データベースが非常に膨大であるために，特に完全なパタン頻度表を生成することは多くの場合困難である．今後も，大容量記憶装置の発展によってデータベースの巨大化が進むことが考えられるので，データベース解析によって生成されるデータをより小さなものにする方法を考えなくてはいけない．次章からその手法として ZBDD の簡単化のための「アイテム変数の順序付け」について述べる．

4 ZBDD の変数順序付け法

本章では，ZBDD において比較的よい変数順序を得るための手法を提示する．

4.1 データベース表現における ZBDD の変数順序付け

前述のように，ZBDD を用いることで頻度表や頻出パタン集合をコンパクトに表現できる可能性があるが，データベースが大規模になると，ZBDD を生成することが困難になる．これを改善する手法として，本稿では,「アイテム変数の順序付け」を考える．すでに述べたように，生成される ZBDD のサイズは変数の順序に大きな影響を受ける．よって，我々の目的は ZBDD のサイズができるだけ最小に近くなる順序付けを見つけることである．

ZBDD の性質上，最悪な順序付けを用いても節点数はパタンを羅列したときの総文字数（各タプルに現れるアイテム数の総和）を超えることがない [20] ため，総文字数が少なければどのような順序付けでも，ZBDD のサイズが膨らむことはない．このことから，アイテム変数の順序付けにより指数関数的な効果が表れるためには，最悪な場合が指数関数的な節点数になる必要があるので，データに関してパタンを羅列したときの総文字数が非常に多くなければならないということがいえる．一般的に，パタンはアイテム数の指数個あるので，アイテム変数の順序付けにより，大きな ZBDD の簡単化効果が得られると期待できる．

本論文で使用した VSOP プログラムでは，データベースに含まれるアイテムをアイテム変数として最初に宣言する．宣言されなかった変数は，そ

の変数が算術式中で使用されたときに，その場で新たに宣言したものとして，最上位に追加される．（以下では，この変数順序を「後出し上位順」と呼ぶ．）例えば，ac, $befg$, ba, h というリストを持つデータベースが与えられた場合，変数の順序付けは，h, g, f, e, b, c, a となる．

我々がこれまで行ってきた ZBDD によるデータマイニングの実験は，ほとんどがこの順序付けを用いている．後出し上位順は，データベースの読み込みと同時に行うことができるので，順序付けにかかる計算コストを無視することができ，経験的には比較的良い順序を得られることが多い．しかし，この順序付けでは，同じ内容のデータベースでも，処理するレコードの順番によって変数の順序が大きく変わるため，結果が不安定で，例題によっては非常に悪い順序付けとなる場合がある．そこで我々は，与えられたデータベース例題の特徴に基づいて，データベース中のアイテムの順序に依存せず，より良い変数順序を安定的に求めるための発見的手法を提案する．この手法は従来の BDD に対して用いられている手法を，データベース解析処理向けに改良したものである．よって，まず従来の BDD に対する手法に関して説明する．

4.1.1 論理回路における BDD の動的重み付け法

通常の BDD の変数順序付け方法については，これまでに VLSI 論理設計の分野で多くの研究がある．一般的に，最適な順序付けを求めることは NP 完全であることが知られている [24] が，現実的な時間で比較的よい順序を求める発見的手法がいくつか提案されている [6, 12, 14]．ここでは，論理回路における BDD の「動的重み付け法」[14] を紹介する．この手法は，論理回路における BDD の入力の順序付けに関して効果を上げている．

通常の BDD においては，グラフの大きさに影響する要素として，次の 2 つの性質が知られている [6].

(1) 局所計算性のある入力の組は，なるべく近い順序にする．これにより，多くのサブグラフが共有できるという傾向がある．例えば 2 進数の加算器の論理回路では，対応するビット同士の変数を並べて配置すると良いことが知られている．

(2) 出力を制御する力の強い入力は上位に配置する．例えば，データセレクタの論理回路では，データ入力を上位に並べた場合は制御入力を上位に並べた場合に比べ，BDD のサイズが指数的に増大してしまう．これは，制御入力の値を先に決定すると，多くのデータ入力を無視できるという性質があるためと考えられる．

これら 2 つの性質を満たすような順序付けを行えばよいのであるが，実際には，2 つの性質が同時に現れて，互いに相反する順序付けを要求する場合があり，これを両立させて最適な順序を求めるのは難しい．また，どんな順序でもそれほど変化の見られない回路もあり，その場合には，順序付けを工夫してもあまり効果が得られない．

以上の考察に基づいて，回路の結線情報から変数の順序付けを行う発見的手法がいくつか知られているが，その中の 1 つとして「動的重み付け法」が提案されている [14]．この方法は，論理回路の出力の論理関数を BDD で表す場合に，回路の結線情報から関数の性質を予想し，最適に近い順序を求めるものである．

動的重み付け法では，まず前述の (2) の性質を満たすため，制御性の高い入力を見つける．これは，以下のような傾向を持つ．

- ファンアウト数が多い（出力に至るパスが多い）．
- 出力までの段数が少ない．
- 出力へ至るパス上のゲートのファンイン数が少ない．

この傾向を評価するために，次に示すような簡単な重み付け法を用いる．

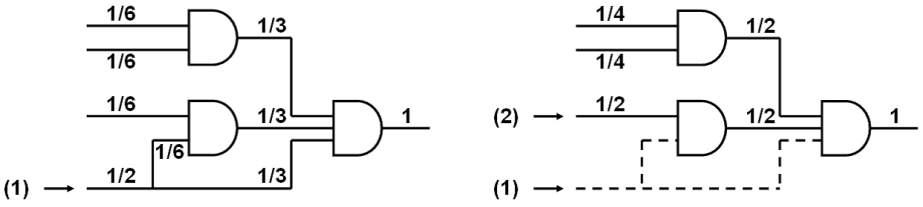
(1) 出力線の重みを 1 とする．

(2) 出力から入力に向かって重みを伝達させる．2 入力以上のゲートでは，出力線の重みをファンイン数で均等に分けて入力線に伝える．

(3) ファンアウトがある信号線で，複数のゲートから重みが伝わるときにはそれらを伝える．

以上の手順で重み付けをした例を図 18(a) に示す．この図では，出力に与えられた重みが三つのファンインに均等に分けられたことが示されている．この重みが最大となった入力が最も出力を制御しやすいとみなし，その入力に最上位の変数を割り当てる．

次に，最上位の変数を決定した後，その下のサブグラフを小さくすることを考える．このとき，すでに決定された入力は 0 か 1 に固定されてしまってい



(a) First (b) Second

図 18: 論理回路における動的重み付け法

ると考えれば，その近くの信号線は，制御性が増大しているはずである．そこで，すでに決定された信号線を切断したと仮定して，改めて重み付けを行う．このように動的に重み付けを行うことにより，前節 (1) の局所計算性も反映することができる．その様子を図 18(b) に示す．以下，これを動的重み付け法と呼ぶ．

動的重み付け法では，順序付けに必要な時間は，入力数 n，回路規模を m として，$O(nm)$ である．順序付けによって，生成される BDD のサイズに指数関数的な違いが見られるので，順序付けにこの程度の時間がかかっても十分有効である．

4.1.2 トランザクションデータベースにおける ZBDD の変数順序付けへの応用

ここまで論理回路における BDD の変数順序付け法に関して説明してきたが，本論文では，これをトランザクションデータベースにおける ZBDD の順序付けに応用する．

まず初めにデータベースの構造を表すグラフを作る．各々のアイテムを表すノードを生成し，次に各々のタプルを表すノードを生成する．最後にデータベース全体を表すノード *All* を作る．さらに，ノード *All* から全てのタプルに対して枝を引き，各々のタプルから，そのタプルに含まれないアイテムに対して枝を引く．論理回路においては，出力から重みを伝えていきゲートの入力数に応じて重みを分配したが，本手法では，

(1) データベース全体の重みを 1 として，これをタプルの数で割った値をそれぞれのタプルの重みとして与える．

(2) 各々のタプルにおいてそのタプルに含まれないアイテムに対して，タプルに与えられた重みをその含まれないアイテムの数で割った値を重みとして伝える．

(3) 複数のタプルから重みが伝わるときにはそれらを加える．

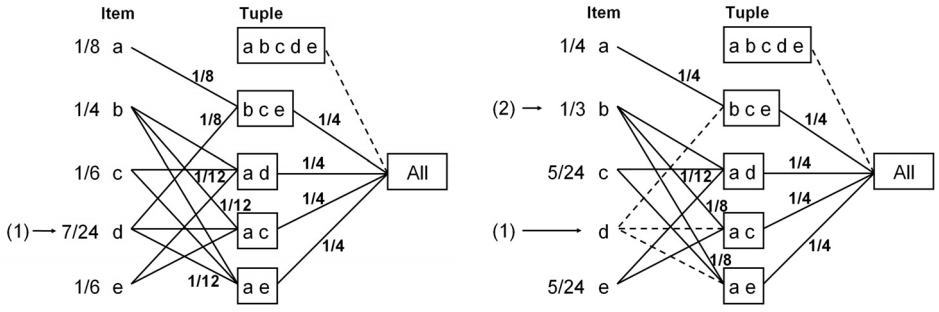
の手順で重み付けを行う．

この (2) の手順において，タプルに含まれるアイテムではなく，タプルに含まれないアイテムに重みを伝えて行くところが，従来の論理回路での動的重み付け法とは大きく異なる点である．タプルに含まれるパタンを抽出する場合に，タプルに含まれるアイテムはパタンを形成するアイテムの組合せに含まれるときと含まれないときがあるが，タプルに含まれないアイテムは当然そのタプルが含むパタンの要素とはなりえず，このことから，タプルに含まれないアイテムは含まれるアイテムより出力に与える影響が大きいと考えられるので，このように重み付けを行う．

このようにして重み付けをした例を図 19(a) に示す．この重みが最大となったアイテムを ZBDD のサイズに特に影響を与えるものとして，変数の最上位に割り当てる．与えられた重みが同じものが複数あった場合は，最も出現が早かったアイテムを選ぶ．

次に，最上位の変数を決定した後の処理について述べる．順序が決定したアイテムに関する線は BDD の場合と同様に取り除き，もしアイテムの順序付けが決まったことで，タプルからアイテムに伸びる線の全てが無くなってしまったタプルがあれば，そのタプルは以降の重み付けの際には存在しないものとして扱う．つまり，存在するタプルには，総タプル数から存在しなくなったタプルの数を引いた数で重み 1 を割った値を重みとして与えることになる．これを図 19(b) に示す．この順序が決定したアイテムに関する線を取り除くという処理によって，取り除かれたアイテムの重みがそのアイテムと関係の深いアイテムに分配されることになるので，局所計算性を反映することができる．

以上の操作を繰り返すことでアイテム変数の順序付けを行う．



(a) First (b) Second

図 19: トランザクションデータベースにおける動的重み付け法

4.2 実験結果

動的重み付け法を適用して行った実験の結果を示す．実験において使用した PC は，Pentium-4 3.0GHz，SuSE Linux 9.3，主記憶 512MByte で ZBDD の最大節点数は 1,000 万個とした．

はじめに，人工的な例題における ZBDD の順序付け法の効果に関して行った実験の結果を示し，次に実際のベンチマーク例題における ZBDD の動的重み付け法に関して実験を行った結果を示す．これらの実験で用いた順序付けは，動的重み付け法，出現頻度の高いアイテムを上位に並べた「高頻度上位順」，逆に出現頻度の低いアイテムを上位に並べた「低頻度上位順」と，アイテムの最初の出現が遅いものから上位に並べた「後出し上位順」の 4 つである．

4.2.1 人工的な例題におけるアイテム変数の順序付け法の評価

まず，ZBDD のアイテム変数の順序付け法に関して，人工的な例題における動的重み付け法の効果を調べる．今回は以下に示すような人工的に生成したデータベースを用いて実験を行う．

a_2	a_3	a_4	$\cdots$	a_n	b_2	b_3	b_4	$\cdots$	b_n
a_1	a_3	a_4	$\cdots$	a_n	b_1	b_3	b_4	$\cdots$	b_n
a_1	a_2	a_4	$\cdots$	a_n	b_1	b_2	b_4	$\cdots$	b_n
	$\vdots$				$\vdots$				$\vdots$
a_1	a_2	a_3	$\cdots$	a_{n-1}	b_1	b_2	b_3	$\cdots$	b_{n-1}

このデータは，$a_1\ a_2\ ...\ a_n\ b_1\ b_2\ ...\ b_n$ という形から a_k と $b_k(k = 1, ..., n)$ を取り除いたものを各レコードとしている．また，このデータベースに関して動的重み付け法を用いた場合のアイテム変数の順序付けと，低頻度上位順に並べた順序付けを以下に示す．

- 動的重み付け法: $a_2b_2a_3b_3a_4b_4 \cdots a_nb_na_1b_1$
- 低頻度上位順: $a_2a_3a_4 \cdots a_nb_2b_3b_4 \cdots b_na_1b_1$

今回は低頻度上位順を用いたが，高頻度上位順や後出し上位順でも低頻度上位順とほぼ同じ順序付けとなった．

以上の場合において，タプル頻度表から頻出パタンを抽出する実験を行った結果を表 4 に示す．

この結果を見ると，アイテム変数の順序付けが ZBDD の節点数に大きな影響を及ぼすことがわかる．動的重み付け法の場合はアイテム変数の数に比例して ZBDD の節点数が増加しているのに対して，低頻度上位順では指数的に節点数が増加している．これは，今回の人工的なデータベースでは全てのアイテムの出現回数が同じであるので，出現回数で順序付けをしようとしても結局アイテムの出現順に並んでしまうため，低頻度上位順では効果的な順序付けができなかったためである．これに対して動的重み付け法では，1 つのアイテムの順序が決まったときに，その決まったアイテムの重みが関連の深いアイテムに分配されることになるので，ZBDD の節点数に関して非常に縮約効果のある順序付けができたのだと考えられる．このことから，この人工的な例題では，局所計算性が大きく影響したと考えられる．

このように，アイテムの出現頻度だけでは効果的な順序付けができない場合においても，動的重み付け法では効果的な順序付けができることが示された．

表 4: 人工的な例題における VSOP の処理時間と ZBDD の節点数

	the DWA Method		Less Freq. Upper Order.	
n	size	time(s)	size	time(s)
2	5	<0.1	5	<0.1
3	10	<0.1	12	<0.1
4	15	<0.1	25	<0.1
5	20	<0.1	50	<0.1
6	25	<0.1	99	<0.1
7	30	<0.1	196	<0.1
8	35	<0.1	389	<0.1
9	40	<0.1	774	<0.1
10	45	<0.1	1,543	<0.1
12	55	<0.1	6,153	<0.1
14	65	<0.1	24,587	<0.1
16	75	<0.1	98,317	<0.1
18	85	<0.1	393,231	0.456
20	95	<0.1	1,572,881	1.954
22	105	<0.1	6,291,475	8.482
24	115	<0.1	Memory overflow	

4.2.2 ベンチマーク例題における動的重み付け法の効果

次に，実際のベンチマーク例題における ZBDD の動的重み付け法による順序付けの効果に関して行った実験の結果を示す．実験は，動的重み付け法を用いてタプル頻度表を構築する VSOP スクリプトを生成し，そこから ZBDD-growth 法を用いて頻出パタンを抽出するという形で，生成される ZBDD の節点数を調べた．その結果を表 5 に示す．データベース名の隣の () 内の数字は，抽出するパタンの最低頻度を表したものである．つまり，この数字より頻度が高いパタンを抽出するということになる．

この実験結果より，今回の動的重み付け法はアイテム変数を低頻度上位順に並べたものと非常に近い効果を得られることがわかった．BMS-WebView-1 に関しては今回の手法の方がよい結果を得られている．また，高頻度上位順と比べると大きな縮約効果が得られていることがわかる．

4.3 動的重み付け法による ZBDD の縮約効果の検証

今回の実験で，ZBDD における動的重み付け法の順序付けによる ZBDD の縮約効果が確認できた．出現頻度だけでは効果的な順序付けができない場合にも，動的重み付け法は効果的な順序付けを行えることがわかった．しかし，実際のベンチマーク例題において実験を行った場合では，高頻度上位順と後出し上位順よりは効果的な順序付けができていることが確認できたが，低頻度上位順とは大きな差は見られなかった．これは，人工的な例題では局所計算性が非常に大きく影響したが，実際のベンチマーク例題ではそれほど大きく影響しなかったためと思われる．また，低頻度上位順では出現頻度の低いアイテムから上位に並べられるが，これは動的重み付け法で出現頻度の低いアイテムに，より大きな重みが伝えられる傾向があるということと類似している．このことから，実際のベンチマーク例題においては動的重み付け法と低頻度上位順では似たような順序付けになったと考えられる．

このように，局所計算性よりもアイテムの出現頻度の方が ZBDD のサイズに大きな影響を与える場合においては，順序付けにかかる計算時間が短い出現頻度に関する順序付け法の方が効果的なこともある．例えば，動的重み付け法を用いた変数の順序付けには，BMS-WebView-1 では 10.6 秒かかっているが，単純に頻度を計算するだけなら 0.8 秒ですむ．実用的な例題では，低頻度上位順で十分な縮約効果が得られるならば，低頻度上位順で順序付けを行っても，比較的有効であるといえるのかもしれない．これは議論すべき課題の一つであるといえる．

そこで，動的重み付け法が特に有効なデータベースの特徴について考察を行った．実際のベンチマーク例題に関して行った実験では，動的重み付け法は低頻度上位順に比べて順序付けに時間がかかるのに，生成した ZBDD のサイズにはほとんど差が生じない，という例が多く見られる．前に述べたとおり，ZBDD のサイズに影響を与える要素として，出力の制御力と局所計算性があるが，出力の制御力は出現頻度が低いアイテムほど大きくなる．したがって，低頻度上位順は，出力の制御力を素直に反映した順序付けであると言える．すなわち，動的重み付け法と低頻度上位順の相違点は，出力制御力だけでなく局所計算性も考慮しているかどうか，ということになる．

今回の実験では FIMI-2003 のベンチマークデー

表 5: トランザクションデータベースにおける ZBDD の動的重み付け法の効果

	the DWA Method		More Freq. Upper Order.		Less Freq. Upper Order.		Late Appear. Upper Order.	
	size	time(s)	size	time(s)	size	time(s)	size	time(s)
chess (2,000)	1,422	5.8s	3,856	2,036.6s	1,415	5.8s	2,778	64.8s
mushroom (1)	16,403	1.0s	448,734	1.9s	15,131	1.0s	40,557	1.1s
connect (60,000)	348	27.5s	1,659	5,402.3s	348	27.5s	374	62.8s
BMS-WebView-1 (30)	106,920	98.8s	389,181	778.2s	109,989	103.1s	152,431	153.4s
BMS-WebView-2 (100)	2,171	352.1s	3,201	460.0s	2,298	354.4s	Memory Overflow	
pumsb (40,000)	808	202.1s	Memory Overflow		813	202.7s	Memory Overflow	

表 6: ランダムに生成した高出現頻度データに対する実験結果

Method	average	max	min
the DWA Method	4,188	9,102	2,448
Less Freq. Upper Order.	10,086	15,339	3,827

表 7: 密なアイテムの固まりを mushroom に付加したデータに対する実験結果

Method	average	max	min
the DWA Method	22,852	26,971	19,121
Less Freq. Upper Order.	25,242	30,495	18,983

タ（表 2 参照）を用いたが，この表 2 を見てわかるように実際のベンチマーク例題は，疎（アイテムの平均出現頻度が小さい）であることが多い．そのような例題では，どの 2 つのアイテムの組を取ってみても，その 2 つのアイテムが同時に欠けているタプルが多数存在する可能性が高く，特定のアイテムの組だけに強い局所計算性が生じにくい構造となっている．したがって，動的重み付け法と低頻度上位順の間に大きな差が生まれなかったと考えられる．

一方，アイテムの平均出現頻度が高い例題では，局所計算性の影響が強く出る可能性がある．その一例として，ランダムに生成した高出現頻度データを用いて実験を行った．このデータは，アイテム 30 個を用意し，出現頻度 90%（27 個のアイテム）を 1 つのタプルとし，これをランダムに 30 タプル生成したものである．このようなデータを異なる乱数列により 10 通り生成し，それぞれに対して，動的重み付け法と低頻度上位順を適用した場合の ZBDD の節点数の平均値，最大値，最小値を比較したところ，上記の表 6 のような結果が得られた．この表を見ると，低頻度上位順よりも動的重み付け法の方が明らかに良い結果が得られている．このように平均出現頻度が高いデータでは，局所計算性の影響が大きくなりやすく，動的重み付け法の方が，単純な低頻度上位順よりも良い順序付けを出す場合がある．

それでは，疎なデータベース例題では動的重み付け法は全く不要かというと，必ずしもそうとは限らない．実際のデータベースでは，アイテムの出現に偏りがあり，全体としては疎であっても，一部に密なアイテムの固まりが存在する例がしばしば見られる．例えば，例題 “BMS-WebView-1” は，ネット販売サイトの 1 トランザクションごとのアクセスログのデータであり， 全体としては極めて疎であるが，特定の Web ページ群を表すアイテム群が固まって出現しているタプルが含まれている．このような密な固まりの部分だけでも局所計算性を考慮しないと，それによって ZBDD のサイズが増大してしまう可能性がある．

疎なデータベースの一部に密なアイテムの固まりが存在する例として，上記の方法でランダムに生成された高出現頻度データを，例題 “mushroom” に付け足して一つのデータベースとした例題を作成した．ただし，両者のアイテムは互いに重なりがないように区別している．このデータベースに対して動的重み付け法と低頻度上位順を適用し ZBDD を生成し，異なる乱数列 10 通りについて，節点数の平均値，最大値，最小値を比較した．その結果，元々の mushroom（表 5 参照）では低頻度上位順の方が良かったのに対し，密なアイテムの固まりを付加した場合には，表 7 の通り傾向が逆転し，動的重み付けの方が低頻度上位順よりも平均して良い結果が得られることが確認できた．このことから，全体として疎なデータベースの一部に密なアイテムの固まりが存在するようなデータベースの場合にも，動的重み

付け法は有効であると言える．

4.4 計算時間

最後に，全体的な計算時間について考察すると，実験では ZBDD-growth 法を用いてタプル頻度表から頻出パタンを抽出したが，その際にデータベースの種類やアイテム変数の順序付けによって，ZBDD の生成にかかる時間が著しく変化することがわかった．例えば，chess では動的重み付け法では 5.8 秒ですんだが，高頻度上位順では 2,036.6 秒もかかった．しかし，mushroom においてはどの順序付けでも大きな処理時間の差はなかった．これは，VSOP プログラムでは処理を高速に行うためにハッシュテーブルを用いており，そのハッシュテーブルの効果によって処理時間に差が出てしまったと考えられる．

5 変数順序付けプログラムの計算時間と高速化手法

本章では，前章で提案した動的重み付け法を高速化する手法を提示し，改良前の手法と比較した結果を示す．

5.1 変数順序付けプログラムの計算時間と高速化手法

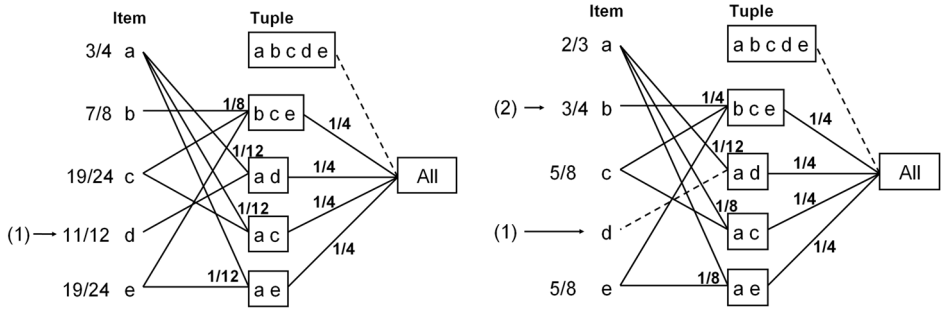
前章で説明したとおり，我々は当初，トランザクションデータに対する順序付け法の説明の (2) に示されているように，各々のタプルに含まれないアイテムに対して重みを分配してきた．しかしこの場合，アイテムの種類の数を n，データベースの総文字数を m，アイテムの平均出現確率を P とすると，計算時間は $O(nm(1-P)/P)$ となる．この場合，疎なデータベース，つまりアイテムの平均出現確率 P が低いデータベースに対しては非常に膨大な時間を要することになる．ここで，各実験で用いた FIMI-2003 のベンチマークデータの表 2 を見てみると，アイテムの平均出現頻度は一番高い chess の場合でも 50%未満であり，BMS-WebView-1 などでは 1%を割っている．このように，実際のデータベースは疎であるものが多く，従来の動的重み付け法では膨大な処理時間が必要となってしまう．よい変数順序付けを行うことで生成される ZBDD はコンパクトになり，生成するのにかかる時間が短くなると考えられるが，その変数順序付けを行う際に膨大な時間がかかってしまってはよい変数順序付けを行う意味が半減してしまう．この問題を解決するために，我々は前章で示した動的重み付け法の改良を行った．

改良した動的重み付け法では，まず全てのアイテムに *All* に与える重みと同じ重み 1 を与える．次に，各々のタプルに含まれるアイテムに関して，タプルに与えられた重みをタプルに含まれないアイテムの数で割った重みをそれぞれのアイテムから引くという処理を行う．この操作の様子を図 20 に示す．図 20 の (a) は 1 回目の重み付け処理が終わったところで，アイテム d が変数順序の 1 番に決まったところである．(b) は 1 回目の操作で順序が決まった変数に関する線を削除してもう一度重み付けを行い，一番大きな重みを持つアイテム b が変数順序の 2 番に決まったところである．

これまでの手法では，タプルに含まれないアイテムに対して重みを加えるという操作で変数の順序付けを行ってきたが，改良した手法では，旧手法でタプルに含まれないアイテムに加算されていた分の重みをタプルに含まれるアイテムから引くという操作に変更する．この操作により，改良手法では旧手法と同じ分だけタプルに含まれないアイテムの重みが相対的に増加するので，同じ順序付けが得られることになる．改良後の計算時間は，アイテムの種類の数を n，データベースの総文字数を m とすると，$O(nm)$ となり，入力されるデータベースの総文字数に依存した計算時間となる．タプルに含まれるアイテムに重みを伝えるという処理に変更することによって，データベースは表 2 に示されるように疎である場合が多いので，重みを伝えるアイテムの数が格段に減ることが予想される．例えば，アイテムの平均出現確率 P が 1%である場合を考えると，旧手法の $O(nm(1-P)/P)$ と改良手法の $O(nm)$ より，約 100 倍の高速化が実現される．

5.2 旧手法と改良手法の比較実験

前章で示した手法と改良した手法とを比較する実験を行った結果を表 8 に示す．以前の手法と改良を加えた手法を比較すると，格段に処理時間が削減されていることがわかる．この処理時間の削減の度合いに違いが見られるのは，データベースがより疎なものほど処理時間の削減が顕著であるからである．この結果，従来手法では変数の順序付けを行うこと



(a) First (b) Second

図 20: トランザクションデータベースにおける動的重み付け法

表 8: 動的重み付け法の順序付けの計算時間

Database	旧手法 (sec)	改良手法 (sec)
chess	2.8	0.7
mushroom	17.9	1.5
connect	219.6	15.1
BMS-WebView-1	1,255.0	10.6
BMS-WebView-2	(>1h)	118.5
pumsb	(>1h)	306.8
pumsb_star	(>1h)	219.8
retail	(>1h)	825.5

ができず，頻出パタン集合を表す ZBDD を生成することができなかったデータベースに関しても，現実的な時間で変数の順序付けを行うことができるようになり，扱えるデータベースの範囲を拡大することができるようになった．

6 サンプリングを用いた変数順序付け法の高速化

本章では，トランザクションデータベースをサンプリングして得られたタプルを用いて変数順序付けを行う手法を提示し実験結果を示す．

6.1 サンプリングを用いた変数順序付け法の高速化

前述のように，動的重み付け法ではアイテム数を n，データベースの総文字数を m とすると，$O(nm)$ の計算時間がかかる．動的重み付け法が改良されて以前よりも速く変数の順序付けを行うことができるようになったが，それでもまだデータベースの規模が大きくなると現実的な時間内で変数順序を求めることは難しくなる．そこで我々は，トランザクションデータベースをサンプリングし，そのサンプリングされたタプルのリストを用いて変数の順序付けを行うということを考える．この手法の目的は，元のデータベースの一部，または大部分を使用しないことになるのでデータベース全体を用いた場合よりも変数順序の質は多少劣るかもしれないが，現実的な計算時間で変数の順序付けを行えるようにしようということである．

サンプリングにより得られるデータには，元のデータベースに現れる全てのアイテムが含まれていない場合がある．このため，一部のアイテムの変数順序は未決定となることがある．順序付けが未決定のアイテムが ZBDD の生成中に参照された場合には，そのアイテムは順序付けの最上位に配置される．サンプリングレートを 0%に設定した場合，全てのアイテムは元のデータベース中で最初に出現する位置が遅いものほど順序付けの上位に配置されることになる．この順序付けを我々は「後出し上位順」と呼び，以前にこの順序付けに関しても実験を行った．その結果，動的重み付け順ほどよくはないが，ある程度よい順序付けを行えることが確認された．今回の実験は，この後出し上位順と動的重み付け法を組み合わせた手法を提案しているとも言える．統計学的に考えれば，元のデータベースとの誤差をある程度の範囲に抑えるレートを求めることは可能であるが，ここで重要なのは，サンプリングされたデータベースを用いて生成された変数順序が，元のデータベース全体を用いて生成された変数順序と比較して，どのくらい似ているかということではなく，最終的に変数順序を生成するのにどの程度の処理時間を要するのか，また，そうして生成された変数順序がデー

タベース全体を用いて生成されたものと比較してどの程度よい順序付けか，ということである．データベース全体を使用した場合の ZBDD の節点数と大差がなく，変数の順序付けにかかる時間を大幅に減らすことのできるサンプリングレートを見つけることが重要となる．

6.2 サンプリングを用いた変数順序付け法の高速化の実験結果

サンプリングを用いて行った実験の結果を示す．使用した PC は，Pentium-4 3.0GHz，SuSE Linux 9.3，主記憶 1GByte で最大節点数は 2,000 万個とした．

今回の実験では，トランザクションデータベース全体を用いて重み付けを行った場合と，データベースからランダムに総タプル数の 0%，5%または 10%のタプルを抽出し，それを用いて重み付けを行った場合とを比較した結果を示す．0%というのは，重み付け法を用いた変数順序付けを行わないということと同義である．実験では，データベースに対してある割合でサンプリングを行い，変数の順序付けをし，ZBDD を生成するということを 20 回行い，その平均節点数と平均計算時間を計算し比較した．また，変数の順序付け法として動的重み付け法を用い，順序付けされていないアイテムが ZBDD の生成中に参照された場合には，そのアイテムは順序付けの最上位に配置される．

表 9 に実験結果を示す．size は ZBDD の平均節点数，time は動的重み付け法の平均計算時間である．データベース名の隣の () 内の数字は，抽出するパタンの最低頻度を表したものである．つまり，この数字より頻度が高いパタンを抽出するということになる．この表を見ると，サンプリングレートが 5%や 10%とかなり小さい場合でも，データベース全体を用いた場合と比べて，ZBDD のサイズにあまり差がないことがわかる．一方で計算時間は，サンプリングレートに比例して減少していることがわかる．これにより，用いるのがデータベースの一部だけでも，比較的よい順序付けをデータベース全体を用いる場合と比べて高速に行えることがわかった．また，ZBDD の生成にかかる時間はサンプリングレートを変えてもほとんど変化しなかった．これらのことから，実際にデータベース解析を行う場合には，その状況に応じてサンプリングレートを調節し，ZBDD を生成することが有効であると考えられる．

7 結論

7.1 おわりに

本論文では，データベース解析において生成される ZBDD の簡単化に関して種々の実験を行った．その結果，トランザクションデータベースにおける動的重み付け法による ZBDD の縮約効果が確認できた．人工的な例題においては局所計算性を反映することで，効果的な順序付けを行うことができた．また，実際のデータベースにおいても効果的な順序付けを行えることが確認できたが，出現頻度を基に行った順序付けと大きな差が生まれない場合もあるということがわかった．そこで，動的重み付け法が効果を発揮するデータベースの特徴について考察し，アイテムの平均出現頻度との関係を示した．

また，動的重み付け法の計算時間 $O(nm)$ を改善するための手法として，トランザクションデータベースをサンプリングし，抽出されたタプルに対して動的重み付け法を適用し変数の順序付けを行う手法を提案し，実験を行った．その結果，サンプリングレートがかなり低い場合においても，データベース全体を用いた場合の変数順序付けの品質と比較して，あまり差が生まれないことがわかった．その一方で変数順序付けに要する計算時間はサンプリングレートに比例して減少することがわかった．これらのことから，実際にデータベース解析を行う場合には，その状況に応じてサンプリングレートを調節し，ZBDD を生成することが有効であると考えられる．

7.2 今後の課題

今後，研究を行っていくにあたって考えられる課題として以下のようなものがある．

(1) 動的重み付け法の計算時間 $O(nm)$ を上回る手法を開発する．

(2) 実際に ZBDD を生成する際に使用したデータベースの数が少ないので，より多くのデータベースに対して実験を行い，考察する．

(3) データベースの性質を数学的に解析することで，よりよい変数順序付けを理論的に求められるようにする．

(1) に示すように，本論文で示した動的重み付け法はアイテムの種類の数を n，データベースの総文字

表 9: サンプリングを用いた ZBDD の動的重み付け法の効果

	100%		10%		5%		0%	
Database (min. support)	size	time(s)	size	time(s)	size	time(s)	size	time(s)
chess (1)	456,536	0.48s	456,801	0.08s	475,656	0.06s	1,029,615	0.0s
mushroom (1)	16,403	1.38s	17,092	0.15s	17,296	0.09s	40,557	0.0s
connect (15,000)	25,327	18.91s	25,296	1.96s	25,267	1.01s	(*)	0.0s
BMS-WebView-1 (30)	106,920	12.71s	108,189	0.99s	108,444	0.30s	152,431	0.0s

*: memory overflow

数を m とすると計算時間は $O(nm)$ となる．今回実験で使用したデータベースの中で変数の順序付けに最も時間がかかった retail の場合，順序付けに 825.5 秒かかった．retail のサイズは約 4.0MByte で必ずしも大規模でないことを考えると，動的重み付け法はよい変数順序を得られるがまだ実用的とはいいがたい．

この点に関して本論文では，サンプリングを行ったデータベースに対して動的重み付け法を適用することで，計算時間を大幅に短縮できることを示した．しかし，サンプリングレートに関しては，どの程度にすれば出力される変数順序の質と順序付けに要する計算時間を両立できるのかはわかっていない．サンプリングレートを低くして変数の順序付けをする実験を何度か行ったとき，その都度サンプリングされるタプルが変化するので，場合によっては得られる結果にランダム性が大きく影響する可能性がある．

このような理由から，データベースを定数回読み込むことで変数の順序付けを行うことができる手法を考案する必要がある．そのためには，課題 (2),(3) に示すように，より多くのデータベースに関して実験を行い，結果を精査することでデータベースの構造などの性質がどのように変数の順序付けや ZBDD の生成に影響を与えているのかを調べなければならない．例えば，T10I4D100K というデータベースは全くランダムに生成されたデータであるので，それぞれのアイテムに関して性質的な違いはほとんど見られない．よって，変数の順序付けがどのようになろうと結果に大きな差は生まれない．またその他にも，T10I4D100K とは逆にそれぞれのアイテムの出現確率に大きな差があるものなどいろいろなケースがあり得る．

このようなデータベースの特徴を調べることで，ある特定の性質を持つデータベースに対して有効な変数の順序付けを定数回のデータベース読み込みで実現できる手法を開発したいと考えている．そして，その成果を生かしてトランザクションデータベースを具体的に解析する研究を行い，本手法のデータマイニングの分野における可能性を追求していきたい．

謝辞

本研究を行うにあたり，日頃より多大な御指導をいただき研究活動を支えて下さいました Thomas Zeugmann 教授，湊真一准教授に深く感謝いたします．また，本研究を行う上で御助言，御協力をいただいた全ての方々にお礼を申し上げたいと思います．

参考文献

[1] R. Agrawal, H. Mannila, R. Strikant, H. Toivonen and A.I. Verkamo, Fast Discovery of Association Rules, In Advances in Knowledge Discovery and Data Mining, MIT Press, 307-328, 1996.

[2] S.B. Akers, Binary decision Diagrams, IEEE Trans. Comput., C-27, 6 (1978), 509-516.

[3] R.K. Brayton, R. Rudell, A. Sangiovnni-Vincentelli, and A.R. Wang, “MIS: A multiple-level logic optimization system,” IEEE Trans. on CAD, vol. CAD-6, pp. 1062-1081, Nov. 1987.

[4] R.E. Bryant, Graph-based algorithms for Boolean function manipulation, IEEE Trans. Comput., C-35, 8 (1986), 677-691.

[5] 藤田昌宏, 佐藤政生 編, 特集「BDD（二分決定グラフ）」, 情報処理学会誌, 34, 5 (1993), 584-630.

[6] M. Fujita, H. Fujisawa and N. Kawato. Evaluation and improvement of Boolean comparison

method based on binary decision diagrams. In Proc. of IEEE/ACM International Conference on Computer-Aided Design (ICCAD-88), pp. 2-5, Novenber 1988.

[7] B. Goethals, "Survey on Frequent Pattern Mining" (Pdf) Manuscript, 2003; http://www.adrem.ua.ac.be/~goethals/publications/aurvey.ps

[8] B. Goethals, M. javeed Zaki (Eds.), Frequent Itemset Mining Dataset Repository, 2003. Frequent Itemset Mining Implementations (FIMI'03), 2003. http://fimi.cs.helsinki.fi/data/

[9] J. Han, j. Pei, Y. Yin, R. Mao, Mining Frequent Patterns without Candidate Generation: A Frequent-Pattern Tree Approach, Data Mining and Knowkedge Discovery, 8(1), 53-87, 2004.

[10] Y. Hayashi and J. Matsuki, "Determination of Optimal System Configuration in Japanease Secondary Power Systems", IEEE Trans. on Power Systems, VOL. 18, NO. 1, pp. 394-399, Feb. 2003.

[11] C.Y. Lee, Representation of switching circuits by binary-decision programs, Bell Sys. Tech. Jour., 38 (1959), 985-999.

[12] S. Malik, A.R. Wang, R.K. Brayton, and A.S. Vincentelli, Logic verification using binary decision diagrams in a logic synthesis environment, In Proc. ACM/IEEE Internationl Conf. on Computer-Aided Design(ICCAD-88), (1988), 6-9.

[13] 正木寛人, 斉藤逸郎, 石塚満, 奥乃博 : 「二分決定グラフを用いた三面図の効率的理解」SIG-AI-98, 人工知能研究会, 情報処理学会. Jan. 1995.

[14] 湊 真一, 石浦 菜岐佐, 矢島 脩三, 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, 情報処理学会論文誌, Vol.32, No.1, pp.77-85, January 1991.

[15] S. Minato, Zero-suppressed BDDs for set manipulation in combinatorial problems, In Proc. 30th ACM/IEEE Design Automation Conf. (DAC-93), (1993), 272-277.

[16] S. Minato: "BEM=II: An Arithmetic Boolean Expression Manipulator Using BDDs", IEICE Trans Fundamentals, Vol. E76-A, No. 10, pp.1721-1729, Oct.1993

[17] S. Minato: "Calculation of Unate Cube Set Algebra Using Zero-Suppressed BDDs", In Proc. of 31st ACM/IEEE Design Automation Conference (DAC'94), pp. 420-424, Jun. 1994.

[18] S. Minato, Zero-suppressed BDDs and Their Applications, International Journal on Software Tools for Technology Transfar (STTT), Springer, Vol. 3, No. 2, pp. 156-170, May 2001.

[19] 湊真一, VSOP: ゼロサプレス型 BDD に基づく「重み付き積和集合」 計算プログラム, IEICE Technical Report COMP2005-13(2005-5)

[20] 湊真一, 有村博紀, ゼロサプレス型二分決定グラフを用いたトランザクションデータベースの効率的解析手法, 電子情報通信学会論文誌, Vol.J89-D, No2, pp. 172-182, 2006.

[21] S. Minato, H. Arimura. Frequent Pattern Mining and Knowledge Indexing Based on Zero-suppressed BDDs. In Proc. The 5th International Workshop on Knowledge Discovery in Inductive Databases (KDID-2006), pp. 83-94, Sep. 2006.

[22] 奥乃 博, 湊 真一: "二分決定グラフによる制約充足問題の解法", 情報処理学会論文誌, Vol. 36, No. 8, pp. 1789-1799, Aug. 1995.

[23] 奥乃博, 下國治, 田中英彦 : 「二分決定グラフによる多重文脈型真偽維持システム BMTMS」人工知能学会誌, Vol.11, No.2 (Mar. 1996).

[24] S. Tani, K. Hamaguchi, and S. Yajima, The complexity of the optimal variable ordering problems of shared binary decision diagrams, In 4th International Symposium on Algorithms and Computation, LNCS-762, Springer(1993), 389-398.

[25] M.J. Zaki, Scalable Algorithms for Association Mining, IEEE Trans. Knowl. Data Eng. 12(2), 372-390, 2000.