F-024

ベイジアンネットワークを表現する ZDD からの高速計算プログラムの自動生成とその評価

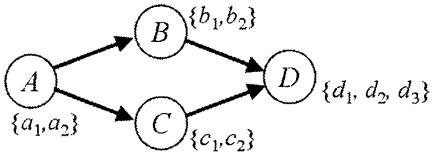
Synthesis of Fast Calculation Programs from ZDDs for Representing Bayesian Networks and Its Evaluation

高橋 渉 †
Wataru Takahashi

湊 真一 †
Shin-ichi Minato

1. はじめに

ベイジアンネットワーク (以後 BN と書く) は，グラフ構造による確率モデルの表現方法の一種であり，近年，様々な用途に広く用いられている [14]．BN では与えられた BN と観測データに対して，BN の一部に観測値を代入する事でネットワーク全体の確率分布を計算する．この確率分布の計算は確率推論と呼ばれる基本的な処理である．機械学習等の応用では，一つの BN に対して異なる観測値を与え，繰り返し推論計算を行う事がしばしばある．従って BN 推論の確率計算の高速化は BN 構造を実問題で活用する上で非常に重要な要素である．このため，様々な確率計算の高速化に関する研究がなされている [4] [5] [6] [7]．湊らは VSLICAD の分野でも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)[3]，その中でも特に「ゼロサプレス型 BDD」(ZDD:Zero-suppressed Binary Decision Diagrams)[11][12] と呼ばれるデータ構造を用いて BN を表現し，効率よく確率計算を行う手法を提案した [13]．本稿では，BN を表現する ZDD のグラフ構造から，C 言語による計算プログラムを自動合成し，これをコンパイルする事により効率の良いサブルーチンを構成する手法について述べ，さらに本手法の性能についての評価を行う．



A	Prb(A)
a_1	$\theta_{a_1} = 0.4$
a_2	$\theta_{a_2} = 0.6$

AB	Prb($B\|A$)
a_1b_1	$\theta_{b_1\|a_1} = 0.2$
a_1b_2	$\theta_{b_2\|a_1} = 0.8$
a_2b_1	$\theta_{b_1\|a_2} = 0.8$
a_2b_2	$\theta_{b_2\|a_2} = 0.2$

AC	Prb($C\|A$)
a_1c_1	$\theta_{c_1\|a_1} = 0.5$
a_1c_2	$\theta_{c_2\|a_1} = 0.5$
a_2c_1	$\theta_{c_1\|a_2} = 0.5$
a_2c_2	$\theta_{c_2\|a_2} = 0.5$

BCD	Prb($D\|B,C$)
$b_1c_1d_1$	$\theta_{d_1\|b_1c_1} = 0.0$
$b_1c_1d_2$	$\theta_{d_2\|b_1c_1} = 0.5$
$b_1c_1d_3$	$\theta_{d_3\|b_1c_1} = 0.5$
$b_1c_2d_1$	$\theta_{d_1\|b_1c_2} = 0.2$
$b_1c_2d_2$	$\theta_{d_2\|b_1c_2} = 0.3$
$b_1c_2d_3$	$\theta_{d_3\|b_1c_2} = 0.5$
$b_2c_1d_1$	$\theta_{d_1\|b_2c_1} = 0.0$
$b_2c_1d_2$	$\theta_{d_2\|b_2c_1} = 0.0$
$b_2c_1d_3$	$\theta_{d_3\|b_2c_1} = 1.0$
$b_2c_2d_1$	$\theta_{d_1\|b_2c_2} = 0.2$
$b_2c_2d_2$	$\theta_{d_2\|b_2c_2} = 0.3$
$b_2c_2d_3$	$\theta_{d_3\|b_2c_2} = 0.5$

図 1: BN 構造と CPT の簡単な例

†北海道大学大学院情報研究科

2. ベイジアンネットワークと MLF 式

BN は図 1 のような非巡回で有向なグラフ構造による確率モデルの表現方法の一つである．各ノード (BN ノードと呼ぶ) はそれぞれ独立した個別の確率変数 X を持っている．X は一般に多値の変数 $\{x_1, x_2, ..., x_k\}$ であり，その中からいずれかの値を取る．また各 BN ノードは，上流側の BN ノードの確率変数の値に依存する条件付き確率テーブル (Conditional Probability Table;CPT) を持っており，確率変数の確率分布が表現されている．BN の確率分布を計算するための方法の 1 つとして，MLF 式というものが知られている．MLF 式は 1 つの確率変数 X に対して，X の確率分布の数値を記号的に表現する λ 変数，そして X の値を表現する θ 変数という 2 種類の論理変数を使って表現する．以下に図 1 の BN を MLF 式で表現したものを示す．

$$
\begin{aligned}
& \lambda_{a1}\,\lambda_{b1}\,\lambda_{c1}\,\lambda_{d1}\,\theta_{a1}\,\theta_{b1|a1}\,\theta_{c1|a1}\,\theta_{d1|b1c1} \\
+\ & \lambda_{a1}\,\lambda_{b1}\,\lambda_{c1}\,\lambda_{d2}\,\theta_{a1}\,\theta_{b1|a1}\,\theta_{c1|a1}\,\theta_{d2|b1c1} \\
+\ & \lambda_{a1}\,\lambda_{b1}\,\lambda_{c1}\,\lambda_{d3}\,\theta_{a1}\,\theta_{b1|a1}\,\theta_{c1|a1}\,\theta_{d3|b1c1} \\
+\ & \lambda_{a1}\,\lambda_{b1}\,\lambda_{c2}\,\lambda_{d1}\,\theta_{a1}\,\theta_{b1|a1}\,\theta_{c2|a1}\,\theta_{d1|b1c2} \\
+\ & \cdots \\
+\ & \lambda_{a2}\,\lambda_{b2}\,\lambda_{c2}\,\lambda_{d3}\,\theta_{a2}\,\theta_{b2|a2}\,\theta_{c2|a2}\,\theta_{d3|b2c2}
\end{aligned}
$$

MLF 式はこのように，全ての組合せの条件付き確率を書き下した算術式の形をとっている．

与えられた BN の MLF 式を生成する事で，ある観測データに対する確率変数の確率分布を自動的に求める事が出来る．この確率計算に要する時間は，MLF 式の長さに比例するが，一般に MLF 式は元の BN サイズに対して指数的に増大する為，計算するのに非常に多くの時間が掛かる．ただし，MLF 式を因数分解して，コンパクトな算術式として表現することで確率計算を高速化する事が出来る．本研究ではこの因数分解を行う一つの手段として，ZDD を用いた手法を使う．ZDD を用いた手法については，次章で詳しく説明する．

3. ZDD によるベイジアンネットワークの表現

3.1 ZDD

ZDD は組み合わせ集合データの処理に特化された BDD である．BDD とは二分木グラフを規約する事で得られる論理関数のグラフによる表現であり，また組合せ集合とは「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．通常の BDD では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，等価な節点を共有する，という 2 つの縮約規則があるが，ZDD では冗長な節点を取り除くことはせず，その代わりに 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．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，特に疎な組合せ集合に対して通常の 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．BDD，及び ZDD についての詳細は [8, 11] を参照のこと．

3.2 ZDD による BN 表現

MLF 式は λ 変数と θ 変数の値からなる多項式で，各項が単に変数の組合せであるため，MLF 式は一種の組合せ集合とみなすことが出来る．よって ZDD としてコンパクトに表現することができ，また確率計算においても ZDD サイズに比例した時間で行うことが可能になる．ここで図 1 のノード B についての MLF 式を例に見てみると，以下のようになる．

$$\begin{aligned} MLF(B) &= \lambda_{a1}\lambda_{b1}\theta_{a1}\theta_{b1|a1} + \lambda_{a1}\lambda_{b2}\theta_{a1}\theta_{b2|a1} \\ &+ \lambda_{a2}\lambda_{b1}\theta_{a2}\theta_{b1|a2} + \lambda_{a2}\lambda_{b2}\theta_{a2}\theta_{b2|a2} \end{aligned}$$

ここで，変数の個数を節約するため，確率変数が同じ確率値を持つ場合には，変数を共有するように θ 変数を置き換えると

$$\begin{aligned} MLF(B) &= \lambda_{a1}\lambda_{b1}\theta_{a(0.4)}\theta_{b(0.2)} + \lambda_{a1}\lambda_{b2}\theta_{a(0.4)}\theta_{b(0.8)} \\ &+ \lambda_{a2}\lambda_{b1}\theta_{a(0.6)}\theta_{b(0.8)} + \lambda_{a2}\lambda_{b2}\theta_{a(0.6)}\theta_{b(0.2)} \end{aligned}$$

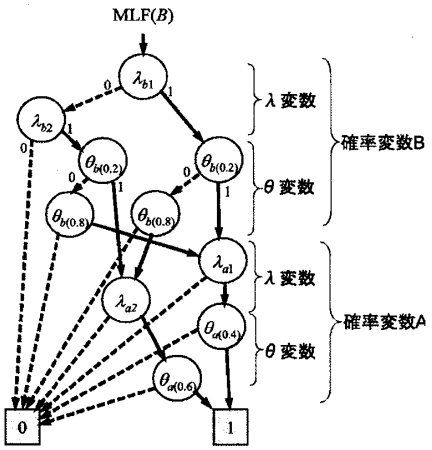


図 2: MLF(B) の ZDD 表現の例

となる．図 2 には MLF(B) による ZDD の一例を示した．この例では最上位の節点から 1 の終端節点までに 4 通りの経路がある．それぞれの変数は MLF 式の項に担当しており，これは MLF 式の暗黙の表現である．ZDD の構造は節点を共有する事で MLF 式を更に因数分解したコンパクトな形に表現しており，指数関数的に MLF 式が巨大になったとしても，類似した部分を共有する事で場合によっては数十倍もの縮約効果が得られる．なお，MLF 式は一つの確率変数が複数の λ 変数と θ 変数を持つ為，図 2 のように同じ変数に対する λ，θ 変数を並べて配置している．このようにして生成された ZDD は，MLF 式を因数分解した多段の算術式と解釈する事が出来る．各節点は下位の節点の計算結果を用いた算術乗算と加算を表している．これは MLF 式の ZDD を生成した後に，ZDD のサイズに比例する回数の算術演算手順が容易に得られる事を意味している．

また，BN が中規模以上になると，ZDD を用いても MLF 式を表現する事が困難になるが，変数の順序付けを行い ZDD サイズをコンパクトにする事で，より大規模な例題に適用する事が出来る．BN から適当な順序で作った ZDD の変数順を改良して，よりよいコンパクトな ZDD を求める逐次改善法 [9] や，ZDD を生成する前に BN の構造情報から比較的良さそうな変数順を求める発見的手法 [10] などの，変数順序に関する研究も行われている．

4. 自動生成プログラム Z2C

本研究で作成した自動生成プログラム Z2C は，ZDD のグラフ構造から対応する MLF 式の計算を行い，その値を返す C コードを出力とするプログラムである．Z2C では 1 行が 1 ノードに対応している .zdd ファイルの情報を読み込み C コードを出力する．.zdd ファイルには ZDD のグラフ構造しか記述されていないため，λ 変数及び θ 変数の値は呼び出し側から設定する必要がある．

Z2C にて変換された C コードのデータ構造は主に二つの関数に分けられる．一つは θ 変数と λ 変数を格納する配列を宣言し，計算結果を出力するメイン関数，もう一つは ZDD の計算をサブルーチン化した MLF 関数である．.zdd 形式と C コードを比較したものを図 3 に示す．同図より、1 行目は変数を格納する double 型配列の宣言，3 行目では得られたノード数と同じ数の double 型変数が，ノード ID と同名で宣言されていることが分かる．4 行目以降はそれぞれに対応した計算式を記述しており，最後の行で出力するノード ID を返している事が分かる．C コードの計算式は .zdd ファイルと同様，1 ノード 1 行に対応している．

以上が Z2C の説明である．次章ではこの Z2C を用いて出力された C コードを使った実験を行い，考察していく．

```
_i 8              →   double x[8] = {0.4,0.6,...,0,1};
_o 1
_n 10             →   double  f216, f220, ..., f288;
216 1 F T         →   f216   = 0      +  x[0]  *  1 ;
220 3 F 216       →   f220   = 0      +  x[2]  *  f216 ;
230 5 F 220       →   f230   = 0      +  x[4]  *  f220 ;
222 2 F T         →   f222   = 0      +  x[1]  *  1 ;
226 4 F 222       →   f226   = 0      +  x[3]  *  f222 ;
236 6 230 226     →   f236   = f230   +  x[5]  *  f226 ;
240 7 F 236       →   f240   = 0      +  x[6]  *  f236 ;
244 5 F 226       →   f244   = 0      +  x[4]  *  f226 ;
246 6 244 220     →   f246   = f244   +  x[5]  *  f220 ;
288 8 240 246     →   f288   = f240   +  x[7]  *  f246 ;
288               →   return f288;
```

図 3: .zdd コード (左) と C コード (右) の比較

5. 実験結果と考察

本実験において使用した PC は intel Core(TM)2 duo, 2.20Ghz, Suse Linux 11.2(i586), 主記憶 2Gbyte で，ZDD の最大節点数は 3,000 万個とした．また，生成され

表 1: ノード数，バイト数，処理時間の関係

BN name	変数の数	ZDD ノード数	C コードバイト数	コンパイル時間 (秒)	実行ファイルバイト数	実行時間 (秒)(千回呼出)†
water	3,031	16,975	780,085	3.405	487,615	0.103
alarm	292	19,289	833,879	3.814	524,445	0.091
hailfinder-a	740	108,275	5,020,304	28.116	2,922,344	1.243
Munin3-a	19,283	140,166	7,782,729	40.167	3,897,237	1.637
hailfinder-b	740	231,553	10,835,274	81.631	6,231,910	2.664
insurance	399	247,778	11,735,014	88.686	6,702,949	3.049
Munin2-a	10,922	303,308	15,485,237	137.443	8,181,649	3.730
Munin2-b	10,928	443,632	22,739,230	359.185	11,954,072	5.428
Munin3-b	4,643	1,003,502	×	—	—	—
Munin3-c	10,428	2,613,497	×	—	—	—

たCコードのコンパイルには gcc version 4.4.1 を，BN データは代表的なベンチマーク例題 [1] を使用した．また，本稿では ZDD のサブルーチン化とそれを用いた実験を主旨とするため，λ 変数及び θ 変数の値は全て 0.5 としている．

5.1 自動生成とコンパイルについての実験

本実験では，まずノード数の異なる .zdd ファイルを用いて，ノード数が増大しても Z2C が .zdd ファイルから C コードを生成出来るかを調べた．Z2C を用いた C コードの生成に関しては，単純なフォーマット変換であるので，およそ 260 万個ものノード数にもなる ZDD においても高速に（約 6 秒）C コードを生成する事が可能であった．この事から，Z2C による自動生成は後述するコンパイル時間に比べれば無視できるほど短い時間で生成が出来る事が分かった．

次に gcc を用いて，変換された C コードに対してコンパイルを行い，機械語に変換出来るかを調べた．表 1 をみると，gcc を利用した C コードのコンパイルでは，計算式が約 44 万行までの C コードはコンパイルに成功したが，約 100 万行以上のコードはコンパイルに失敗した．これはコードの増大によってメモリ容量が不足する事で起こるエラーであると推測される．また同表から，C コードのファイルサイズは入力変数の数には殆ど影響されず，総じてノード数に比例して増加している事が分かる．これは ZDD のノード数と C コードの計算式が 1 ノード 1 行に対応しているため，ノード数の増加がすなわち計算式の増加になるためである．

C コードのバイト数に関しては，生成元である ZDD のノード数のおよそ 50 倍である事から，Z2C で生成するにはノード 1 個につき約 50byte のファイル容量が必要になる事が分かる．また，実行ファイルのバイト数は元の C コードのバイト数の約 1/2 となった．

次に C コードのコンパイル時間を見ると，グラフ図 4 に示したように，コンパイルに要する時間は線形時間よりも若干長い時間が掛かっている事が観測される．また，実行ファイルの実行時間も同様にノード数に応じてほぼ線形で増加していくが，こちらはいずれも 1 秒に満たない僅かな時間で計算可能である．

5.2 最適化オプション付加実験

gcc でコンパイルに成功した C コードに，最適化オプション [2] をそれぞれ適用してコンパイルを行った．hailfinder-a を例に最適化オプションを付けた場合と付けなかった場合のそれぞれについて比較したものを

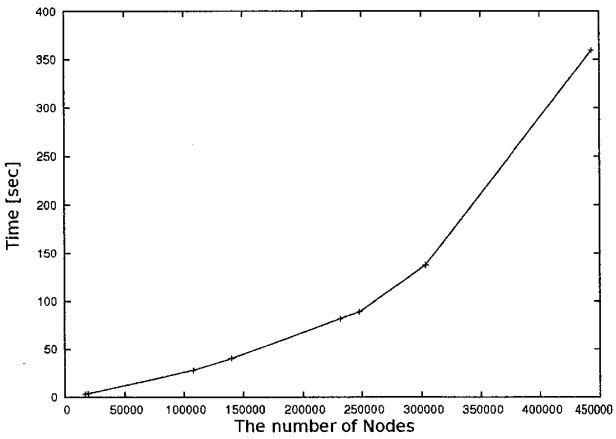


図 4: ノード数とコンパイル時間の関係

表 2 に示す．同表より，最適化オプションを付ける事によってファイルサイズ，及び実行時間は半分以下に削減する事が出来たが，コンパイル時間は約 9 倍〜25 倍に増大している事が分かる．実際にプログラムを利用する事を想定すると，実行ファイルを膨大な回数実行する必要がある場合には処理時間の総合値が削減出来る可能性がある．表 2 の場合においては，どの最適化オプションも 1 万回実行ファイルを実行する事で，オプションを適用しない場合に比べて約 10 秒実行時間を削減出来る事が分かる．特に “-O” オプションを利用する場合であれば，およそ 20 万回ほど実行ファイルを実行すれば，コンパイル時間に掛かったロスを殆ど回収する事が出来る．

表 2: ノード数 108,275 の場合のコンパイル時間，バイト数，実行時間の関係

オプション	コンパイル時間 (秒)(1 回呼出)	実行ファイルバイト数	実行時間の平均 (秒)‡
—	28.116	2,922,344	12.198×10^{-4}
-O	239.253	1,419,113	2.500×10^{-4}
“-O2”	700.432	1,464,169	2.462×10^{-4}
“-O3”	700.488	1,464,169	2.266×10^{-4}
“-Os”	590.695	1,460,072	2.308×10^{-4}

†変換した C コードに，MLF 関数を 1000 回呼び出す for 文を追加してコンパイルしたものを実行．

‡サブルーチンを 2 万回呼び出して計算させた場合の計算 1 回あたりの平均時間

5.3 他の手法との速度比較

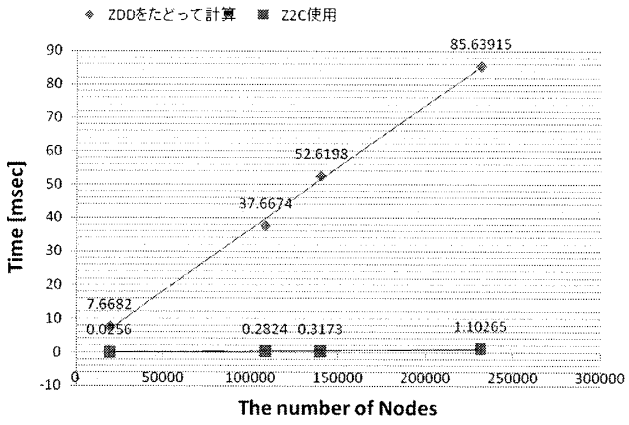


図 5: Z2C と直接算術演算プログラムとの速度比較

ここでは，Z2C の計算処理の速度を，メモリ上に確保された ZDD を深さ優先で辿りながら直接算術演算を行う手法と比べ，どの程度計算処理を高速化出来たかの評価を行う．図 5 では，19,289 個 (alarm)，108,275 個 (hailfinder-a)，140,166 個 (Munin3-a)，231,553 個 (hailfinder-b) の .zdd ファイルを用いて，Z2C で計算処理を 2 万回行った場合と，湊らが開発した既存の BDD パッケージを用いて，直接算術演算を行うプログラムを作成し，計算処理を 2 万回行った場合の実行時間から，1 回処理を行うのに掛かる時間を求め，それらの近似直線をプロットしたグラフである．図が示す通り，Z2C で処理を行った場合では実行時間が少なくとも数十倍短縮されている事が分かる．この事から，直接演算処理を行う場合に比べ，C コードをコンパイルし機械語にする事で，高速化に成功していると言える．

6. おわりに

本稿では BN を実問題に適用するための準備として，人手では扱いきれない膨大な計算を行うために高速計算プログラムの自動生成を行った．今回作成した Z2C と gcc を利用する事で，一度実行ファイルを生成した後はかなり大きな ZDD においても高速に計算できる事が出来た．また実行回数が多い場合には最適化オプションを適用してコンパイルを行う事で，ほぼ同じコンパイル時間で実行時間の短縮が行えた．メモリ上の ZDD を辿りながら直接算術演算をおこなう場合との速度比較においては，Z2C が優位である事を示した．今後の課題としては，ノード数が一定の個数を超えると C コードをコンパイル出来ないという問題の解決と，λ 変数や θ 変数の値を変化させながら代入するという機能の実装である．今後はこれらのプログラムに関する問題を改善していきながら，実際にこの計算プログラムを用いて，ベイジアンネットワークを積極的に取り入れようとしている分野において実際に計算を行い，実問題に対して処理を行わせる事でどれほどの高速化が出来るのかというような研究を行っていきたいと思っている．

参考文献

[1] Bayesian Network Repository, http://www.cs.huji.ac.il/labs/compbio/Repository/

[2] Manpage of GCC, http://www.linux.or.jp/JM/html/GNU_gcc/man1/gcc.1.html

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

[4] M. Chavira and A. Darwhe, “Compiling Bayesian Networks with Local Structure”, In Proc. 19th International Joint Conference on Artificial Intelligence(IJCAI-2005), pp-1306-1312, Aug. 2005.

[5] A. Darwiche, “A logical approach to factoring belief networks”, In Proc. KR, pp.409-420, 2002.

[6] A. Darwiche, “A differential approach to inference in Bayesian networks”, JACM, Vol.50, No.3, pp.280-305, 2003.

[7] A. Darwiche, “New advances in compiling CNF to decomposable negational normal form”, In Proc. European Conference on Artificial Intelligence(ECAI-2004), pp. 328-332, 2004.

[8] 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.

[9] 礒松 紘平, 湊 真一, ベイジアンネットワークを表現するゼロサプレス型 BDD の変数順序付け方法に関する考察, 人工知能基本問題研究会 72, 49〜53, 2008/11/7・8

[10] 金崎健之, 湊真一, ベイジアンネットワークを表現する zdd の初期変数順序付け方法の改良 FIT 2009 IEICE/IPSJ 第 8 回 情報科学技術フォーラム, F-061, pp. 553-555, 2009.

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

[12] Minato, S., 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.

[13] S. Minato, K. Satoh, and T.Sato. “Compiling Bayesian Networks by Symbolic Probability Calculation Based on Zero-suppressed BDDs”, In Proc. 19th International Joint Conference on Artificial Intelligence (IJCAI-2005), pp. 2550-2555, Aug. 2005.

[14] R. E. Neapolitan. *LearningBayesianNetworks*. Pearson Education, Inc., 2004.