

ZDD の区間メモ化探索技法による コスト制約組合せ問題の高速な解列挙

湊 真一^{1,a)} 番原 睦則^{2,b)} 堀山 貴史^{3,c)} 川原 純^{1,d)} 瀧川 一学^{4,5,e)} 山口 勇太郎^{6,f)}

概要：線形のコスト関数を持つ組合せ最適化問題は、最も基本的な数理モデルの 1 つであり、現実の場面でしばしば現れる。本研究では、コスト総和が所定の閾値以下となるような全ての解を、高速かつ厳密に列挙する手法を提案する。本手法は、ZDD (Zero-suppressed Binary Decision Diagram) で与えられた実行可能解の集合に対して、枝刈り可能なコスト値区間を管理する新しいメモ化の技法を用いてバックトラック探索を行い、高速にコスト制約解を抽出する。既存の動的計画法では、コスト値の総和（有効桁数）が大きいと急激に計算量が増大するが、本手法ではコスト値が非常に大きくても ZDD の圧縮率が高ければ効率良く実行できる。実験の結果、ハミルトンパス問題の現実的な例題において、数十億通りにおよぶコスト制約解をわずか数秒で厳密に全列挙することが可能であった。本手法は分枝限定法による最適化と動的計画法による列挙を統合したアルゴリズムとみなすことができる。正確なランキング計算や、近似率を保証した準最適解のランダムサンプリングなど、様々な応用が期待される。

Fast Enumeration of All Solutions for Cost-Bounded Combinatorial Problems Using Interval-Memoized Backtracking on ZDDs

1. はじめに

多数のアイテムの中から制約を満たすアイテム組合せを探索する問題は、組合せ問題と総称され、実社会の様々な場面で出現する。その中でも線形のコスト関数が与えられ、各アイテムのコスト総和が最小（または最大）になるような解を探索するタイプの組合せ最適化問題は、最短路問題や最小全域木問題、巡回セールスマン問題、ナップザック問題など多くのバリエーションがあり、最も基本的な数理モデルの 1 つとして古くから研究されている。様々な組合

せ最適化問題に対応する汎用的で強力なソフトウェアツールも開発されており、整数線形計画法 (ILP) をベースとするもの [1], [2]、SAT ソルバをベースとするもの [19], [20]、BDD 処理系を用いるもの [18] などがある。

しかし現実の工学的応用では、利用者が本当に求めたい制約条件やコスト関数を正確にモデル化できていないことがしばしばあり、そのような場合には得られた最適解が必ずしも最適問題解決とはならない。そこで、近年の計算機性能の大幅な向上に伴い、最適解を 1 つ求めるだけでなく、実行可能解を全て列挙しようとする、いわゆる列挙アルゴリズムの研究 [21] も、理論と応用の両面から注目され始めている。その中でも、ZDD (Zero-suppressed Binary Decision Diagram) を用いた高速なグラフ列挙アルゴリズム [6], [11], [15] が開発され、解集合を圧縮・索引化することにより、膨大な個数の解集合をまとめて効率よく列挙して提示することが可能となってきたことも、列挙手法の有用性を高めている大きな技術的要因となっている。

列挙アルゴリズムを用いて求めたい解集合を絞り込んでいくアプローチにおいては、提示する解の個数が多すぎると、利用者が解集合の内容を把握することが困難になる。解の個数を制御する手段としては、コスト関数は利用者にとって理解しやすい指標の 1 つであり、コスト値で絞り込むことができれば利便性が高いと考えられる。本研究では、コスト総和が所定の閾値以下（または以上）であることを制約条件として付加した問題を「コスト制約組合せ問題」として定式化し、膨大な個数となりうる全てのコスト制約解を、高速かつ厳密に列挙する手法を提案する。本研究で提案するアルゴリズムは、ZDD で表現された全

¹ 京都大学大学院情報学研究科
606-8501 京都市左京区吉田本町
² 名古屋大学大学院情報学研究科
464-8603 名古屋市千種区不老町
³ 北海道大学大学院情報科学研究院
060-0814 札幌市北区北 14 条西 9 丁目
⁴ 理化学研究所革新知能統合研究センター (AIP)
103-0027 東京都中央区日本橋 1-4-1 三井ビル 15 階
⁵ 北海道大学化学反応創成研究拠点 (ICReDD)
001-0021 札幌市北区北 21 条西 10 丁目
⁶ 大阪大学大学院情報科学研究科
565-0871 大阪府吹田市山田丘 1-5
a) minato@i.kyoto-u.ac.jp
b) banbara@nagoya-u.jp
c) horiyama@ist.hokudai.ac.jp
d) jkawahara@i.kyoto-u.ac.jp
e) ichigaku.takigawa@riken.jp
f) yutaro.yamaguchi@ist.osaka-u.ac.jp

ての実行可能解の集合に対して網羅的にバックトラック探索を行い、コスト制約を満たす実行可能解のみを抽出した ZDD を出力する。我々の研究グループでは、以前よりこの問題に取り組んでおり、入力 ZDD サイズと出力の解の個数の和で計算時間が抑えられるアルゴリズム [23] を過去に提案している。この既発表手法では、厳しいコスト閾値を与えれば解の個数が少なくなるので高速に動作するが、コスト閾値を緩めると解の個数が急激に増大し、現実的な時間では終わらなくなるという問題点があった。本稿で新しく提案するアルゴリズムは、区間メモ化と呼ぶ高速化技法を用いることにより、ZDD の節点が共有される部分の探索を大幅に枝刈りできる。そのため、解の個数が膨大であっても出力の ZDD の圧縮率が高ければ、格段に高速に動作する。

本研究で扱うような線形不等式制約の組合せ問題では、コスト部分和で分類したテーブルを用いる古典的な動的計画法による解法が知られており、その計算量は擬多項式時間（コスト値の総和に対して多項式）で抑えられる。しかし現実の応用では、例えば都市間の距離、国家規模の面積、人口、平均所得など、大きな桁数のコスト値を扱いたい場面がしばしばあり、古典的方法ではテーブルが大きくなり過ぎる。それに対して提案アルゴリズムは、コスト値の大きさには直接依存せず、入力と出力の ZDD サイズにのみ計算量が依存するため、コスト値が非常に大きくても ZDD の圧縮率が高ければ効率良く実行できる。実験の結果、ハミルトンパス問題の現実的な例題において、数十億通りにおよぶコスト制約解をわずか数秒で厳密に全列挙することが可能であった。本手法は分枝限定法による最適化と動的計画法による列挙を統合したアルゴリズムとみなすことができる。正確なランキング計算や、近似率を保証した準最適解のランダムサンプリングなど、様々な応用が期待される。

以下の本文では、2 章で問題設定を行い、3 章で既存手法による解法とその問題点を述べる。続いて 4 章で提案アルゴリズムとその計算量を示す。5 章で実用的な規模の例題に対する実験結果を示し、最後に 6 章で今後の課題等を述べる。

2. 準備

2.1 コスト制約組合せ問題

本稿では、まず n 個のアイテム集合 $I = \{1, 2, \dots, n\}$ を定義し、 I の任意の部分集合 $X \subseteq I$ を「アイテム組合せ」と呼ぶ。制約条件を表す集合関数 $f: 2^I \rightarrow \{0, 1\}$ が与えられたとき、 $f(X) = 1$ を満たすような実行可能解の集合は $S_f = \{X \subseteq I \mid f(X) = 1\}$ と表せる。さらに各アイテム i ($1 \leq i \leq n$) について整数値のコスト $c_i \in \mathbf{Z}$ が定義されているものとし、アイテム組合せ X のコストを $\text{Cost}(X) = \sum_{i \in X} c_i$ と定義する。組合せ最適化問題とは

$\text{Cost}(X^*)$ が最小値（または最大値）となるような実行可能解 $X^* \in S_f$ を探索する問題として定式化できる^{*1}。

本研究では、上記の最適化問題から派生する列挙問題について考察する。コスト関数に閾値 $b \in \mathbf{Z}$ を与えることで、 $\text{Cost}(X) \leq b$ を満たすような解 $X \in S_f$ を求める組合せ問題を考えることができる。本研究ではこのような問題を「コスト制約組合せ問題」と呼び、その解を高速かつ厳密に全列挙することを考える。

2.2 ZDD と組合せ最適化問題

ZDD (Zero-suppressed Binary Decision Diagram; ゼロサプレス型二分決定グラフ) [17] は、図 1(a) に示すよう

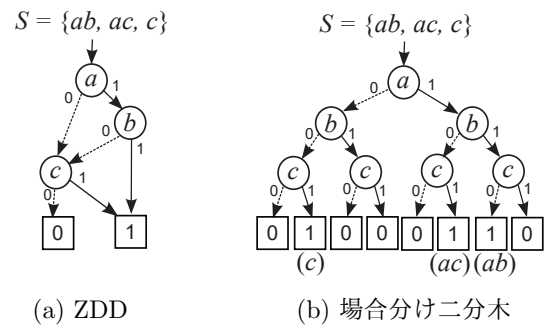


図 1 ZDD と場合分け二分木

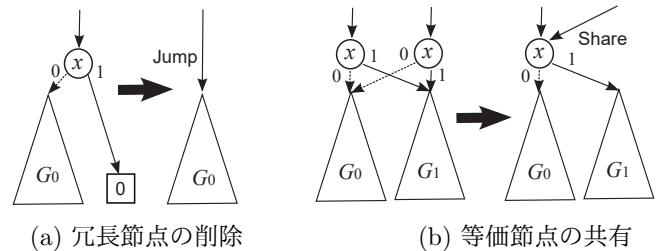


図 2 ZDD の圧縮規則

な非巡回有向グラフによる組合せ集合の表現方法である。これは、図 1(b) の場合分け二分木を圧縮することにより得られる。この場合分け二分木では、各分岐節点の 1-枝と 0-枝は、その節点にラベル付けされたアイテムを選ぶかどうかの場合分けを表し、葉の値 (1/0) はその葉に対応する組合せが集合に属するかどうかを示している。場合分け二分木から ZDD を得るための圧縮を行う際には、まず場合分けするアイテムの順序を固定し、以下の 2 つの圧縮規則を可能な限り適用する。

- 1-枝が 0 の葉を直接指しているような節点を削除し、0-枝の行き先に直結させる。(図 2(a))
- 等価な節点（アイテムが同じで、0-枝、1-枝の行き先がそれぞれ同じ）を共有する。(図 2(b))

これにより「既約」な形が得られ、組合せ集合をコンパクトかつ一意に表せることが知られている。ZDD によるデータ圧縮率は、組合せ集合の性質にも依存するが、例題によっては数十倍～数百倍以上もの圧縮率が得られる場合がある。ZDD を組織的に構築する方法として、2 つの ZDD に対する集合演算（共通集合、和集合、差集合など）を実行するアルゴリズムが知られている。これはハッシュテーブルに途中の ZDD 節点を記録しながら深さ優先順にたどっていく再帰的アルゴリズム [4] に基づく。理論的な計算時間は 2 つの ZDD サイズの積となるが、実験的には入出力 ZDD サイズの総和にほぼ比例する。

ZDD を用いれば、組合せ最適化問題を簡潔な手続きで解くことができる。まず離散的な制約 f を集合演算の式で記述し、その式の構造にしたがって ZDD の集合演算アルゴリズムを順に適用していけば、実行可能解の集合 S_f を表す ZDD を構築することができる。このとき、 S_f の ZDD の根節点から 1 の葉に至るパス（以後、1-パスと呼ぶ）が見つかるならば、それぞれの 1-パスが組合せ問題の 1 つの実行可能解に対応している。そこで、 S_f の ZDD を深さ優先順に探索して、コストを最小（または最大）にする 1-パスを見つけ出せば、それが最適解となる。もしも愚直に深さ優先順で探索すると同じ節点を何度もたどることになり、 n に関して指数的な時間がかかってしまうが、各節点においてその節点を始点とする 1-パスの最小（または最大）コストを記録しておけば、一度通った節点を重複して探索

^{*1} 本稿では、整数値重みの線形コスト関数に限定して議論を行う。

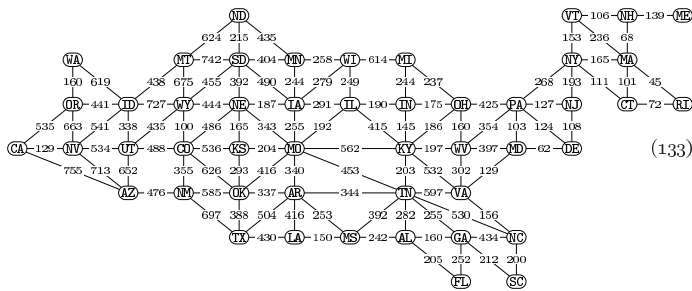


図 3 Knuth の全米 48 州の隣接グラフ [15]

する必要がなくなるので、ZDD の節点数に比例する時間 $O(|S_f|)$ で必ず最適解を見つけられる。したがって、実行可能解の ZDD を構築できれば、最適解を 1 つ求めることは容易である。それでは、最適解 1 つだけではなく、ある閾値よりも良いコストの解を効率良く全列挙することができるか、というのが本研究で取り組む課題である。

3. コスト制約解の全列挙

3.1 典型的例題

Knuth は、著書「The Art of Computer Programming」Vol. 4-1 [15] において、図 3 に示すような米国主要 48 州の隣接グラフを示し、全ての州を一度だけ通って大陸を横断するハミルトンパスを求める問題について論じている。このグラフは 48 頂点 105 辺からなり、各辺には州都の間の距離（マイル数）がコスト値として設定されている。Knuth は、この問題の全ての実行可能解の集合を表す ZDD を超高速に構築する「simpath アルゴリズム」を提案している。このアルゴリズムは、与えられたグラフを「フロンティア」と呼ばれる断面でスキャンしていく一種の動的計画法で、ZDD を上位から下位に向かって幅優先順にトップダウンに構築していくものである。この構築法は、ハミルトンパスに限らず、単純パス、サイクル、連結部分グラフ、木、森、カットなど、様々なグラフの制約を扱うことができる。筆者らは、このような ZDD 構築方法を総称して「フロンティア法」[22]と呼んでおり、電力網の最適化 [12]、疫学ホットスポットの検出 [13]、選挙区割りの網羅的解析 [7], [14] など、様々な応用研究がある。

Knuth のアルゴリズムを用いれば、米国西海岸ワシントン州 (WA) から東海岸メイン州 (ME) に至る 6,876,928 通りの全てのハミルトンパスを列挙する ZDD を構築できる。その ZDD の節点数はわずか 3,616 であり、構築に要する時間はわずか 0.02 秒である。Knuth の方法は驚くほど高速に全ての実行可能解を列挙できるが、残念ながらコスト値で制約した上位解の集合（例えば最短ハミルトンパスの 20% 増以内の全ての解）を列挙する方法は示されていない。現実の応用では、ある程度のコスト増は許容し、他のソフトな制約（例えば冬季に積雪の多い峠道は避けたい等）を考慮して絞り込みたい、という場面は多いと思われる。また、統計分野の応用では、観測した結果の p 値を計算するために、解のランキング（自分よりコスト値が上位の解が全部で何個あるか）を正確に求めたいという場面も考えられる。このように「コスト制約組合せ問題の解列挙」という問題設定は、実応用での重要性が高いと考えられる。

3.2 既存の最適化手法を用いる Top-k 探索法

既存の最適化アルゴリズムをベースとして、上位 k 個の解を列挙する top- k search または k -best search と呼ばれる手法の研究が以前より行われている。例えば、巡回セールスマン問題（最短ハミルトンサイクル）の最適解を 1 個だ

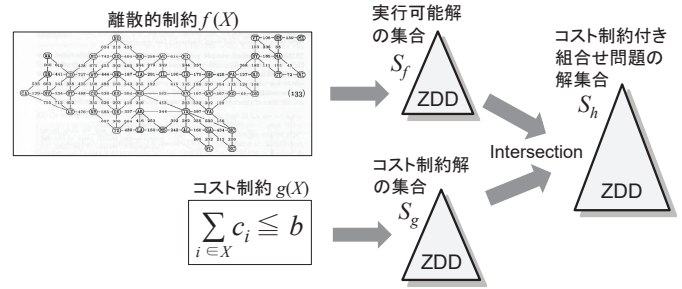


図 4 ZDD 共通集合演算に基づく既存手法

表 1 S_g を表す ZDD の節点数（一様ランダムなコスト値を設定）

n	コスト = 1	乱数値 [1, 100]	乱数値 [1, 10000]
10	30	53	52
20	110	1,645	2,128
30	240	7,225	54,302
40	420	16,124	546,035
50	650	27,589	1,501,267
60	930	39,931	2,729,644
70	1,260	55,387	4,193,333
80	1,640	76,355	6,033,872
90	2,070	102,831	8,394,357
100	2,550	123,131	10,484,185
110	3,080	148,508	12,933,163
120	3,660	173,369	16,219,732

け求めるので良ければ、整数線形計画法により実用規模の例題を高速に解く古典的な方法 [16] が知られている。そこで、まずコスト最適解を 1 つ探索し、得られた最適解を否定するような制約を追加して、再度同じ問題を解くことを k 回繰り返して top- k の解を列挙する。このアプローチでは k が大きい場合には計算時間がかかり過ぎる。例えば、Knuth の米国横断の例題では、最短パスの 10% 増で 16,251 通り、20% 増では 941,214 通りの解が存在している。このように解が多い場合には、既存の繰り返し型 top- k アプローチは現実的ではない。ハミルトン条件がない単純な最短パス問題に限れば、最適化を繰り返し行わずに Dijkstra 法とヒープ構造を組合せた非常に高速な解法 [5] があるが、それでも k が 100 万の単位になると苦しくなる。

3.3 動的計画法による古典的手法

本研究で扱うような線形不等式制約の組合せ問題では、コスト部分で分類したテーブルを用いる古典的な動的計画法による解法が知られており、その計算量は擬多項式時間（コスト値の総和に対して多項式）で抑えられる。しかし現実の応用では、例えば都市間の距離、国家規模の面積、人口、平均所得など、大きな桁数のコスト値を扱いたい場面がしばしばあり、古典的方法ではテーブルが大きくなり過ぎる。例えば、Knuth の米国横断の例題では、コスト値の総和は 35,461 (マイル) となり、特に工夫しなければ動的計画法のテーブルのサイズは約 300 万セルに達する。これは最近の PC であれば何とか扱える規模であるが、そのテーブル上でハミルトンパスを探索しようとする大幅に速度が低下する。

3.4 ZDD 共通集合演算に基づく既存手法

解の個数が非常に多い場合に、ZDD でまとめて圧縮して列挙する方法は非常に有力である。図 4 は、ZDD の共通集合演算を用いて、コスト制約組合せ問題の解を列挙する既存手法を説明している。まず全ての実行可能解の集合 S_f を表す ZDD f を構築し、それとは別に、コスト制約の不等式を満たす全てのアイテム組合

せの集合 $S_g = \{X \subseteq I \mid \text{Cost}(X) \leq b\}$ を表す ZDD g を構築する。その後、2つの ZDD 同士の共通集合演算を実行すれば、コスト制約を満たす実行可能解の集合 $S_h = \{X \in S_f \mid \text{Cost}(X) \leq b\}$ を表す ZDD h が得られる。

この方法は一見すると有効そうに見えるが、重大な弱点がある。 S_g を表現する ZDD g は、古典的な動的計画法のテーブルと本質的には同じ内部構造となる。つまり、線形コスト関数の ZDD サイズが小さくなるのはコスト値の総和が小さな定数で抑えられる場合に限り、一般には指数的に増大する [9]*2。表 1 に我々の予備実験の結果を示す。 n アイテムの線形コスト関数において、各アイテムのコスト値を $[1, 100]$ または $[1, 10000]$ の範囲の一樣ランダムな値に設定し、その平均値の約半分の閾値 b を与えたときの S_g を表す ZDD 節点数を計測している。実験結果を見ると、コスト値の平均値が数千程度で、アイテム数が 50 を超えると、ZDD g の節点数は 100 万を超えて増大することが観察できる。このようなコスト値に対しては、もしも f や h の ZDD が小さかったとしても、 g のサイズが支配的となり計算時間が大幅に悪化する。

次章では、 S_g を表す ZDD g を陽に生成することなく、ZDD f から直接 ZDD h を生成する新しいアルゴリズムについて述べる。

4. ZDD のバックトラック探索とその高速化

4.1 基本探索アルゴリズム

本研究で提案するアルゴリズムは、アイテムの個数 n 、全ての実行可能解の集合 S_f を表す ZDD f 、および各アイテムのコスト値 c_i ($1 \leq i \leq n$) と閾値 b を入力とする。コスト値と閾値は整数値であれば 0 や負の値でも構わない。整数値の最大ビット幅はあらかじめ決めなくてもよい。アルゴリズムの出力は、 $S_h = \{X \in S_f \mid \text{Cost}(X) \leq b\}$ を表す ZDD h である。以下に基本アルゴリズム **BacktrackNaive** の擬似コードを記す。

```

1: ZDD BacktrackNaive(ZDD  $f$ , int  $b$ )
2: {
3:   if  $f = [0]$  return  $[0]$  // 0-terminal case
4:   if  $f = [1]$  then // 1-terminal case
5:     if  $b \geq 0$  return  $[1]$ 
6:     else return  $[0]$ 
7:   //  $f$  consists of  $(x, f_0, f_1)$ 
8:    $h_0 \leftarrow \text{BacktrackNaive}(f_0, b)$ 
9:    $h_1 \leftarrow \text{BacktrackNaive}(f_1, b - \text{cost}(x))$ 
10:   $h \leftarrow \text{ZDD}(x, h_0, h_1)$  // applying ZDD reduction
11:  return  $h$ 
12: }
```

このアルゴリズムは、再帰的なバックトラック探索により入力の ZDD f を深さ優先順にたどって行く。すなわち、与えられた問題 (f, b) を 2つの部分問題 (f_0, b) と $(f_1, b - \text{cost}(x))$ に分解し、再帰的に実行する。再帰呼び出しをくり返していくと最終的に ZDD の終端節点に到達する。もし 1-終端節点に到達してコスト閾値が負でなければ、そのときのアイテム組合せはコスト制約を満たす解として受理され、出力として 1-終端節点を返してバックトラックする。そうでなければコスト制約を満たさないで 0-終端節点が返される。 f が終端節点でない場合は 2つの部分問題の結果 h_0 と h_1 を子節点とする新しい ZDD 節点 h が生成され、出力値として返される。10 行目

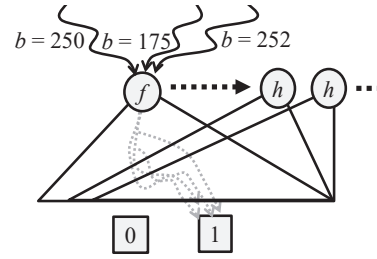


図 5 異なる閾値で同じ ZDD 節点を再訪した場合

の $\text{ZDD}(x, h_0, h_1)$ は、分岐節点を生成する内部手続きを表す。等価な節点がすでに存在していた場合には、新しい ZDD 節点を作らずに、既存の節点へのポインタを返すようになっている。

4.2 通常のメモ化技法とその問題点

上記の手続き **BacktrackNaive** を再帰的に呼び出すことで出力結果の ZDD を構築できるが、これだけでは、元の ZDD で共有されているサブグラフを何度も重複して訪問してしまい、常に指数的な計算時間を要することになる。ZDD の演算アルゴリズムの常套手段として、同じ節点を複数回訪問した場合には、最初に訪問したときの演算結果を憶えておいて、その結果を再利用することにより、重複する処理を回避するメモ化技法（演算キャッシュとも呼ばれる）が使われる。

ところが、これはコスト制約を持たない ZDD の演算の場合には有効であるが、コスト制約付きの場合は、図 5 に示す通り、その節点までにたどってきた途中までのアイテムのコスト部分和が、前回訪問時と異なる場合があるので、同じ節点を訪問していたとしても閾値 b が異なり、演算結果が前回と同じになるとは限らない。したがって、演算結果を再利用するためには、 (f, b) のペアが共に前回と同じであるかをチェックする必要がある。

以下に、通常のメモ化技法を用いて演算結果を再利用するアルゴリズム **BacktrackMemo** を示す。

```

1: ZDD BacktrackMemo(ZDD  $f$ , int  $b$ )
2: {
3:   if  $f = [0]$  return  $[0]$  // 0-terminal case
4:   if  $f = [1]$  then // 1-terminal case
5:     if  $b \geq 0$  return  $[1]$ 
6:     else return  $[0]$ 
7:    $h \leftarrow \text{memo}[f, b]$ ; if  $h$  exists return  $h$ 
8:   //  $f$  consists of  $(x, f_0, f_1)$ 
9:    $h_0 \leftarrow \text{BacktrackMemo}(f_0, b)$ 
10:   $h_1 \leftarrow \text{BacktrackMemo}(f_1, b - \text{cost}(x))$ 
11:   $h \leftarrow \text{ZDD}(x, h_0, h_1)$  // applying ZDD reduction
12:   $\text{memo}[f, b] \leftarrow h$ 
13:  return  $h$ 
14: }
```

この擬似コードでは、7 行目と 12 行目の $\text{memo}[f, b]$ が、冗長な探索を避けるために追加されたメモ化の機能である。このアルゴリズムは、素朴なバックトラックよりは高速になるが、探索中に出現する全ての部分和の組合せがメモ化されるので、古典的な動的計画法のテーブルで部分和を分類するのと本質的には同じ処理を行うことになる。したがって、もしもコスト値の有効桁数が大きくて、偏りなく分布していた場合には、部分和の値が完全に一致する確率は非常に低くなるため、あまり効率が向上しないという問題点がある。

*2 この論文では BDD の節点数について論じているが ZDD でも本質的な性質は変わらない。

4.3 分枝限定法による高速化（既発表手法）

我々の研究グループでは、以前よりこの問題に取り組んでおり、分枝限定法による高速化アルゴリズム [23] を過去に提案している。この既発表手法は、深さ優先探索で訪問済みの部分については、実行可能解のコストの最小値と最大値が正確に分かるので、それを各節점에記録しておき、同じ節点を再訪した場合に、そこまでの部分和による閾値が残りの最小最大値の範囲を超える場合には、出力は解なし（空集合）、または f がそのまま解集合となる、というアイデアに基づく。以下に擬似コード **BacktrackMinMax** を示す。

```

1: ZDD BacktrackMinMax(ZDD  $f$ , int  $b$ )
2: {
3:   if  $f = [0]$  return  $[0]$  // 0-terminal case
4:   if  $f = [1]$  then // 1-terminal case
5:     if  $b \geq 0$  return  $[1]$ 
6:   else return  $[0]$ 
7:   if  $f$ .min exists &  $f$ .min >  $b$  return  $[0]$ 
8:   if  $f$ .max exists &  $f$ .max  $\leq b$  return  $f$ 
9:   //  $f$  consists of  $(x, f_0, f_1)$ 
10:   $h_0 \leftarrow \text{BacktrackMinMax}(f_0, b)$ 
11:   $h_1 \leftarrow \text{BacktrackMinMax}(f_1, b - \text{cost}(x))$ 
12:   $h \leftarrow \text{ZDD}(x, h_0, h_1)$  // applying ZDD reduction
13:   $f$ .min  $\leftarrow \min(f_0$ .min,  $f_1$ .min + cost( $x$ ))
14:   $f$ .max  $\leftarrow \max(f_0$ .max,  $f_1$ .max + cost( $x$ ))
15:  return  $h$ 
16: }
```

上記 13,14 行目で ZDD f の実行可能解のコストの最小値と最大値を求め記録している。そして 7,8 行目で過去の最小最大値をチェックし、範囲外と分かれば直ちに結果を返しバックトラックする。ただしこの枝刈りがヒットしなければ、素朴なバックトラック法と同様に、同じ節点を何度でも訪問するアルゴリズムとなっている。この既発表手法では、再帰呼び出し回数が、コスト制約を満たす解の個数と再帰の深さ（高々 n ）の積で抑えられることが証明されており、解の個数が少なければ非常に高速に動作する。しかし、緩いコスト閾値を与えると解の個数が急激に増大し、現実的な時間では終わらなくなるという問題点があった。それに対して本稿では、解の個数が膨大であっても、ZDD の圧縮率が高ければ効率良く動作するアルゴリズムを提案する。

4.4 区間メモ化技法による高速化（本提案手法）

図 5 で示した通り、深さ優先探索で同じ ZDD 節点 f を複数回訪問する場合、初回のコスト閾値 b と異なる閾値 b' で訪問した場合は、計算結果の ZDD h は同じになるとは限らない。しかし直感的には、もしも b と b' が非常に近い値であれば、高い確率で計算結果が一致することが期待される。より正確に言えば、 b と b' の間のコスト区間に解が存在しなければ、計算結果の ZDD は必ず一致するはずである。これが本提案手法のキーとなるアイデアである。ここで 2 つの重要な関数を定義する。

- $\text{accept_worst}(f, b)$:
 f が表す解集合の中で b を超えない最大のコスト値
- $\text{reject_best}(f, b)$:
 f が表す解集合の中で b を超える最小のコスト値

この 2 つの関数を使うと、閾値 b と b' で計算結果が一致するための必要十分条件は以下の式で表せる。

$$\text{accept_worst}(f, b) \leq b' < \text{reject_best}(f, b)$$

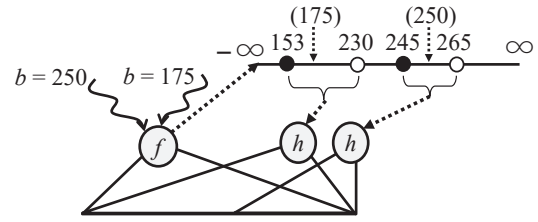


図 6 区間メモ化技法のアイデア

本研究で提案する区間メモ化技法のアイデアを図 6 に示す。ZDD の各節点 f に対して、数値の大小関係を保持できる数直線型のメモ（区間メモと呼ぶ）を用意する。線上の黒丸は accept_worst 、白丸は reject_best のポイントを表す。この例では、まず最初に閾値 $b = 250$ で節点 f を訪問したとすると、その計算結果 h が区間 $[245, 265]$ に記録される。次に $b = 252$ で節点 f を再訪したときには、区間メモを参照することで冗長な再帰呼び出しを回避する。一方で $b = 175$ で f を再訪したときには、区間メモの範囲外なので探索を継続し、異なる計算結果の ZDD h を生成し、区間 $[153, 230]$ に記録することになる。

上記のような大小関係を保持する区間メモは、平衡二分探索木（self-balancing binary search tree）を用いれば効率良く実装でき、例えば C++ 標準ライブラリの `std::map` が利用できる。1 回の読み書きの計算量は、記録件数を m として $O(\log m)$ で抑えられる。これは定数ではないが、 m が極端に大きくならない限りは十分許容できるオーバヘッドである。

もう一つの重要な論点は、 accept_worst と reject_best の区間の値をどのようにして得るかということであるが、ZDD のバックトラック探索の手順に少し手を加えるだけで簡単に計算することができる。以下に、提案アルゴリズム **BacktrackIntMemo** を示す。

```

1: BacktrackIntMemo(ZDD  $f$ , int  $b$ )
2: // returns a triple (ZDD  $h$ , int  $\text{accept\_worst}$ ,  $\text{reject\_best}$ )
3: {
4:   if  $f = [0]$  return  $([0], -\infty, \infty)$ 
5:   if  $f = [1]$  then
6:     if  $b \geq 0$  return  $([1], 0, 0)$ 
7:   else return  $([0], -\infty, 0)$ 
8:   ( $h$ ,  $aw$ ,  $rb$ )  $\leftarrow \text{memo}[f, b]$ ; if exists return  $(h, aw, rb)$ 
9:   //  $f$  consists of  $(x, f_0, f_1)$ 
10:  ( $h_0$ ,  $aw_0$ ,  $rb_0$ )  $\leftarrow \text{BacktrackIntMemo}(f_0, b)$ 
11:  ( $h_1$ ,  $aw_1$ ,  $rb_1$ )  $\leftarrow \text{BacktrackIntMemo}(f_1, b - \text{cost}(x))$ 
12:   $h \leftarrow \text{ZDD}(x, h_0, h_1)$  // applying ZDD reduction
13:   $aw \leftarrow \max(aw_0, aw_1 + \text{cost}(x))$ 
14:   $rb \leftarrow \min(rb_0, rb_1 + \text{cost}(x))$ 
15:   $\text{memo}[f, [aw, rb]] \leftarrow h$ 
16:  return  $(h, aw, rb)$ 
17: }
```

このアルゴリズムは、出力として ZDD h を返すだけでなく、2 つの整数値 aw (accept_worst) と rb (reject_best) を追加した 3 つ組を返すよう拡張されている。終端節点の場合の処理は 4,6,7 行目に書かれている。もしも f が 1-終端節点で $b \geq 0$ の場合は、 $b = 0$ のときが同じ結果を返す最小の閾値となるので $aw \leftarrow 0$ とする。同様に $b < 0$ の場合は $b = 0$ のときが計算結果が一致しなくなる最小の閾値となるので $rb \leftarrow 0$ とする。一方、13,14 行目には f が終端節点でない場合の処理が書かれている。ここでは 2 つの部分問題の再帰呼び出しの結果として、 $[aw_0, rb_0]$ と $[aw_1, rb_1]$ の 2 つの区間が既に得られているので、両者の最大値と最小値を選ぶことで、定数ステップのオーバヘッドで $[aw, rb]$

を求めることができ、これを用いて区間メモに計算結果 h を登録することができる。

本提案アルゴリズムはいくつかの興味深い性質を持つ。もしも閾値 $b = -\infty$ として呼び出すと、常に空集合を返し、そのときの *reject_best* は入力された解集合 f の最小コスト値となる。同様に $b = \infty$ とすれば、常に f そのものを返し、そのときの *accept_worst* は f の最大コスト値を表す。上記の2つの場合のアルゴリズムの動作は、分枝限定法により最大・最小値を求める最適化アルゴリズムと本質的に一致し、その計算時間は入力 ZDD サイズに対して線形 $O(|f|)$ で抑えられる。一方で、適当な閾値 b を与えて本アルゴリズムを実行した場合には、古典的な動的計画法で構築するような部分和のテーブルを、より無駄なく共有して構築する処理となっている。つまり、本提案手法は、分枝限定法と動的計画法という2つの古典的技法を統合した形になっており、最適化と列挙を同じプログラムで効率良く実行できるという優れた性質を持つ。

区間メモ化の技法は他にも利点がある。同じ実行可能解 f に対して複数の異なるコスト閾値を与えて何度も解列挙を実行し、解の分布状況を解析するという利用法が考えられるが、そのような場合には計算結果の複数の ZDD 同士でも互いに共有部分が多く生じることになる。区間メモ化技法により、共有可能な部分グラフに関する結果は保存されているので、2回目以降の実行では直ちに結果を得ることが可能となる。

コスト制約組合せ問題の解集合の ZDD を構築すれば、ZDD 処理系が提供する種々の集合演算を用いて多様な解析処理が可能となる。例えば、2つのコスト閾値 lb と ub についてそれぞれ解集合の ZDD h_{lb}, h_{ub} を生成すれば、ZDD 同士の差集合演算 ($h_{ub} \setminus h_{lb}$) によって、コスト区間 $(lb, ub]$ の範囲の全ての解集合が得られる。

4.5 計算量解析

提案アルゴリズムの計算量について述べる。直感的には計算時間は入力と出力の ZDD サイズに依存する。

定理 4.1. アイテムのべき集合全体 2^I を表す ZDD f が入力として与えられた場合、**BacktrackIntMemo** の計算時間は、入力と出力の ZDD サイズを $|f|$ と $|h|$ とすると $O(|f| + |h| \log |h|)$ で抑えられる。

証明. ZDD f の同じ節点を再訪したときに、区間メモ化技法により出力 ZDD が前回と同じになるかどうかを正しく判定できる。したがって、出力 ZDD h の同じ節点を重複してたどることはないで、再帰呼び出し回数は $O(|h|)$ で抑えられる。区間メモの1回の読み書きは、記録件数 m に対して $O(\log m)$ ステップ必要とするが、メモに記録されるのは h の内部節点なので m は $|h|$ で抑えられる。一方で、 h が空集合になる場合や入力 f よりも著しく小さい場合でも、 f の各節点を1回ずつたどるので、少なくとも $O(|f|)$ ステップは必要である。以上をまとめると計算時間は $O(|f| + |h| \log |h|)$ となる。□

上記の定理では f がべき集合全体であるとき、すなわちコスト制約のみが与えられるような最も基本的ケースのみについて論じている。残念ながらこの定理は一般の f に対しては反例が存在する。 f の同じ節点を訪問したときに出力結果が同じ節点になる場合は区間メモで必ず判定できるので h を重複してたどることはない、という観察は正しい。しかし f の異なる節点に対してたまたま同じ出力結果が発生する場合があります、区間メモではそれを検出することができず、 h の同じ節点を重複してたどる可能性がある。そのような反例を人工的に作ることは可能であるが、現実の例題において自然発生することは稀であると考えられ、多く

の実用的な応用では経験的に $O(|f| + |h| \log |h|)$ で抑えられると予想される。

5. 実験結果と考察

提案アルゴリズムを実装して現実的な規模の例題に対する実験を行った。プログラムは Graphillion[10] の内部で使用しているものと同じ SAPPOROBDD パッケージを用いて、C および C++ で記述した。計算機は汎用 Linux PC (Intel Xeon W-2225, 4 コア 4.1GHz, 主記憶 256GB) である。Ubuntu 20.04, g++ 9.3.0 を使用した。

まず3章で述べた Knuth の米国 48 州の例題 (ME から WA までのハミルトンパスの列挙) の実験結果を示す。この例題は、頂点数 48、辺数 105 で、各辺にコストが付与されていて、アイテム数 $n = 105$ である。Knuth のアルゴリズムを用いてコスト制約のない実行可能解の ZDD f を構築すると、ZDD 節点数は 3,616、解の総数は 6,876,928 通り、最小コスト 11,698、最大コスト 18,040 であった。この実行可能解に様々なコスト閾値を与えて実行した結果を表 2 に示す。計算時間は実行可能解の ZDD f を構築する時間、コスト制約解の ZDD h を構築する時間、および h の節点数を数えるのにかかった時間を全て含んでいる。閾値として最大コスト値を与えたときには、入力と出力の ZDD f と h は一致する。この例題では、入力 ZDD f を構築するための計算時間は、コスト制約解に絞り込む時間に比べて極めて短いことが分かる。なお、この実験では閾値を変えて実行する度に区間メモを初期化しているが、初期化せずに続けて実行した場合はさらに高速となる。

表の右側半分には、区間メモ化技法を用いた本提案手法の計算時間と再帰呼び出し回数、通常の方法だけを用いた場合の再帰呼び出し回数、そして分枝限定法だけを用いた既発表手法 [23] の再帰呼び出し回数を示す。実験結果を見ると、本提案手法により、100 万通りを超える膨大なコスト制約解の集合を 0.1 秒程度で全列挙することに成功している。計算時間については、前章で述べた通り、実験的には $O(|f| + |h| \log |h|)$ に概ねしたがっていることが観察できる。通常の方法だけを用いた場合と呼出し回数を比べると、計算結果の ZDD の圧縮率が高い場合に特に効果が顕著であり、提案手法の方が 10~600 倍程度高速である。一方、分枝限定法ではコスト制約解が非常に少ない場合と、ほとんどの解がコスト制約を満たす場合には本手法と大差ないが、中間領域では効率が悪く、本手法の方が最大 30 倍程度高速であることが分かる。

次に、別の性質を持つグラフとして、ランダムなコストを持つ $n \times n$ 格子グラフの例題に適用した実験結果を示す。このグラフは $(n+1)^2$ 個の頂点と $2n(n+1)$ 本の辺からなり、各辺は [1000, 1999] の範囲のランダムなコスト値を与えられている。このグラフに対して対角頂点を結ぶハミルトンパスを全列挙する ZDD をまず構築し、その中からコストが閾値よりも小さい解だけを本提案手法を用いて抽出する実験を行った。表 3 に 8×8 と 10×10 の例題に対する実験結果を示す。各項目の見方は表 2 と同じである。 10×10 の例題では、既存手法の実験は時間がかかり過ぎるため省略し、本提案手法だけの結果を示した。これらの実験結果より、本提案手法では 10 億通りを超えるコスト制約解の集合を数秒程度で厳密に全列挙できることが分かる。最も大規模な例では 1 兆通りのコスト上位解を約 1 時間で抽出できている。既存手法との差は問題が大規模になるほど顕著であり、 8×8 の例題で、通常の方法と比べて数千倍、分枝限定法と比べて数百倍の違いが現れるケースがある。

最後に本手法と既存の列挙手法の性能を比較するため、近年注目されている ASP (解集合プログラミング) システム clingo [8] を用いて、同じ例題で比較実験を行った。

表 2 Knuth の米国 48 州の例題への適用結果

米国 48 州の隣接グラフ ($V: 48, E: 105$)

閾値 (近似率)	解の個数	ZDD 節点数	提案手法		通常のメモ化 呼出し回数	分枝限定法 [23] 呼出し回数
			時間 (秒)	呼出し回数		
11,698 (1.00)	1	47	0.028	7,329	197,891	7,309
11,814 (1.01)	8	99	0.029	7,441	197,891	7,397
11,931 (1.02)	28	152	0.028	7,555	197,891	7,739
12,282 (1.05)	388	1,001	0.029	9,489	229,257	13,743
12,867 (1.10)	16,180	9,679	0.035	28,127	340,277	180,101
14,037 (1.20)	939,209	72,808	0.089	155,985	1,573,161	3,640,343
15,207 (1.30)	4,525,541	99,759	0.113	206,089	3,180,475	6,505,603
16,377 (1.40)	6,702,964	38,548	0.051	78,295	4,062,971	1,480,169
17,547 (1.50)	6,876,526	4,934	0.029	9,971	4,274,085	22,461
(*) 18,040 (1.54)	6,876,928	3,616	0.028	7,233	4,281,461	7,233

(*): 最大コスト値 (このとき $f = h$ となる)

表 3 ランダムコストを持つ $n \times n$ 格子グラフへの適用結果

8 × 8 格子グラフ ($V: 81, E: 144$)

閾値 (近似率)	解の個数	ZDD 節点数	提案手法		通常のメモ化 呼出し回数	分枝限定法 [23] 呼出し回数
			時間 (秒)	呼出し回数		
113,552 (1.00)	1	80	0.087	93,393	151,236,577	93,375
114,688 (1.01)	2,854	7,693	0.092	112,121	153,108,163	164,809
115,823 (1.02)	125,880	82,363	0.165	285,709	155,502,227	2,251,091
116,959 (1.03)	2,364,994	432,455	0.639	1,046,119	157,794,855	31,308,123
119,230 (1.05)	133,629,862	3,094,302	5.523	6,473,755	163,635,511	1,018,509,845
122,636 (1.08)	1,893,484,003	5,848,008	10.976	11,893,469	213,488,793	3,555,322,287
124,907 (1.10)	2,635,409,530	2,101,940	3.412	4,281,999	258,640,783	530,661,095
127,178 (1.12)	2,687,915,888	218,309	0.255	450,137	274,526,635	7,463,305
(*) 130,079 (1.15)	2,688,307,514	46,613	0.090	93,227	275,391,699	93,227

(*): 最大コスト値 (このとき $f = h$ となる)

10 × 10 格子グラフ ($V: 121, E: 220$)

閾値 (近似率)	解の個数	ZDD 節点数	提案手法	
			時間 (秒)	呼出し回数
170,010 (1.00)	1	120	0.588	997,797
171,710 (1.01)	416,589	276,180	0.896	1,641,231
173,410 (1.02)	270,414,340	10,388,829	20.667	23,437,909
175,110 (1.03)	26,560,896,936	89,730,352	219.796	186,280,687
178,511 (1.05)	10,319,390,767,690	586,360,102	1,684.215	1,183,335,939
183,611 (1.08)	623,456,177,103,148	1,154,540,999	3,411.512	2,318,089,817
187,011 (1.10)	1,311,263,635,264,660	1,002,804,299	2,980.704	2,009,425,775
190,411 (1.12)	1,442,845,484,382,530	460,708,572	1,255.781	923,313,563
195,512 (1.15)	1,445,778,909,234,550	3,599,172	5.565	7,224,627
(*) 198,385 (1.17)	1,445,778,936,756,068	498,417	0.664	996,835

(*): 最大コスト値 (このとき $f = h$ となる)

実験環境は ASP システム clingo 5.4.0, MacBook Pro (3.5 GHz Intel Core i7, 16GB メモリ) を使用した。まず Knuth の例題に適用したところ、本手法と同じ個数の解を列挙することができたが、例えばコスト閾値 15,207 (倍率 1.30) の場合で 64.7 秒の計算時間を要し、本手法よりも数百倍程度の計算時間を要することが観察された。10 × 10 格子グラフの例題では、clingo では 1 時間かかってもハミルトンパスを 1 つも見つけない状態であった。圧倒的な性能差が存在すると思われる。

6. 関連研究

区間メモ化の類似技法は、どこまで過去に遡れるか定かではないが、BDD 技術との関連としては文献 [3] がおそらく初出と思われる。この既存研究では、SAT ソルバで疑似ブール制約 (線形不等式の論理式) を扱うために、一旦 BDD を構築し、それをさらに CNF 論理式に変換して SAT ソルバに入力する手法について論じているが、BDD を効率良く構築するために本研究と同様の区間メモ化の技法を提案している。ただし既存研究では対象が疑似ブール論理

関数のみに限定されているのに対し、本提案手法は任意の論理関数を扱える技法であることが異なる。また既存研究では最終的に SAT ソルバを用いるため、充足判定はできるが解を効率良く列挙できないという点でも大きく異なる。

7. おわりに

本稿では、ZDD の区間メモ化探索技法により、コスト制約組合せ問題の解を高速かつ厳密に列挙するアルゴリズムを提案した。本手法の活用により、観測した解のランキング（自分よりコスト値が上位の解が全部で何個あるか）を正確に求めることや、近似率を保証した準最適解のランダムサンプリングなど、コストに関係する様々な解析が可能となる。本手法の計算時間は、入力と出力の ZDD サイズを $|f|$ と $|h|$ とすると、多くの実用的な例題で $O(|f| + |h| \log |h|)$ で抑えられる。実験の結果、ハミルトンパス問題の現実的な例題において、数十億通りにおよぶコスト制約解をわずか数秒で厳密に全列挙できることが示された。特に、ZDD の圧縮率が高い場合には、既存手法と比べて圧倒的に高速に動作する。

本稿で提案したアルゴリズムは分枝限定法と動的計画法という 2 つの古典的技法を統合した形になっており、最適化と列挙を同じプログラムで効率良く実行できるという優れた性質を持つ。さらに本手法は、実行可能解の集合を ZDD で表現できるならば、ハミルトンパス問題に限らずどのような組合せ問題にも同様に適用可能であることから、現実の様々な問題への工学的応用が期待される。各アイテムのコストは正負の値が混ざっていても良く、コストの整数値の有効桁数が非常に大きくても支障なく動作する。今後の課題としては、非線形のコスト関数や多目的関数の問題への拡張も興味深い。

謝辞

本研究の一部は、JSPS 科研費 基盤 (A) 「列挙と最適化の統合的技法」(課題番号 20H00605) および学術変革 (A) 「アルゴリズム基盤」(課題番号 20H05964) の助成による。

参考文献

- [1] IBM ILOG CPLEX Optimization Studio V12.10.0 documentation, <https://www.ibm.com/jp-ja/products/ilog-cplex-optimization-studio/> (2019).
- [2] Gurobi Optimization: Gurobi Optimizer Reference Manual Version 9.1, <https://gurobi.com/> (2020).
- [3] Abío, I., Nieuwenhuis, R., Oliveras, A., Rodríguez-Carbonell, E. and Mayer-Eichberger, V.: A New Look at BDDs for Pseudo-Boolean Constraints, *Journal of Artificial Intelligence Research*, Vol. 45, pp. 443–480 (online) (2012).
- [4] Bryant, R. E.: Graph-Based Algorithms for Boolean Function Manipulation, *IEEE Trans. Computers*, Vol. 35, No. 8, pp. 677–691 (online) (1986).
- [5] Eppstein, D.: Finding the k Shortest Paths, *SIAM J. Comput.*, Vol. 28, No. 2, pp. 652–673 (online) (1998).
- [6] 湊 真一 (編) ERATO 湊離散構造処理系プロジェクト (著): 超高速グラフ列挙アルゴリズム—〈フカシギの数え方〉が拓く、組合せ問題への新アプローチ、森北出版 (2015).
- [7] Fifield, B., Imai, K., Kawahara, J. and Kenny, C. T.: The Essential Role of Empirical Validation in Legislative Redistricting Simulation, *Statistics and Public Policy*, Vol. 7, No. 1, pp. 52–68 (online) (2020).
- [8] Gebser, M., Kaminski, R., Kaufmann, B. and Schaub, T.: *Answer Set Solving in Practice*, Synthesis Lectures on Artificial Intelligence and Machine Learning, Morgan and Claypool Publishers (2012).
- [9] Hosaka, K., Takenaga, Y., Kaneda, T. and Yajima, S.: Size of ordered binary decision diagrams representing threshold functions, *Theoretical Computer Science*, Vol. 180, No. 1-2, pp. 47–60 (1997).
- [10] Inoue, T. et al.: Graphillion, <http://graphillion.org/> (2013).
- [11] Inoue, T., Iwashita, H., Kawahara, J. and Minato, S.: Graphillion: software library for very large sets of labeled graphs, *International Journal on Software Tools for Technology Transfer (STTT)*, Vol. 18, No. 1, pp. 57–66 (2016).
- [12] Inoue, T., Takano, K., Watanabe, T., Kawahara, J., Yoshinaka, R., Kishimoto, A., Tsuda, K., Minato, S. and Hayashi, Y.: Distribution Loss Minimization With Guaranteed Error Bound, *IEEE Transactions on Smart Grid*, Vol. 5, No. 1, pp. 102–111 (2014).
- [13] Ishioka, F., Kawahara, J., Mizuta, M., Minato, S. and Kurihara, K.: Evaluation of Hotspot Cluster Detection using Spatial Scan Statistic based on Exact Counting, *Japanese Journal of Statistics and Data Science*, Springer, Vol. 2, No. 1 (online) (2019).
- [14] Kawahara, J., Horiyama, T., Hotta, K. and Minato, S.: Generating All Patterns of Graph Partitions within a Disparity Bound, *Proc. of the 11th International Conference and Workshops on Algorithms and Computation*, Vol. 10167, pp. 119–131 (online) (2017).
- [15] Knuth, D. E.: *The Art of Computer Programming: Bit-wise Tricks & Techniques; Binary Decision Diagrams*, Vol. 4, fascicle 1, Addison-Wesley (2009).
- [16] Miller, C. E., Tucker, A. W. and Zemlin, R. A.: Integer programming formulation of traveling salesman problems, *Journal of the ACM (JACM)*, Vol. 7, No. 4, pp. 326–329 (1960).
- [17] Minato, S.: Zero-suppressed BDDs for set manipulation in combinatorial problems, *Proc. of 30th ACM/IEEE Design Automation Conference (DAC'93)*, pp. 272–277 (1993).
- [18] Nishino, M., Yasuda, N., ichi Minato, S. and Nagata, M.: BDD-constrained Search: A Unified Approach to Constrained Shortest Path Problems, *Proc. of the 29th AAAI Conference on Artificial Intelligence (AAAI2015)*, pp. 1219–1225 (2015).
- [19] Tamura et al.: Sugar: a SAT-based Constraint Solver, <https://cspSAT.gitlab.io/sugar/> (2018).
- [20] Tamura, N., Taga, A., Kitagawa, S. and Banbara, M.: Compiling Finite Linear CSP into SAT, *Constraints*, Vol. 14, No. 2, pp. 254–272 (online) (2009).
- [21] Wasa, K.: Enumeration of Enumeration Algorithms, *CoRR*, Vol. abs/1605.05102 (online), <http://arxiv.org/abs/1605.05102> (2016).
- [22] 湊 真一: BDD/ZDD を用いたグラフ列挙索引化技法 (特集 BDD/ZDD を用いた新しい列挙索引化技法 (フロンティア法) とその応用), *OR 学会誌*, Vol. 57, No. 11, pp. 597–603 (2012).
- [23] 湊 真一, 番原睦則, 堀山貴史, 川原 純, 瀧川一学, 山口 勇太郎: コスト制約つき組合せ問題に対する ZDD を用いた高速な解列挙手法, *電子情報通信学会技術研究報告*, Vol. 120, No. 276 (COMP2020-19), pp. 8–15 (2020).