BDD/ZDDを用いたグラフ列挙索引化技法

湊　真一

本稿では，与えられたグラフ構造のなかから，ある制約条件を満たすような部分グラフ構造をすべて列挙し，それらをBDD/ZDDを用いて圧縮表現して索引化する技法について述べる．まず，BDD/ZDDとその演算処理系について簡単に説明し，次に，ZDDを用いた高速なパス列挙索引化アルゴリズムSimpathの概要を解説する．このアルゴリズムはKnuthが近年提案したものであるが，制約条件によって多くのバリエーションが存在することから，われわれはそれらを総称して「フロンティア法」と呼んでいる．本稿ではフロンティア法がなぜ高速であるか，そして従来のBDD/ZDDを用いた制約充足問題の解法との違いについて述べる．

キーワード：BDD, ZDD，グラフ列挙，索引化

1. はじめに

あるグラフ構造が与えられたときに，そのなかから制約条件を満たすような部分グラフ構造を抽出することは，現実社会のさまざまな局面に現れる基本的かつ重要な手続きである．本稿では，種々の制約条件を満たす部分グラフ構造をすべて列挙し，それらをZDD（ゼロサプレス型二分決定グラフ）と呼ばれるデータ構造を用いて圧縮表現して索引化する技法について述べる．まず，BDD（二分決定グラフ）と，その改良形であるZDDについて簡単に説明し，次に，ZDDを用いた高速なパス列挙索引化アルゴリズムである「Simpath」の概要を解説する．このアルゴリズムはKnuthが近年，彼の著書において提案したものであるが，制約条件の種類に応じて多くのバリエーションが存在することから，われわれはそれらを総称して「フロンティア法」と呼んでいる．本稿ではフロンティア法がなぜ高速であるか，そして従来のBDD/ZDDを用いた制約充足問題の解法との違いについて述べる．

2. BDD/ZDDとグラフ列挙索引化

BDD（Binary Decision Diagram; 二分決定グラフ）は，グラフ構造による論理関数の表現である．$F(a,b,c) = a\bar{b}c \vee \bar{a}b\bar{c}$ を表現した例を図1に示す．これは，論理関数の値をすべての変数について場合分けした結果を二分決定木で表し，これを縮約することにより得られる．このとき，場合分けする変数の順序を固定し，「0でも1でも分岐先が同じ場合は分岐節点を削除して直結」「共通の行き先をもつ分岐節点が2個以上あれば1個にまとめて他を削除」という2種類の縮約処理を可能な限り行うことにより「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[1]．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ば，互いにサブグラフを共有することが可能であり，1つのメモリ空間の中で多数の論理関数データをコンパクトに圧縮して索引化することができる．

BDDを構築する際に，まず二分決定木を生成してからそれを圧縮したのでは，常に指数関数的な時間と記憶量を要するため現実的でない．これに対して，BDD同士の二項論理演算（AND, ORなど）の結果を表すBDDを直接生成するアルゴリズム（通称Apply演算）[2]がBryantにより提案され，以降，BDDが広く用いられるようになった．Apply演算は，圧縮されたデータ量にほぼ比例する計算時間で実行できる[1]．つまり，圧縮データを元に戻すことなく，圧縮したままで高速に演算処理できるという優れた特長がある．ほとんどのBDDの応用では，このApply演算を繰り返し適用して所望のBDDを構築している（日本語の解説記事としては文献[11]がある）．

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「組合せ集合」を表現することもできる．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．組合せ集

みなと　しんいち
北海道大学/JST ERATO
〒060–0814 北海道札幌市北区北14条西9丁目

[1] 入力サイズと出力サイズの和に関して線形時間であると予想されていたが，最近，反証された[8]．ただし，ほとんどの実用的な問題では線形時間と考えて差し支えない．

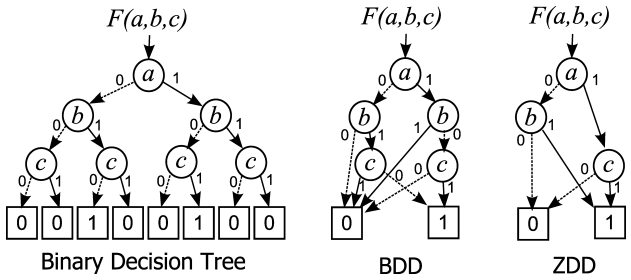

図 1　二分決定木，BDD, ZDD

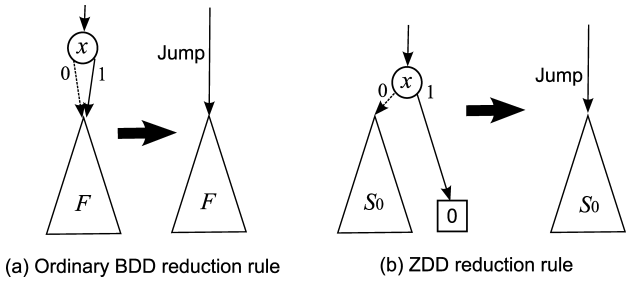

図 3　BDD, ZDD の簡約化規則

a	b	c	F	
0	0	0	0	$\to S$
0	0	1	0	
0	1	0	1	$\to b$
0	1	1	0	
1	0	0	0	
1	0	1	1	$\to ac$
1	1	0	0	
1	1	1	0	

論理関数とみた場合:
$F = a\bar{b}c \vee \bar{a}b\bar{c}$

組合せ集合とみた場合:
$S = \{ac,\ b\}$

図 2　論理関数と組合せ集合の対応

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，購入履歴データベース，Web のリンクの表現，システム故障要因の表現等，さまざまな局面で現れる．ある 1 つの組合せ集合は，1 つの論理関数（特性関数と呼ばれる）に対応づけることができる．例えば，図 2 は $a\bar{b}c \vee \bar{a}b\bar{c}$ 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 $\{ac,\ b\}$ という組合せ集合も表現している．特性関数を 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，共通集合 (intersection)/和集合 (union) の演算にそのまま対応するので，多数の組合せ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．類似した組合せが多く出現する場合，BDD では等価なサブグラフが多く出現し，それらが互いに共有されて，記憶量や計算時間が大幅に（ときには指数関数的に）削減できる場合がある．

ZDD（Zero-suppressed BDD; ゼロサプレス型 BDD）[5] は，組合せ集合データの処理に特化された BDD の変化形である．ZDD では，等価な節点を共有する規則は通常の BDD と同様であるが，冗長な節点を削除する際の規則が異なる．すなわち，図 3 に示すように，1–枝が 0–終端節点を直接指している場合に，この節点を取り除く．その代わり，通常の BDD で削除されるような節点は削除しない．このような ZDD の簡約化規則によっても表現の一意性は保たれる．ZDD を構築する方法として，通常の BDD を構築する場合と同様に，Apply 演算により 2 つの ZDD 同士の集合演算（和集合，共通集合，差集合など）を実行し，その計算結果の ZDD を得る方法がある．

ZDD は，疎な組合せの集合に対して顕著な効果がある．例えば，組合せ集合の各要素に含まれるアイテムの平均出現頻度が 1%であれば，ZDD は BDD よりも 100 倍コンパクトになる可能性がある．現実の応用でもそのような事例はしばしば見られる（例えば，店舗に並ぶ商品総数に比べて，顧客が 1 度に購入するアイテム数は極めて少ない）．ZDD は，BDD の種々の変化形のなかで，最も重要なものとして認識されており，Knuth の有名な教科書 “The Art of Computer Programming” の最新巻 [4] でも詳しく解説されている．

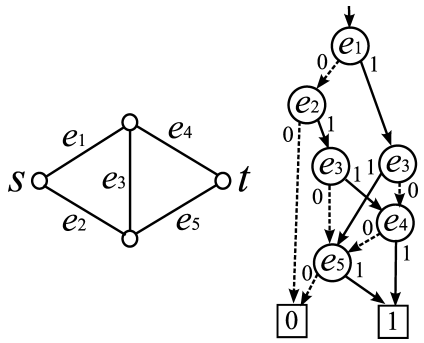

図 4　s,t 間のパスを列挙する ZDD

BDD/ZDD は，グラフを列挙索引化するためのデータ構造としても有用である．節点集合 $V=\{v_1,v_2,\ldots,v_n\}$，辺集合 $E=\{e_1,e_2,\ldots,e_m\}$ からなるグラフ $G=(V,E)$ を考えると，グラフ列挙の問題とは，E のべき集合 2^E（または V のべき集合 2^V）の中から，ある条件を満たすような部分集合を求めることであり，これは解集合を辺（または節点）の組合せの集合と考えれば，そのまま BDD/ZDD で表現することができる．例えば，図 4 ではグラフの 2 節点 s,t を結ぶパスの集合を表す ZDD を示している．この例では s,t 間のパスは 4 通り存在するが，それぞれのパスを辺の組合せと考えれば $\{e_1e_4,\ e_1e_3e_5,\ e_2e_5,\ e_2e_3e_4\}$ という組合せの集合として解集合を表現できる．これを表す ZDD では，最上位の根の節点から 1 の終端節点に至るパスが 4 通り存在し，それぞれがこの問題の解に対応していることが確かめられる．

グラフの部分集合を列挙する BDD/ZDD を構築するには，まず列挙したいグラフの制約条件を論理演算や集合演算でうまく定式化すればよい．あとはその式に従って Apply 演算を繰り返し適用して効率よく BDD/ZDD を生成し，求めるグラフを列挙索引化できる [3]．そのようにして BDD/ZDD をいったん構築してしまえば，列挙した解の個数をすばやく数え上げたり，さらに条件をつけて解を絞り込むといった操作が容易になる．

グラフ列挙の問題で，BDD と ZDD のどちらが適しているかは，問題の性質に依存する．まず両者に共通する性質として，部分的に類似した組合せが多数生じるほど，BDD/ZDD の部分的な共有が頻繁に起きて圧縮度が高くなる．そのうえで，ある辺の有無を決めると別の辺を使っても使わなくてもどちらでもよくなることが多い問題では BDD が有利であり，ある辺を使うと決めたら別の辺が使えなくなることが多い問題では ZDD が有利である．種々のグラフ列挙の問題では，どちらかというと ZDD が有利な場合が多いと考えられる．

3. Knuth のパス列挙アルゴリズム

先に述べた Knuth の教科書 (Vol. 4 Fascicle 1)[4] の p. 121（Vol. 4A では p. 254) において，与えられたグラフの 2 点 s,t を結ぶパス（遠回りを許すが同じ節点は 2 度通らない）を全列挙するアルゴリズム Simpath (Simple Paths) が示されている．これは，ちょうど s–t パスとなるような辺の組合せを全列挙した集合を表す ZDD を作るアルゴリズムである．Knuth 本人が実装したソースコードが氏のホームページで公開されているが，試してみるとこれが驚くほど高速である．例えば 15×15 の格子グラフ（辺の総数 420）の左上隅から右下隅に至る s–t パスを全列挙する ZDD を生成させると，パスの総数は実に 227449714676812739631826459327989863387613323440（2.27×10^{47}) 通りもあるが，ZDD の節点数はたった 144,759,636 個ですんでおり，その ZDD 生成に要する時間はわずか数分である．

Simpath アルゴリズムは，Apply 演算を用いることなく，与えられたグラフから解集合の ZDD を直接生成するアルゴリズムである．従来の常識では，ある特定の問題専用の BDD/ZDD 生成アルゴリズムを一から実装するよりも，Apply 演算を組合せて実装する方が工学的にはエレガントであり，速度の面で多少のオーバヘッドがあってもそちらのほうが好ましいと考えていた．そこでわれわれは，Apply 演算だけを組合せて s–t パス列挙の ZDD を生成するアルゴリズムを作って Simpath と比較してみたが，相当がんばって工夫しても，Simpath 法のほうが圧倒的に（例題にもよるが数十倍以上も）速く，単なるコーディング技量の違いでは済まされないことがわかってきた．

図 5 を用いて，Simpath アルゴリズムの概略を紹介する．最初に与えられたグラフ G の辺に適当な順序をつけ，$E=\{e_1,e_2,\ldots,e_m\}$ とする．これより，上位か

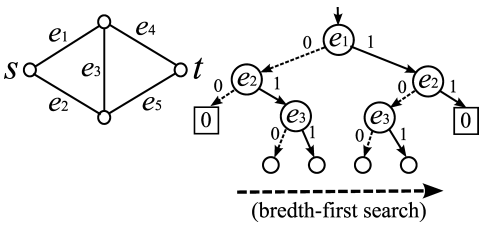

図 5　Simpath アルゴリズムの処理手順

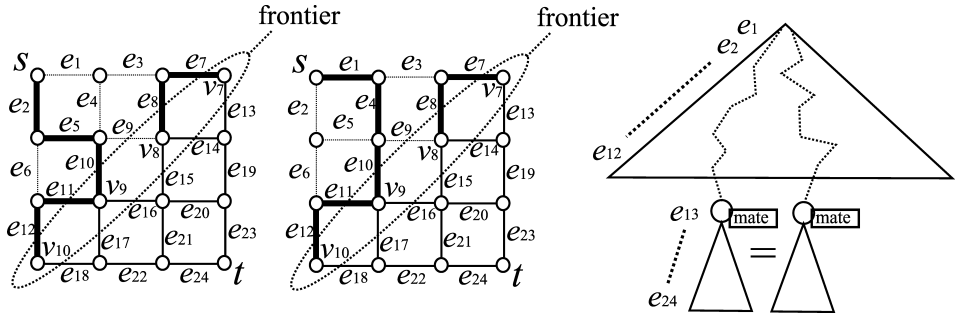

図 6　等価な途中状態の例

ら $e_1, e_2, \ldots$ の変数順の二分決定木を，上位から下位に向かって幅優先順でたどりながら構築して行く．まずグラフ G において，s–t パスが辺 e_1 を通らない場合を 0，通る場合を 1 として場合分けし，e_1 の変数名をもつ分岐節点を 1 個作り，0–枝，1–枝の先の二分決定木の葉には，それぞれについてグラフ G の途中状態を記録した情報を記録しておく．次に，それぞれの葉を順番に訪問し，記録されていた途中状態を見ながら s–t パスが辺 e_2 を通るかどうかでさらに場合分けして e_2 による分岐節点を継ぎ足して，それぞれの葉には新しい途中状態を記録する．このようにして上から k 段目までの二分決定木の葉を順番にたどって，記録されていた途中状態を見ながら e_{k+1} による分岐節点を継ぎ足して，$(k+1)$ 段目の二分決定木を作る，という処理を繰り返し行う．ただし途中で明らかに s–t パスにならないような辺の組合せ（非連結になったり分岐したりするなど）が生じた場合は，葉に 0 と記録し，以降の段ではそれ以上分岐させないことにする．最後に辺 e_m まで場合分けが終わったら，ちょうど s–t パスとなっている葉に 1 を記録すれば，解集合を表す二分決定木が完成する．その後，この二分決定木を下位から上位に向かって幅優先でたどりながら ZDD の縮約規則を適用していくと，最終的に求める ZDD が得られる．

上記の手続きでは，途中で葉に 0 を書き込んだ分だけ無駄な計算を省いているが，それだけではまだ十分ではない．Simpath アルゴリズムでは，二分決定木を構築する際に，「等価な途中状態」をもつ複数の葉を 1 つに共有させることで，計算量の大幅な圧縮を実現している．ここで，等価な途中状態とは，残りの辺の取捨選択によって s–t パスになるか否かが全く同じ結果となるような辺の組合せのことを指す．図 6 に格子グラフのパスを列挙する場合の途中状態の例を示す．辺 e_1 から e_{12} までの場合分けが終わっているとして，$(e_2, e_5, e_7, e_8, e_{10}, e_{11}, e_{12})$ を選択した場合（左側の格子グラフ）と $(e_1, e_4, e_7, e_8, e_{10}, e_{11}, e_{12})$ を選択した場合（右側の格子グラフ）とで途中状態を比較すると，残りの辺で節点 v_8 と v_{10} を連結し，さらに v_7 と終点 t が連結できれば良いということで両者は一致しており，以後の辺の取捨選択の制約条件は全く同じであることがわかる．これを右側の ZDD の模式図に当てはめると，e_{13} より下位のサブグラフが完全に一致するということであり，このように途中状態が等価であると判れば，e_{13} のレベルで節点を共有してしまうことができる．s–t パス列挙問題では，このような等価な途中状態が非常に多く発生するため，この共有の効果は大きい．途中状態としてどのような情報を記憶しておけばよいかというと，図中の破線で囲まれている部分の各節点が端点か否かということと，端点だった場合にもう一方の端点はどこか，ということだけがわかればよい．Simpath ではこの情報を mate と呼ぶ配列構造でコンパクトに表現し，ハッシュテーブルに登録することで高速に一致判定を行っている．さらに，上記の破線で囲まれている節点，すなわち，処理済みの辺と未処理の辺の両方に接続している節点の集合を Knuth はフロンティア (frontier) と呼び，このフロンティアを始点から終点まで徐々に移動させながら，必要最小限の途中状態を記憶して処理を行っている．

Simpath はフロンティアの mate 情報を途中状態とする動的計画法の一種である．計算途中でフロンティアが膨らむと，互いに異なる途中状態が多く発生して計算量が増大するため，フロンティアがなるべく小さいままで処理が進むことが望ましい．フロンティアが小さくてすむかどうかは，与えられたグラフの形と，それに応じた辺の変数順序付けに依存する．平面グラフに近くて細長い形のグラフであるほうが都合がよい．

Knuth は Simpath の一部を変更するだけで，s–t パスだけでなく，ハミルトンパス，有向パス，さらに各

種のサイクルの列挙もほぼ同様に実行できることを示している．ほかにも mate 情報の仕組みを拡張することで，多くの類似問題に適用できる．

4. Tutte 多項式を表す BDD との関係

グラフ理論における基本的な不変量として Tutte 多項式が知られている．グラフ $G=(V,E)$ の Tutte 多項式は，以下の式で定義される [7].

$$T(x,y)=\sum_{A\subseteq E}(x-1)^{\rho(E)-\rho(A)}(y-1)^{|A|-\rho(A)}$$

ここで $\rho(A)$ は，グラフの節点数 $|V|$ から辺集合 A の連結成分の個数を引いたものである．Tutte 多項式が得られれば，グラフの大域木の総数やグラフの連結確率計算など，グラフの連結性に関係する種々の性質がわかるため非常に強力であるが，そのためにはすべての辺部分集合について上記の式を展開して計算する必要があり，中規模以上のグラフで Tutte 多項式の係数を正確に求めることは容易ではない．

関根，今井ら [6, 10] は，与えられたグラフの Tutte 多項式を表す BDD を効率よく生成するアルゴリズムを 1995 年頃にすでに提案しているが，これが Knuth が提案した Simpath と極めて類似していることがわかってきた[2]．トップダウンに幅優先で辺の有無を場合分けして BDD を構築していくことと，フロンティアに相当する概念を消去波面と呼び，その途中状態によって処理をまとめて計算量を抑えるという点では，全く同一の手法と言ってよい．主な相違点は，ZDD ではなく BDD を用いていることと，フロンティアの途中状態として mate 情報を記録するのではなく，各節点がどの連結成分に所属しているか（すなわち節点集合の分割の情報）を記録していることである．

さらに関根らは，平面グラフや $n\times n$ 格子グラフなどに対する理論的な計算量解析も行っており，そのようなグラフでは，フロンティアの最大サイズが小さく抑えられ，それを得るための変数順序付けも容易に得られることを示している．そして生成される BDD のサイズや時間計算量についても理論的な評価を行っている [10]．つまり Simpath が効率よく動作するようなグラフに関する理論的考察は，90 年代にすでにかなり行われていたことになる．

しかし，当時は計算機の性能が今より貧弱だったことや，BDD の研究そのものが成熟したと思われていた時期でもあったため，フロンティア法による BDD のトップダウンな生成方法が，常識的な Apply 演算による方法よりも優れているかどうかを，さまざまな実問題に対する実測値で評価するというところまで至っていなかった．近年になって，Knuth が極めて高性能なコードを実装して公開したことがきっかけとなり，本アプローチが理論的な興味だけではなく，動的計画法に基づく極めて高性能な列挙索引化の基盤技術として利用できることが明らかとなり，実問題への応用が始まっている．

[2] 筆者らは今井先生から指摘されるまで両手法の類似性を認識していなかった．Knuth 先生もそれを知らずに Simpath を書いたと思われる．

5. フロンティア法としての一般化と工学的応用

以上で述べたように，グラフをたどりながら ZDD をトップダウンで幅優先で生成する技法は，パスを列挙する Simpath だけでなく，さまざまな列挙問題に応用することができる．われわれはそのようなアプローチを総称してフロンティア法と呼ぶことにした．これまでのところ，次のような問題に適用できそうだということがわかっている（より詳細な実装技術については本特集記事 [12] を参照いただきたい）．

- **Simpath の派生問題（端点の情報を記録）**
  k 組の s–t パス集合の列挙（ナンバーリンク問題），サイクルの列挙，通り道に制約のあるサイクルの列挙，長さ k 以下または以上のサイクルの列挙，ハミルトンパス・サイクルの列挙，オイラーパス・サイクルの列挙，有向グラフのパス・サイクル列挙
- **Tutte 多項式計算の派生問題（各節点の分割を記録）**
  連結部分グラフの列挙，大域木/林の列挙，カットセットの列挙，グラフの k 分割問題，連結確率の計算
- **上記以外（問題ごとに途中状態の情報が異なる）**
  クリーク列挙（独立集合列挙），擬似クリーク列挙，グラフ彩色問題，集合被覆問題，タイル敷き詰め問題，部分文字列の列挙，完全・不完全マッチングの列挙

これらのグラフの列挙問題は社会的に重要なさまざまな実問題と関係している．例えばパス列挙は，地理情報システムでは中心的な問題であるし，有向グラフでの列挙は，大規模システムの依存関係の解析や，フローチャートの解析，ナンバーリンクやスリザーリンクのパズル，文字列集合から「しりとり」を完成させる問題など，多くの応用がある．さらに，グラフ k 分

割問題に関しては，電力網を k 箇所の変電所でカバーして配電する区割りの列挙に活用できるし，ほかにも避難所の配置問題や選挙区割り問題など社会インフラに関する重要な応用が多く考えられる．しかも電力網や道路網のような社会インフラを構成するシステムの構造は，平面グラフや格子グラフに近い形であることが多いので，フロンティア法による圧縮効果が極めて高くなる傾向がある．実際，電力網の現実的なモデルで変電所の給電パタン候補を列挙する問題をグラフ k 分割問題で定式化して実験したところ，10^{71} 通りもの膨大な給電パタンを全列挙する ZDD をわずか 1 秒程度で生成できることがわかった（詳細は本特集記事 [9] に記載）．

フロンティア法は，ある 1 つの問題に対してアルゴリズムを作ればそのままほかの問題にも簡単に流用できるという性質の技法ではない．個々の問題それぞれについて，どのような途中状態を記録すると効率よく解けるか，フロンティアが小さいままで処理を進められるようなグラフの形をしているか，良い変数順序付けを見つけられるか，ということを考慮して，動的計画法としてのアルゴリズムを設計する必要がある．一方で，従来の Apply 演算でも効率よく解ける問題もあるので，苦労してフロンティア法で実装しても改善効果が薄いという結果に終わる場合もある．例えばクリーク列挙の ZDD 生成は Coudert の方法 [3] がエレガントで非常に性能が良いので，フロンティア法を工夫しても勝てないかもしれない．あるいは非常に疎な例題では共有がほとんど起こらないため，ZDD を使わずにオーソドックスな branch and bound や reverse search で解くほうが速いという場合もある．個々の問題や例題ごとに吟味が必要である．

現実の問題では制約条件が単純な論理演算や集合演算では表せない場合がしばしばある．例えば電力網の例題では，電圧降下がある一定の範囲になければならないとか，電流が送電線の定格を超えてはならないとか，三相交流のバランスを取らなければいけない等々，組合せ的にシンプルに表せない制約が多いため，処理途中で共有することが難しく，フロンティア法ですべての条件を一気に計算することは困難である．あるいは津波に備えた避難所の配置問題では，津波の高さや，避難者の移動速度のモデル，地震による避難路の損傷状況など，制約条件もさまざまに変化するため，その都度，フロンティア法を設計し直すことは現実的でない．このような問題に対する現実的な対処方法としては，まず比較的簡単で確実に必要となるトポロジカルな条件だけでフロンティア法を適用して，膨大な解候補を ZDD で全列挙索引化しておいて，一方で問題依存の複雑な制約条件については，局所性や極大性などを利用する別の方法で ZDD をいくつか作っておいて，先にフロンティア法で求めた ZDD との Apply 演算で解集合の絞り込みを行う，というアプローチが良いと思われる．

6. おわりに

フロンティア法を用いることで，ある一定規模の問題であれば，膨大な候補を ZDD により全列挙して索引化しておくことが現実的に可能になり，これによってすべての可能性を尽くして探索や診断を行うことができる．さらに ZDD は単なる検索だけでなくリッチな代数演算処理系をもつため，事前に想定できなかった制約条件を追加して候補を絞り込んだり，今日と昨日との差分だけをまとめて取り出したり，といった柔軟な操作ができることも大きな特長である．

最適解を 1 つ求めるだけであれば，解を全列挙しなくても，それよりはるかに短い時間で計算できる場合がしばしばある．しかし，制約条件の性質によっては，結局，全列挙する以外に真の最適解を見つけられない問題も存在する．さらに現実の問題では，最適性の評価尺度や制約条件が刻々と変化する場合も多いため，ある条件を満たす解を列挙索引化する技術も有用性が高い．筆者らは「最適化と列挙は車の両輪」と言っても過言ではないと考えている．

大規模なシステムの解析には，従来，確率的な手法が多く用いられてきた．確率的手法では，計算能力の制約から，個々の事象の独立性を仮定することが多いが，現実の問題で特に非常に小さな確率を扱う場合には，独立性について細心の注意をすべきであることは，昨年の大震災による原発事故災害で明らかになった教訓の 1 つである．いったん，確率を計算して数値（例えば 10^{-6} など）に直してしまうと，その数値だけが社会で独り歩きしてしまい，それを求めたときの前提条件が忘れられやすい傾向がある．それに対して，列挙の場合は独立性を仮定しなくても数え上げることができ，非常に小さい確率の事象であっても「全部で何通りあって，例えばこういうときに発生」と示すことができる．そのような社会的な側面からも，全列挙索引化の技術は今後さらに重要になると考えている．

謝辞　本研究は，ERATO プロジェクトの各メンバや共同・連携研究者と協力して進めているものであり，

関係各位に感謝いたします．また Tutte 多項式計算についてご助言いただいた東京大学 今井浩先生に感謝いたします．

参考文献

[1] S. B. Akers, Binary decision diagrams, *IEEE Transactions on Computers*, Vol. C-27, No. 6, pp. 509–516, 1978.

[2] R. E. Bryant, Graph-based algorithms for Boolean function manipulation, *IEEE Transactions on Computers*, Vol. C-35, No. 8, pp. 677–691, 1986.

[3] O. Coudert, Solving graph optimization problems with ZBDDs, In *Proc. of ACM/IEEE European Design and Test Conference* (*ED&TC '97*), pp. 224–228, 1997.

[4] D. E. Knuth, *The Art of Computer Programming: Bitwise Tricks & Techniques; Binary Decision Diagrams*, Vol. 4, fascicle 1, Addison-Wesley, 2009.

[5] S. Minato, Zero-suppressed BDDs for set manipulation in combinatorial problems. In *Proc. of 30th ACM/IEEE Design Automation Conference (DAC'93)*, pp. 272–277, 1993.

[6] K. Sekine, H. Imai, and S. Tani, Computing the Tutte polynomial of a graph of moderate size. In *Proc. of the 6th International Symposium on Algorithms and Computation (ISAAC'95), LNCS-1004*, pp. 224–233, 1995.

[7] D. J. A. Welsh, Complexity, knots, colourings and counting. *London Mathematical Society Lecture Note Series*, Vol. 186, pp. 372–390, 1993.

[8] R. Yoshinaka, J. Kawahara, S. Denzumi, H. Arimura, and S. Minato, Counterexamples to the long-standing conjecture on the complexity of BDD binary operations. *Information Processing Letters*, Vol. 112, No. 16, pp. 636–640, 2012.

[9] 井上武 他，フロンティア法による電力網構成制御．オペレーションズ・リサーチ，Vol. 57, No. 11, pp. 610–615, 2012.

[10] 今井浩，ネットワーク信頼度計算の周辺—組合せ数え上げの新展開，離散構造とアルゴリズム V，第 5 巻，第 1 章，pp. 1–50，近代科学社，1998.

[11] 藤田昌宏，佐藤政生，特集　BDD（二分決定グラフ），情報処理学会誌，Vol. 34, No. 5, pp. 584–630, 1993.

[12] 川原純，湊真一，グラフ列挙索引化技法の種々の問題への適用，オペレーションズ・リサーチ，Vol. 57, No. 11, pp. 604–609, 2012.