3I1-R-9-8

ZDDを用いた行列圧縮における演算高速化

Fast Construction Method of ZDD Representing a Binary Matrix

西野 正彬[*1] 安田 宜仁[*1] 湊 真一[*2] 片岡 良治[*1]
Masaaki Nishino Norihito Yasuda Shin-ichi Minato Ryoji Kataoka

[*1]日本電信電話株式会社 NTT サイバーソリューション研究所
NTT Cyber Solutions Laboratories, NTT Corporation

[*2]北海道大学 大学院情報科学研究科
Graduate School of Information Science and Technology, Hokkaido University

Multiplication between a binary matrix and a real value vector can be seen in various situations. In the previous work, we showed the number of computation needed for multiplications can be reduced by using ZDD (Zero-suppressed Binary Decision Diagrams). The proposed calculation method, however, needs to translate a binary matrix to corresponding ZDD before operation, and the translation needs non-negligible computation time. In this paper, we will show a translation method that can directly translate a binary matrix into a reduced ZDD. Our method is first and can work with less memory.

1. はじめに

行列とベクトルとの積の計算は，様々な場面で広く用いられる基本的な計算のひとつである．例えば集合拡張 *(Set Expansion)* とよばれる情報検索手法の一種では，種集合を入力として受け取り，アイテム集合から種集合に似たアイテムを出力する．この処理はアイテム集合を表す行列と種集合とを表すベクトルとの積によって実現される．筆者らは以前に集合拡張アルゴリズムの高速化を目的として，ゼロサプレス型二分決定グラフ *(ZDD)* を用いて二値行列を表現することによって，二値行列とベクトルとの積の計算を高速化する手法を示した [西野 12]．ZDD を用いて二値行列をコンパクトに表現し，ZDD の構造を利用した演算として行列とベクトルの積を計算することによって，計算回数を削減することができる．

ZDD を用いた行列とベクトルとの乗算は，集合拡張に限らず様々な場面で適用可能である．しかし，ZDD を使うには処理に先立ち二値行列を ZDD に変換する必要がある．一旦変換すれば繰り返し計算に利用できるが，変換にはかなりの計算時間を要し，またメモリ量も膨大になるという問題があった．

本稿では ZDD 構築に時間がかかる問題に対処するため，二値行列を高速に ZDD に変換する手法を示す．通常の ZDD 処理系を用いた二値行列から ZDD への変換においては，小さな組合せを表す ZDD 間で集合演算を繰り返すことで，ZDD を段階的に構築する必要があった．本稿で提案する ZDD への変換手法では，行列を，列ごとに一度ずつ処理することによって既約な ZDD を集合演算用いずに構築することができるため，高速に動作し，かつメモリ量も削減することができる．

本稿の構成を示す．2 章と 3 章で，既存手法である ZDD と，ZDD を用いた行列とベクトルの積の計算方法についてそれぞれ述べる．4 章で提案する二値行列から ZDD への変換手法について述べたのち，5 章で検証結果，6 章でまとめを述べる．

2. ZDD

ZDD は，Minato[Minato 93] によって提案された，組み合わせ集合の表現に特化した二分決定グラフ (BDD: Binary Decision Diagrams) [Bryant 86] である．ZDD は指向性をもったループのないグラフの構造をしており，グラフの終端以外の各節点はすべて 0-枝と 1-枝のふたつのリンクをもつ．グラフ末端の節点は終端節点とよばれ，いずれの ZDD も 0-終端節点と 1-終端節点の 2 つの終端節点を持つ．終端以外の節点は，すべて組み合わせ集合に含まれるあるシンボルに対応づけられている．シンボル a, b, c からなる組み合わせ集合 $\{ab, bc\}$ を表した ZDD の例を図 1 に示す．ある組み合わせ集合を表す ZDD の構造は，変数の順序が一定ならば同一となる．

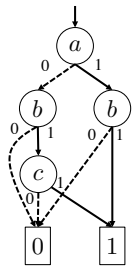
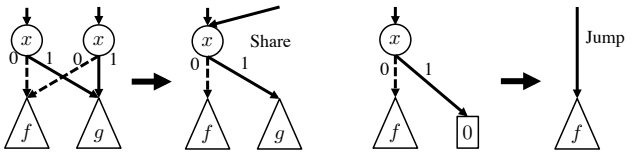


図 1: 組み合わせ集合 $\{ab, bc\}$ を表す ZDD の例

順序づけされた ZDD に対して簡約化規則を可能な限り適用することによって，それ以上簡約化できない既約な ZDD を得ることができる．ZDD の簡約化規則には，冗長な節点の削除と等価な節点の共有の 2 種類がある．2 種類の規則による簡約化の様子を図 2 に示す．ある ZDD に対してより多くの簡約化規則を適用ことが可能ならば，最終的に得られる簡約化された ZDD の節点数を少なくすることができ，ZDD を保持するために必要なメモリ量を削減できる．

ZDD は各節点ごとに対応するシンボル，1-枝が指す節点のアドレス，0-枝が指す接点のアドレスの 3 つのフィールドを用意することで計算機上で表現することができる．ある ZDD f の総節点数を $B(f)$ とすると，ZDD は上記 3 フィールドを格納する要素数 $B(f)$ の配列で表現できる．配列中の k 番目の節点に対応するシンボルを v_k, 1-枝，0-枝が指す節点の配列中での添字を，それぞれ h_k, l_k とする．

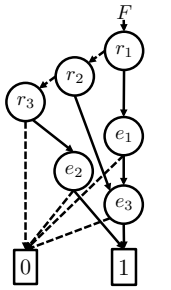
連絡先: nishino.masaaki@lab.ntt.co.jp



(a) 等価なノードの共有　(b) 冗長なノードの削除

図 2: ZDD の簡約化規則

$$\begin{pmatrix} 1 & 0 & 1 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix}$$



(a) 2値行列の例　(b) 対応するZDD

図 3: 二値行列と対応する ZDD の例

3. ZDD を用いた疎行列とベクトルの乗算

ZDD によって行列との積を計算する手法について述べる．まず ZDD によって二値行列を表現する手法を示したのちに，ZDD の構造を用いて二値行列と実数ベクトルとの積を計算する手法について説明する．

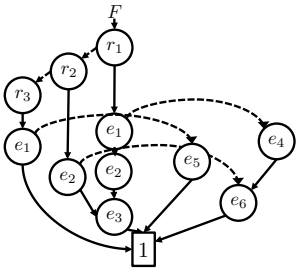
ZDD によって二値行列を表現する手法について述べる．N 行 M 列の二値行列 $\mathbf{X}$ を組み合わせ集合として表現する事を考える．組み合わせ集合で用いられるシンボルの集合 S を，$S = \{r_1, \ldots, r_N, e_1, \ldots, e_M\}$ とする．ここで，$r_1, \ldots, r_N$ は，それぞれ行列の各行に対応するするシンボルであり，$e_1, \ldots, e_M$ は行ベクトル（M 次元ベクトル）の各成分にそれぞれ対応するシンボルとする．これらのシンボルを用いて，$\mathbf{X}$ に対応する組み合わせ集合 $Z_{\mathbf{X}}$ を，

$$Z_{\mathbf{X}} = \bigcup_{i=1}^{N} \{r_i e_{a_i(1)} \cdots e_{a_i(b_i)}\} \quad (1)$$

として表現する．ここで $a_i(j)$ は，i 行目に対応する行ベクトル $\mathbf{x}_i$ の要素 $x_{i1}, \ldots, x_{iM}$ のうち，値が 1 となる j 番目要素の添字を表す．また b_i は $x_{i1}, \ldots, x_{iM}$ のうち値が 1 となる要素の総数である．$1 \le a_i(1) < \cdots < a_i(b_i) \le M$ を満たす．例えば図 3(a) の行列には，組み合わせ集合 $\{r_1e_1e_3, r_2e_3, r_3e_2\}$ が対応する．これを ZDD として表現したものが図 3(b) となる．ここで，シンボル間には $r_1 < r_2 < \cdots < r_N < e_1 < e_1 < e_2 < \cdots < e_M$ という順序を与えている．順序は変更してもよいが，任意の $1 \le i \le N, 1 \le j \le M$ について $r_i < e_j$ となるようにする．

このようにして構成された ZDD にはいくつかの特徴がある．まず，このように構成された ZDD の節点数は，$N + \sum_{i=1}^{N} b_i$ 以下であり，かつシンボル $r_1, \ldots, r_N$ に対応する節点は，常に高々1 つしか出現しないという特徴がある．次に，節点の共有による簡約化については，共有が起こりうるのは $e_1, \ldots, e_N$ に対応する節点のみとなり，かつ共有が起きる条件が，各項に含まれるシンボルのうち，順序的に後にくるものがすべて共通な場合にのみ，ZDD において構造の共有が行われるというものになる．図 3 では，行列の 1 行目に相当する項が $r_1e_1e_3$，3 行目に相当する項が r_3e_3 であり，e_3 に相当する節点で共有による圧縮が発生している．

$$\left(\begin{array}{ccc|ccc} 1 & 1 & 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 & 0 \end{array}\right)$$



(a) 2値行列の例　(b) 対応するZDD

図 4: 行分割の例

構築された ZDD の構造を利用した動的計画法によって二値行列と実数ベクトルとの積を計算することができる．手法の説明は [西野 12] にある．ZDD で行列を表現することによって必要な計算回数が ZDD の節点数回となるため，簡約化規則を多数適用できてコンパクトな ZDD として二値行列を表現できるならば，計算回数を削減することができる．

3.1 行分割による節点数削減

より ZDD の簡約化が進むための工夫として，筆者らは [西野 12] で行分割とよぶ手法を提案した．行分割とはひとつの二値行列を L 列ごとに部分行列に分割して ZDD として表現することであり，節点の共有による簡約化が起こりやすくなるという効果がある．図 4 は 3 つの 6 次元ベクトルからなる行列について，$L = 3$ として分割を行ったときに生成される ZDD を示している*1．分割を行わなかった場合にはラベル e_6 をもつ節点でしか共有は行われなかったが，分割することによって e_3 ラベルをもつ節点でも共有が起きているため，よりコンパクトに行列を表現できている．

一見，L を小さな値に設定するほど節点の共有による圧縮が起こるように見えるが，分割にはコストが伴う．図 4(b) の 1,2,3 列を 1 つの行列として考えると，1 行目と 2 行目では e_2 と e_3 が等しいため，共有が起こるべきである．しかし，4,5,6 列の行列へのリンクを 0-枝として保持しなければならないため，e_2 での共有が行われなくなっている．このように，0-枝が 0-終端節点の節点に繋がっている，ラベルが $e_1, \ldots, e_M$ であるような節点を以下では先頭節点とよぶ．先頭節点は 0-枝が終端節点以外の節点につながっているため共有による圧縮が起こりにくい．分割によって作られる各部分行列ごとに先頭節点が必要となるため，L が小さな値だと先頭節点が多く作られ，共有が起こりにくくなるという性質がある．そのため，L はデータに合わせた適切な値に設定する必要がある．

4. 二値行列を表す ZDD の構築

一旦 ZDD を構築できれば，それを用いた行列とベクトルとの積の計算は前述のように ZDD の節点数回の計算で効率的に実行することができる．しかし，ZDD 構築後の処理と比べて，二値行列を ZDD に変換する処理は，対象とする二値行列

*1 図が複雑になるのを防ぐため，図 4(b) では 0 終端節点に達するリンクは省略している．

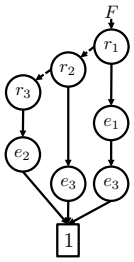


図 5: 節点の共有を行っていない ZDD

が大きな場合にはそれなりの処理時間を要する．ZDD の構築は，一般には ZDD の小さなパーツに集合演算を繰り返し適用することによって行われる．行の分割を行いつつ，二値行列を ZDD に変換する場合，各行について項を生成し，その項の和集合を繰り返し計算する必要がある．もし行列の行数が N であったならば，少なくとも N 回の和集合の演算を実行しなければならない．

ある ZDD f と g について，それぞれの節点数を $B(f)$, $B(g)$ としたとき，集合演算を行うために必要な計算回数は $O(B(f)B(g))$ となる．一般の ZDD 処理系ではキャッシュを効果的に利用することによって，集合演算に必要な計算を大幅に削減することができる．しかし，行列とベクトルの積の計算は，様々な場面で利用される汎用的な計算であるため，高速化の効果は大きい．また，最終的に生成される ZDD のサイズに比べて演算途中に多くの記憶領域を必要とするため，大きなサイズの行列は ZDD に変換できない可能性がある．そこで，以下では集合演算を用いずに高速に ZDD を構築する方法を述べる．

4.1 二値行列から ZDD への変換手続き

本稿で提案する，二値行列から対応する ZDD への変換手続きを説明する．処理の説明に先立ち，二値行列がどのような ZDD で表現されるか，すなわち，ZDD の 2 種類の簡約化規則がどのような場面で適用されるかを説明する．図 3 の行列に対応する組合せ集合は $\{r_1e_1e_3, r_2e_3, r_3e_2\}$ である．この組合せ集合から節点の共有を行わずに ZDD を構築すると，図 5 に示す既約でない ZDD が得られる．図より，節点の共有を行わない ZDD は，各行を表す節点 (r_*) の 1-枝からスタートして，その行の非ゼロ成分に対応する節点 (e_*) を 1-枝で順に辿ったものとして表現できることが分かる．また，この ZDD の節点の個数は行列中の非ゼロ成分の数と行列の行数の和となっており，ゼロ成分に関する節点は存在しないことが分かる．つまり，値がゼロである成分について，節点削除の簡約化規則が適用されていることが分かる．

次にこの ZDD を既約な ZDD とするためにどのような節点の共有が行えるかを考える．図中の ZDD では，1, 2 行目 3 列に対応する節点の 0-枝，1-枝の指す先が一致するため，これらの節点の共有を行う．いま，2 つの行ベクトルにおいて，その i 番目以降の成分が等しいとする．すると，e_i に対応する節点の 1-枝は共通の i の次の非ゼロ要素に対応する節点，0-枝はゼロ終端節点を指すので，e_i で節点の共有が発生する．以上より，ZDD の 2 種類の簡約規則 (節点の削除，共有) は，それぞれ (1) ゼロ成分に対応する節点の削除，(2) 以降の行の成分が等しい節点の共有，という 2 種類の簡約化に対応することが分かる．

行列が与えられた時にその各列ベクトルを順に処理することで，既約な ZDD を作成する手順をアルゴリズム 1 に示す．ここで top は N 次元の配列である．top には，列を順次処理していく上で，ある列の処理が終わった時点での各行に対応する ZDD の節点のうち，最も小さなシンボルに対応する節点のアドレスを格納する．格納されたアドレスは，別の ZDD の節点を追加する際に 1-枝のアドレスを知るために用いられる．G は行番号の分割であり，分割の 1 つのコンポーネントは，節点が共有されるべき行番号から構成される．G はすべての行番号からなるコンポーネントを 1 つのみもった集合として初期化される．図中の *update* は，G を列ベクトル $\boldsymbol{c}$ の値をもとに，最新の分割に更新する処理である．G 中のあるコンポーネント g に含まれる行番号間で，$\boldsymbol{c}_i$ の行番号に対応する値が異なるとき，g を分割する．例えば図 3 の行列で 3 列目のみを見た時のコンポーネントは $\{\{1,2\},3\}$ であるが，2 列目を読み込んだ後では $x_{21} \neq x_{22}$ であるから，1, 2 行目が分割され $\{1,2,3\}$ となる．各列について G を更新したのちに，G のコンポーネントのうち，その列で 1 となっているものを選択する．そのような各コンポーネントについて新しい節点をひとつ ZDD に追加する．こうして追加された節点では，共有による節点数の削減が起きている．なお，新たに追加する ZDD の節点は，ラベルが e_i, 0-枝が 0-終端節点，1-枝が top[j] (j はコンポーネントに含まれるいずれかの行番号) に格納された節点を指すとして追加する．

Algorithm 1 二値行列を表す ZDD の生成手順

Input: 行列 $\mathbf{X}$
Output: 行列を変換した部分 ZDD

```
1: G ← {{1, 2, ..., N}} // 初期化
2: top[i] ← NULL for i ∈ {1, ..., N} // 初期化
3: for i ∈ {1, ..., M} do
4:    G ← update(G, c_i)
5:    for g ∈ G do
6:       if g に含まれる行について，現在の列の成分が 1 then
7:          ZDD に g に対応する節点を追加.
8:          for j ∈ g do
9:             top[j] ← 追加した節点のアドレス
10: for i ∈ {1, ..., N} do
11:    節点 r_i を ZDD に追加.
12: return 構築された ZDD
```

4.2 行分割を導入した際の変換手続き

次に行分割を導入した場合の ZDD 構築手順について説明する．行分割時の ZDD の構築は，基本的には分割によって生成された各部分行列について前述の手続きを実行することによって実現される．ただし，行分割を行わない場合と違い，各部分行列での先頭節点の扱いに注意する必要がある．先頭節点は各部分行列中で，ある行に対応する節点のうち先頭にくるものである．先頭節点の 0-枝は別の部分行列の先頭節点を指す必要がある．先頭節点以外の節点では部分行列中で等しい行同士で節点の共有が発生するが，先頭節点は 0-枝を持たなければいけないため，共有が発生しない．

行分割を導入した際の ZDD への変換手続きをアルゴリズム 2 に示す．なお，図中で globalTop はサイズ N の配列である．globalTop には各行に関連する ZDD の節点のうち，処理した部分行列すべての中で最も小さなシンボルに対応する節点のアドレスを保持する．globalTop は先頭節点の 0-枝が指す先を記憶するために用いられる．$\mathbf{Y}_{ij}$ は分割された部分行列 $\mathbf{Y}_i$ の j 列目の列ベクトルを表す．基本的な処理は行分割を行わ

Algorithm 2 行分割を行う場合の二値行列を表す ZDD の生成手順

Input: 行列 $\mathbf{X}$, 分割単位 L
Output: 行列を変換した部分 ZDD

1: $\mathbf{X}$ を部分行列 $\mathbf{Y}_1, \mathbf{Y}_2, \ldots, \mathbf{Y}_{\lceil M/L \rceil}$ に分割
2: globalTop[i] ← NULL **for** $i \in \{1, \ldots, N\}$ // 初期化
3: **for** $i \in \{\lceil M/L \rceil, \ldots, 1\}$ **do**
4: $G \leftarrow \{\{1, \ldots, N\}\}$ // 初期化
5: top[i] ← NULL **for** $i \in \{1, \ldots, N\}$ // 初期化
6: **for** $j \in \{L, \ldots, 1\}$ **do**
7: $G \leftarrow update(G, \mathbf{Y}_{ij})$
8: **for** $g \in G$ **do**
9: **if** g に含まれる行番号の現在の列に対応する成分が 1 **then**
10: ZDD に g に対応する節点を追加.
11: **for** $k \in g$ **do**
12: top[k] ← 追加した節点のアドレス
13: **for** $j \in \{1, \ldots, N\}$ **do**
14: $p \leftarrow$ top[j] が指す ZDD の節点
15: j 行目に対応する先頭節点を p をもとに作成，ZDD に追加
16: globalTop[j] ← top[j]
17: **for** $j \in \{1, \ldots, N\}$ **do**
18: 節点 r_i を ZDD に追加
19: **return** 構築された ZDD

ない場合と同様の手続きを，各分割された部分行列について適用することである．(ステップ 3-12)．それに加え，各部分行列で先頭節点を生成する処理が追加される．まず各行について top[i] で部分行列でいちばん上位に来る節点 p を取得し (ステップ 14)，p の値をもとに先頭節点を作成し，ZDD に新しい節点として追加する．作成した節点のシンボルは v_p，0-枝は globalTop[i] の節点，1-枝は p の 1-枝と同じ節点を指す．

提案手法では，対象の二値行列を各列ごとに処理するだけの計算回数と，各部分行列における先頭節点の追加の処理の計算回数とで，二値行列に対応する ZDD を構築することが可能である．また，構築される ZDD と入力の二値行列のほかにメモリに保持する必要があるデータは，top，globalTop，G の 3 つのみであるため，メモリを消費せずに ZDD を構築することができる．

5. 検証

いくつかのサイズの二値行列から ZDD を構築し，計算に要する時間とメモリ量を調べた．検証には，それぞれサイズが $1,000 \times 1,000$, $5,000 \times 5,000$, $10,000 \times 10,000$ の，非ゼロ成分が 10%の確率で出現するとしてランダムに作成した二値行列を用いた．ベースラインとして，湊らによる $C/C++$ で実装された既存の BDD/ZDD ライブラリ [湊 05] を用いて，集合演算を用いて ZDD を構築する方法を用いた．提案手法は C 言語を用いて実装した．いずれの検証も 1.8GHz Intel Core i7，4GB メモリを搭載した Mac OS X 10.7 上で行った．

それぞれの二値行列からの ZDD の構築に要した時間を表 1 に示す．いずれの行列においても提案手法が高速に計算できていることが確認できる．とくに行列のサイズが大きくなるほど速度の差が大きくなる傾向が見える．

二値行列からの ZDD の構築に要したメモリを表 2 に示す．

表 1: 二値行列を表す ZDD の構築に要した時間

行列サイズ	計算時間 (秒) 提案手法	集合演算
$1,000 \times 1,000$	0.046	0.112
$5,000 \times 5,000$	0.303	1.651
$10,000 \times 10,000$	1.153	8.024

表 2: 二値行列を表す ZDD の構築途中に要したメモリ

行列サイズ	メモリ使用量 (MByte) 提案手法	集合演算
$1,000 \times 1,000$	8	26
$5,000 \times 5,000$	112	258
$10,000 \times 10,000$	444	1,040

いずれの行列においても提案手法が半分程度のメモリ使用量で ZDD の構築を行なっていることが分かる．さらに，この値には入力である二値行列を保持するために必要なメモリ量も含まれており，かつ $10,000 \times 10,000$ の行列では入力データの保持に約 380MByte 程度のメモリを必要としたことから，計算途中に必要とする追加のメモリ量としては提案手法が 1/10 未満であることが分かる．

6. おわりに

本稿では，ベクトルと二値行列との乗算の高速化のために，ZDD を用いて行列を表現する処理において，二値行列から高速に ZDD を生成する手法を示した．行列を ZDD で表現した際に，ZDD の簡約化規則がどのような場面で適用されるかを分析し，行列の各列を一度ずつ処理することで規則適用後の既約な ZDD を得る方法を示した．提案手法を利用することによって，一般的な ZDD 構築時のように集合演算を繰り返し適用するよりも高速に，かつ省メモリで二値行列に対応する ZDD を構築することを可能とした．行列とベクトルの積の計算はさまざまな場面で用いられる基本的な演算であるため，ZDD を用いた計算，および本稿で示した ZDD への高速な変換手続きは有用な技術であると考える．

参考文献

[Bryant 86] Bryant, R. E.: Graph-Based Algorithms for Boolean Function Manipulation, *IEEE Trans. Comput.*, Vol. C-35, No. 8, pp. 677–691 (1986)

[Minato 93] Minato, S.: Zero-Suppressed BDDs for Set Manipulation in Combinatiorial Problems, in Proc. of DAC'93, pp. 272–277 (1993)

[西野 12] 西野 正彬, 安田 宜仁, 小林 透：ZDD を用いた効率的な集合拡張の計算, 人工知能学会論文誌, Vol. 27, No. 2, pp. 22–27 (2012)

[湊 05] 湊 真一：VSOP:ゼロサプレス型 BDD に基づく「重みつき積和集合」計算プログラム, 信学技報 COMP2005-10, Vol. 105, No. 72, pp. 31–38 (2005)