Vol.2014-AL-147 No.16
2014/3/4

ZDDを用いたExact Cover問題に対するパレート最適な解の列挙

松永　涼[1]　斎藤　寿樹[1]　近藤　広樹[1]　中野　浩太郎[2]

概要：Exact Cover 問題は数多くの実応用をもつ最適化問題の 1 つである. 多くの最適化問題では, 目的関数を複数持つ多目的最適化問題を考えることが多い. 多目的最適化問題ではすべての目的関数の値が最適である解が存在することはほとんどない. そのため, パレート最適解を求める発見的手法が提案されている. しかし，これらの発見的手法には最適性の保証がなく，出力される解に偏りが生じやすい. 本研究では一部のパレート最適解を出力するのではなく, 全てのパレート最適解を出力することを考える．そのため,Exact Cover を全て出力する 3 つのアルゴリズムを紹介し, それらのアルゴリズムを拡張してパレート最適解を列挙する手法を提案する.

1. はじめに

Exact Cover 問題は集合 X と X の部分集合の集合 Y が与えられたとき，Y からいくつかの集合を選んで，X の分割を求める問題である．この問題は，集合分割問題やハイパーグラフの完全マッチング問題などとも呼ばれる NP-完全問題の１つで，領域分割問題やスケジューリング問題など，数多くの実応用が存在することが知られている [1], [2]．こうした実応用では，目的関数を複数持つ多目的最適化問題を考えることが多い．

多目的最適化問題は各目的関数値の最適化を行うが，すべての目的関数が最適である解が存在するとは限らない．そのため，パレート最適解，つまり，ある目的関数を改善するには他の目的関数値を悪くしなければならない解，を求める発見的手法が提案されている [3]．しかし，これらの発見的手法はパレート最適性の保証を与えておらず，また出力解の偏りにより類似した解が出力される傾向にある．この問題を解決するために，パレート最適解を列挙することを考える．

パレート最適解を列挙する素朴なアルゴリズムに，まず，実行可能解を列挙し，その実行可能解の中からパレート最適解を抽出する，という方法がある．Exact Cover を列挙するアルゴリズムは，Knuth による Algorithm X [4], ZDD の演算を用いた手法 [2], [5]，ZDD のトップダウン構築法 [6], [7] の３つが知られている．１つ目の Algorithm X は単純なバックトラックアルゴリズムであるが，Dancing Links と呼ばれるデータ構造を用いることで，高速に動作することが知られている．２つ目と３つ目の手法は ZDD(Zero-suppressed Binary Decision Diagram) と呼ばれるデータ構造を用いている．ZDD は集合の集合をコンパクトに表現することができるだけでなく，ZDD が表現する集合の集合同士に対しての様々な集合族演算を効率的に行うことができる [8]．２つ目の手法はこれらの演算体系を用いたもので，Exact Cover の集合を ZDD で構築する．しかし，この演算体系を用いた手法では，演算の途中で ZDD のサイズが指数爆発してしまうことがある．そのため，近年，ZDD をトップダウンに直接構築する手法が注目を集めている．３つ目の手法はそのトップダウン構築手法を用いて，ZDD を構築する．

本研究では，まず，Exact Cover を列挙する３つのアルゴリズムを計算機を用いて性能を比較する．３つのアルゴリズムはそれぞれ独立して開発されたものだが，実際に計算機上での比較を行われておらず，いずれの手法が有効であるかの議論が行われていなかった．また，それぞれのアルゴリズムを拡張し，重み付き Exact Cover 問題に対するパレート最適解を列挙するアルゴリズムを提案する．

2. 問題の定式化

2.1 Exact Cover 問題

Exact Cover **問題**は *Exact Cover* を求める問題である．X を $\{1,\ldots,m\}$ と Y を X の部分集合の集合 $\{Y_1, Y_2, \ldots, Y_n\}$ とする．このとき，X の要素がちょうど一回だけ現れる Y の部分集合を Exact Cover という．

このとき部分集合の集合は n 行 m 列の行列 Z で表現で

[1] 神戸大学大学院工学研究科
[2] 京都大学大学院工学研究科

きる．行列 Z の i 行 j 列を $Z[i][j]$ と表し, 集合 Y_i が要素 j を含む時，$Z[i][j]$ を 1 とし，含まれないとき $Z[i][j]$ を 0 とする．また, ある行 y' がある列 x' に 1 をもつことを y' が列 x を**カバー**すると呼ぶ. 全体集合 X を列集合，部分集合の集合 Y を行集合 I と呼ぶ.

具体例を用いて説明する．$X = \{1,2,3,4,5,6,7\}$ とし，その部分集合を $Y = \{1,2,3,4,5,6\}$, $Y_1 = \{3,5,6\}$, $Y_2 = \{1,4,7\}$, $Y_3 = \{2,3,6\}$, $Y_4 = \{1,4\}$, $Y_5 = \{2,6\}$, $Y_6 = \{4,5,7\}$ とする．集合 Y_1 は要素 3 をもつため，$Z[1][3] = 1$ と表される．また, 集合 Y_1 は要素 4 をもたないため，$Z[1][4] = 0$ となる．式 (1) に部分集合の集合を表す行列を示す.

$$Z = \begin{array}{c} \\ Y_1 \\ Y_2 \\ Y_3 \\ Y_4 \\ Y_5 \\ Y_6 \end{array} \begin{array}{c} \begin{array}{ccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 \end{array} \\ \begin{pmatrix} 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 \end{pmatrix} \end{array} \quad (1)$$

(1) の行列において，Exact Cover 問題とは，各列の 1 の数がちょうど 1 となる行の集合を求めることである．この場合，行の集合 $Y^* = \{1,4,5\}$ は Exact Cover である．なぜなら，集合 $\{Y_1, Y_4, Y_5\}$ はそれぞれ共通の列に 1 を持たず，各列はいずれかの行がカバーしているためでる.

また，$Y^{*'} = \{Y_3, Y_4, Y_6\}$ は Exact Cover ではない．なぜならば，集合 $\{Y_3, Y_4, Y_6\}$ は 4 行と 6 行が共に 4 列をカバーしてしまっているためである.

重み付き *Exact Cover* **問題**は，各部分集合に重みがつき, さらに目的関数が与えられた Exact Cover 問題である．重み付き Exact Cover 問題は，避難所割り当て問題や選挙区割り問題など実用的な応用が知られている.

2.2 多目的最適化とパレート最適

目的関数が複数存在する最適化問題を**多目的最適化問題**と呼ぶ．l 個の目的関数を持つ多目的最適化問題を以下のように定義する.

$$minimize\ (f_1(x), \ldots, f_l(x))$$
$$subject\ to\ \ x \in \Omega$$

ここで $f_1(x), \ldots, f_l(x)$ は目的関数であり, $x \in \Omega$ は実行可能領域である.

単目的最適解とは，ある目的関数に対して最適な解である. ある x' がある目的関数 k に対して単目的最適解であるとき，以下の式 (2) を満たす.

$$\forall x \in \Omega, \exists k \in \{1, \ldots, l\},\ \ f_k(x') \leq f_k(x)$$

完全最適解とは，どの目的関数に対しても最適な解である．ある x' が完全最適解であるとき，以下の式 (2) を満たす.

Input : n 行 m 列の行列 Z
Output : Exact Cover である行の集合
1.**if** Y が空行列 (Exact Cover を発見) **then** Exact Cover を出力
2. 1 の数が最も少ない列 c を選択する
3. **if** 列 c の 1 の数が 0 **then** 解探索失敗
4.**for** 列 c が 1 である行 r
5. 行 r に 1 がある列と, その列に 1 をもつ行は削除
6. 削除された行列に AlgorithmX を適用
7. 行列を復元

図 1 AlgorithmX の動作

$$\forall x \in \Omega, \forall k \in \{1, \ldots, l\}, f_k(x') \leq f_k(x)$$

複数の目的関数を考慮すると, 完全最適解を発見したい. しかし, 完全最適解は常に存在するとは限らない．通常はある目的関数に対して最適な x' は，他の目的関数に対しては最適ではない場合が多い．このような問題に対処するために，**パレート最適**という概念が考え出された．解 x' がパレート最適解であるとき, 任意の $x \neq x'$ に対して以下の式 (2) を満たす.

$$\exists j \in \{1, \ldots, l, \}\ f_j(x') < f_j(x)$$

パレート最適の直感的な定義は，次のように表現できる「x' がパレート最適解であるとは， $f_x(x)$ を全ての点で下回るような $x \in \Omega$ は存在しない事を言う」[3].

3. パレート最適解の列挙アルゴリズム

本章では Exact Cover を列挙する 3 つの手法を紹介する. また，各行が複数の重みを持つ重み付き Exact Cover 問題も扱う. この問題はそれぞれの重みの和を最小化する多目的最適化問題である. そこで, それぞれの手法を拡張したパレート最適な解を列挙する手法を提案する.

3.1 AlgorithmX

AlgorithmX とは，深さ優先のバックトラッキングアルゴリズムで Knuth によって提案された [4].

3.1.1 解の列挙

AlgorithmX を図 1 に示す．アルゴリズムのステップ 4 では行 r を選択する. このとき, 選択した行を解の候補として記録しておく. ステップ 1 で出力する Exact Cover はそのとき記録していた行の集合である. また, このとき選択した行 r が 1 をもつ列 c' は, 行 r によってカバーされる. よって列 c' に 1 をもつ各行をこの後選択することはできなくなる. そのため, ステップ 5 で行 r に 1 がある列と, その列に 1 をもつ行 c' を行列 Z から削除する. ステップ 6 では AlgorithmX を再帰的に実行し, ステップ 7 では削除した行, 列を復元をする.

行列を配列で表現すると行や列の削除，および行や列の復元を効率よく行うことができない．データ構造 Dancing Links を用いることでこの問題を解決する．この Dancing

Input : 解の集合 $\{X' = \{x_1, \ldots, x_n\}\}$
Output : パレート最適解の集合 P
0.P を X とする.
1.1 つの目的関数値を 1 桁として，基数ソートを行う．ソート結果を $\{x'_1, \ldots, x'_n\}$とする.
2. **for** i=1 **to** n
3. **for** j=i+1 **to** n
4. **if** $\forall k \in \{1,\ldots,l\}$ $f_k(x_i) \leq f_k(x_j)$
5. **then** 解 x_c はパレート最適でないので P から削除する.
6. 残った解の集合 P を出力する.

図 2 パレートサーチ

Links についての詳しい説明は文献 [4] を参照せよ．このデータ構造ではある要素を削除することや，"1"の検索も効率よく行うことができる．よって，Dancing links を用いる事により，配列を用いた場合より高速に実行可能な AlgorithmX を動作させることが出来る.

3.1.2 パレート最適解の列挙

AlgorithmX は Exact Cover（解）を全て列挙するので，その中には必ずパレート最適である解が含まれている. そのため，パレート解を列挙するアルゴリズムとして，まず AlgorithmX で全解を列挙して，その中からパレート最適解を抽出することが考えられる. 全ての解の集合からパレート最適解を列挙するナイーブなアルゴリズムを図 3.1.2 に示す. このアルゴリズムをパレートサーチと呼ぶ. ステップ 2 では, 目的関数 f_i に対して解をソートする. 次にステップ 3 と 4 で, ある解 c が f_i の値で劣っている解に対して他の目的関数値でも劣っているかを調べる. 他の目的関数値でも劣っていた場合, この解はパレート最適ではない. 劣っている解がみつからない場合, これはパレート最適解といえ, ステップ 7 で出力する.

AlgorithmX の特徴として Exact Cover を発見するとその場で出力する点がある．また, 全解を探索するため Exact Cover の数が多くなると時間が多くかかる. そこでパレート最適を含む一部の解空間を列挙し, そこからパレートサーチ (図 3.1.2) でパレート最適解を列挙するアルゴリズムを提案する. AlgorithmX 中で発見した各単目的の暫定最適解を l 個保持する. これらを使い,AlgorithmX のステップ 4 である行を選択するとき, その先で得られる解がパレート最適になりえないと判断できる探索を枝刈りしたい. 例えば,2 つの目的関数 f_1, f_2 を考えるとき, 図 3 で白い解空間はパレート最適解が必ずないと判断できる．白い部分に存在する解は X_1, X_2 のどちらか，もしくは両方に目的関数 f_1 及び f_2 の値で劣っているためパレート最適ではない. 探索途中の解候補がこの空間に到達したとき，目的関数値が減ることはないので，これ以上探索して得られる解はパレート最適になりえない.

そこで,AlgorithmX のステップ 4 である行を選択するとき図 (4) の判定をする．これが true ならそのまま探索を続け,false なら探索失敗とすることで探索範囲を狭くする. 目

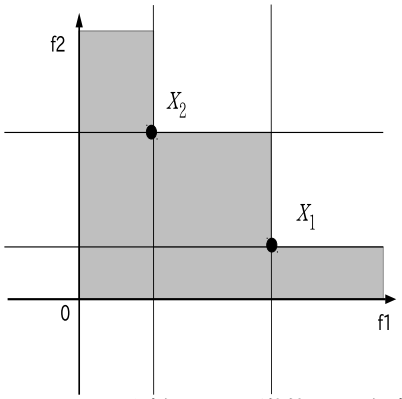

図 3 パレート最適解がある可能性がある解空間

Input : 暫定最適解の集合 $\{x_1, \ldots, x_l\}$, 暫定解候補 x
Output : true **or** false
1.$\exists f_i(x), f_j(x) \in (f_1(x), \ldots, f_l(x))$
2. **if** $f_i(x) \leq f_i(x_i)$ **then** a=true **else** a=false
3. **else if** $f_j(x) \leq f_j(x_i)$ **then** b=true **else** b=false
4. **if** $f_j(x) \leq f_j(x_i)$ **then** c=true **else** c=false
5. **else if** $f_i(x) \leq f_i(x_j)$ **then** d=true **else** d=false
6. **if** (a または c) または (b かつ d) **then** return true **else** return false

図 4 解空間を狭くするための判定アルゴリズム

的関数 $f_1(x)$ における暫定最適解を x_1, 同様に $f_2(x)$ における暫定最適解を $x_2, \ldots, f_n(x)$ における暫定最適解を x_l とする. 変数 a_1, a_2, b_1, b_2 をブール変数とおく. true か false を格納できるブール変数である. 入力の暫定の暫定最適解は最初 (AlgorithmX の動作前) はそれぞれの目的関数値の重みの総和としている. しかし, あらかじめいくつか解を保持しているような場合があればそれを利用することでさらに解の探索空間を減らすことができる. ステップ 2, および 4 では, 単目的最適解より解候補の目的関数値が優れているかどうかを判定する. 単目的最適解より優れている解候補はステップ 6 のとおり, そのまま探索を続ける. ステップ 3, および 5 では, 単目的最適解より劣っている解候補のなかでもパレート最適な解があるかどうかを判定する. 単目的最適解に対して, 他の目的関数値でも劣っている解はパレート最適ではない. 逆に, 他の目的関数値で勝っているならばパレート最適な解である可能性が残されている. よって探索を続ける.

また,AlgorithmX が解を出力するステップ 1 で, 単目的最適解の更新をおこなう. 解の更新を行うアルゴリズムを図 5 に示す. 出力する解 x' がある目的関数 f_i に対して以下の式を満たすとき x_i を更新する.

$$\exists f_i(x) \in (f_1(x), \ldots, f_l(x)), \quad if \quad f_i(x') \leq f_i(x) \tag{2}$$

出力する解 x' がある単目的最適解 x_i に対して目的関数 f_i の値で優れているとき,x' を x_i と更新する. また, 値が等しいならば x_i に対して別の目的関数 $f_j \in (f_1(x), \ldots, f_l(x))$ の値で優れているときのみ, x' を x_i と更新する.

Input : 暫定最適解の集合 $\{x_1, \ldots, x_l\}$, ある解 x'
Output : 暫定最適解の集合 $\{x_1, \ldots, x_l\}$
1.$\exists f_i(x) \in (f_1(x), \ldots, f_l(x))$
2. **if** $f_i(x') \leq f_i(x_i)$ **then** $x_i \leftarrow x'$
3. **else if** $f_i(x') == f_i(x_i)$
4. **if** $i \neq j$ かつ $\exists f_j(x) \in (f_1(x), \ldots, f_l(x)), f_j(x') \leq f_j(x_i)$
then $x_i \leftarrow x'$

図 5 暫定の単目的最適解を更新するアルゴリズム

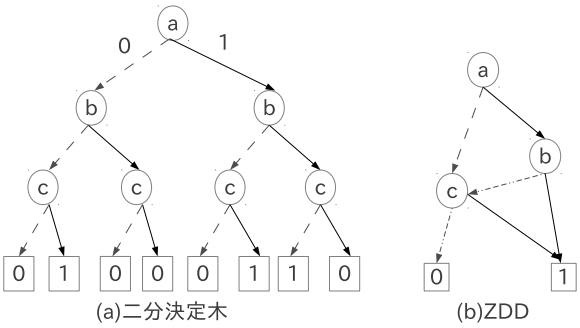

図 6 二分決定木と等価な ZDD

3.2 ZDD の演算を用いた手法

ゼロサプレス型 BDD(ZDD) は集合族（集合の集合）をコンパクトに表現できる非閉路有向グラフである．まず，簡単のため，二分決定木で集合の集合を表現する（図 6(a)）．二分決定木の内部ノードは集合の１つの要素と２つの枝を持ち，２つの枝のうち一方を１枝，もう一方を０枝という．また，葉には２種類存在し，それぞれ０終端と１終端と呼ぶ．また，根以外のノードは入力枝を持つ．ここで，二分決定木の根から１終端までの経路と出力枝が１枝を通っているノードの要素の集合が一対一に対応する．例えば，図 6(a) の図において，根から１終端までの経路は３つあり，それぞれの経路と対応する集合は $\{c\}, \{a, c\}, \{a, b\}$ である．そのため，この二分決定木は集合の集合 $F = \{\{c\}, \{a, c\}, \{a, b\}\}$ を表している．

しかし，集合の集合を二分決定木で表現すると，集合の要素数が増えると二分決定木のノード数は指数的に増加する. そこで，二分決定木から冗長なノードを取り除いて，集合の集合をコンパクトに表現する，というのが *ZDD* である (6(b)).

ZDD は以下の２つの簡約化規則を適用してノードを削減する.

- ノードの削除 (図 7 (a))
- 等価なノードの共有 (図 7 (b))

ZDD を集合の集合同士を処理する様々な演算が知られている． 以下に, 本研究で使用する ZDD の演算について説明する．F，G は ZDD を示し，x は集合の要素を示す.

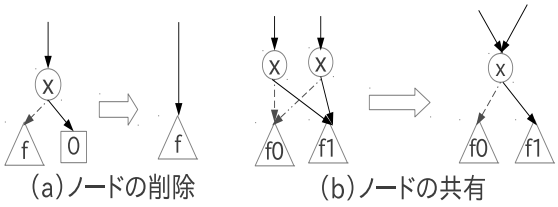

図 7 ZDD の二つの簡約化規則

要素加算

F のすべての集合に要素 x を加える.

集合加算

集合加算 $(F \times G)$ とは, F の各集合と G の各集合のすべての組合せについて，和集合を行ったものの集合を表す.

Restrict 演算

$F.Restrict(G)$ とは, F に含まれる集合のうち G の集合の少なくとも 1 つを包含するような集合だけを残して, それ以外を削除する．例えば，$F = \{b, ac, cd, abc, bcd\}, G = \{a, bc\}$ のとき，$F.Restrict(G) = \{ab, abc, bcd\}$ である.

これらの演算を用いて，ExactCover をすべて保持する ZDD を構築する.

3.2.1 解の列挙

実用的な応用を考えた Exact Cover 問題では, 列集合に特別な s 個の要素 $S' = \{j_1, j_2, \ldots, j_s\}$ が存在することが多い. これらの要素は Exact Cover の中でそれぞれ同じ集合に含まれてはならない要素である．つまり，Exact Cover この特別な要素をちょうど１つ持つ

列集合にこの要素の部分集合があることで, 集合をちょうど s 個に分割することができる. この列集合 S を**分割集合**と呼ぶことにする. 本論文で示す Exact Cover の応用でも, 分割集合 S を使って分割する数を指定している. 分割要素 s_i を要素に持つ行の部分集合を Y_i と書く. また，ZDD_i とは Y_i を含む ZDD のことである. *OZDD* とは，二つ以上の分割要素を含む集合の集合を示す ZDD である. *SZDD* はいずれかの分割要素を含む集合を示す ZDD である. 分割集合を利用したアルゴリズムを 8 に示す.

Input : n 行 m 列の行列 Z, 分割集合 $S = \{1, \ldots, s\}$
Output : 全ての Exact Cover をもつ ZDD F
1.$ZDD \leftarrow \emptyset$
2.**for** 0 から s までの分割集合 s_i
3. s_i を含む行の部分集合 Y_i について ZDD_i を構築
4. **if** i が 1 のとき $ZDD \leftarrow ZDD_i$
5. **else**
6. $ZDD \leftarrow ZDD \times ZDD_i$
7. $ZDD \leftarrow ZDD \setminus ZDD.Restrict(OZDD)$
8. $ZDD \leftarrow ZDD.Restrict(SZDD)$
9.ZDD を出力

図 8 ZDD の演算を用いた ZDD の構築法

Input : ZDD F
Output : パレート最適解の集合 sol
1.**for** F のノード i (根から)
2. **if** 枝の先のノードで他の経路と合流
3. **then** 枝の sol がもつ解候補と, i から伸びる解候補のうち, パレートサーチを用いてパレート最適な解候補の集合を sol に残す
4. **else if** 枝の先のノードが他の経路を持たない **then** 枝の先のノードの sol に i から伸びる解候補を格納
5.1 終端がもつ sol を出力

図 9 ZDD からパレート最適な解を取り出すアルゴリズム

3.2.2 パレート最適解の列挙

演算で構築した ZDD を入力として, パレート最適な解のみを出力する. ZDD のトップのノードから順に枝を見て, あるノードが共有するとき, このノードに至る経路はこのノードから先の部分で同じ経路を通る. よってこのノードに至る経路が示す解候補がすでに他の経路に対してパレート最適でないとき, この解候補がパレート最適になることはないと考えられる. そのノードに解候補のうち, パレート最適であるものだけを残す. 各ノードが, パレート最適となり得る解候補の集合を sol に格納する. ステップ 2, 3, 4 で枝と記述されている部分は 0 枝, 1 枝ともに同じ動作を行う. 実行可能な解が示す経路はすべて 1 終端に到達する. よって, 最終的に 1 終端の sol には全てのパレート最適解が格納されている.

3.3 フロンティア法を用いた手法

ZDD の演算を使う場合, 計算途中でノード数が爆発的に増えてしまい計算できなくなる傾向がある. そこで, ZDD の構築に工夫が必要になる. ZDD を一番上から幅優先的に構築する手法をトップダウン構築法という. そこでトップダウンで効率よくノードの削除, 共有を行って ZDD を構築する手法を提案する.

3.3.1 解の列挙

トップダウンで構築する過程で以下の操作を適宜行って行く. アルゴリズムを図 10 に示す. この中の枝刈り, ノードの共有については後に説明する.

Input : n 行 m 列の行列 Z
Output : ZDD F
0. 初期ノード v_0 を作成する 1.**for** $i = 0$ to n
2. **for** 変数番号 i のノードの
3. **if** 各枝が枝刈りができるとき **then** 枝刈りを行う
4. **else if** 枝の先のノードが共有できるとき **then** 共有を行う
5. **else** 枝の先にノードを作成, 枝をノードに直結させる
6.v_0 を根とする DAG を既約化したものを F とする **return** F

図 10 フロンティア法を用いた ZDD の構築法

枝刈り

あるノード v から出る枝の先のノード v' を構築中に, v' から到達できる終端が全て 0 終端と判定できるとき, その枝は枝刈りできる. 具体的には以下のときに, 枝刈りができる.

- ある列が 2 度カバーされるとき
- ある列がカバーされないとき

ノードの共有

各ノードから枝を伸ばす前に, その枝の先が他のノード v と共有できるかを判定する. ノードが v と共有できると判断できた場合, すでに v と枝を伸ばす.

上記の操作をする判定は $cover$ とフロンティアを使って判定する.

cover 配列

各ノードが cover という配列をもつ. cover はそのノードに至る経路がどの列に 1 を持つかの情報をもつ. 同じ深さの 2 つのノードの cover が一致するとき, それらのノードは共有できる. なぜなら, それらの経路がど 1 をもつ列が一致し, これから 1 を持たなくてはいけない列が一致するからである.

フロンティア

全ノードが全ての要素の情報を cover を持つと計算機の容量が足りなくなる. そこで枝刈り, 共有が判定できる部分だけ cover を持つことで容量を節約する. この部分を フロンティアという. また, フロンティアを利用して枝刈りを行うことができる. ある要素がフロンティアから外れるとき, その要素はこれ以上カバーされることはありえない. よって, フロンティアから外れる要素がカバーされていないとき, 枝刈りができる.

フロンティアに属するノードのカバー配列をみることで, ZDD をトップダウンで構築しながら枝刈りと共有を行うことができる.

3.3.2 パレート最適解の列挙

3.2.2 と同様に, ノードの共有を行うときにパレートサーチでパレート最適である解候補を示す経路だけを記録しておく. ただし, フロンティア法ではこの操作を ZDD の構築と同時に行う. ここでも各ノードが, パレート最適となり得

る解候補の集合を *sol* に格納する. *sol* がもつ解候補と *i* から伸びる解候補からパレートサーチを用いてにパレート最適な解候補を残す.

4. 実験環境

- CPU:Intel(R)Xeon(R)E5-2650@2.00GHz
- OS:lCentOS6.5
- Memory:128GB

5. インスタンス

5.1 敷き詰め問題

敷き詰め問題とは，与えられたピースを与えられた立体内に配置する問題のことである．ただしピースどうしの重なりがなく，立体内の領域全てを敷き詰めなくてはならない．敷き詰め問題の例としてペントミノパズル等などが知られている．この問題は Exact Cover 問題に定式化することができる.

敷き詰め問題を次のように Exact Cover 問題に定式化する．与えられた立体の盤面とピースの数を指定する分割要素を合わせて列集合 J，与えられた各ピースの全てのパターンを行集合 I として行列で表す．各ピースは一つしかなく，各ピースは三次元グリッド上の平行移動および３つの回転 (x 軸，y 軸，z 軸) 操作を行うことができる．ピースを平行移動，回転させることで，考えられる全てのピースの形を考える．一つの部分集合はどのピースか区別する要素とピースがどの形置かれているかを示す要素からなる.

この問題では特に目的関数は存在しないが, ランダムデータを用いて目的関数があると仮定してパレート最適な解の列挙をおこなった.

5.2 避難所割り当て問題

避難所割当問題はある地域とそこに住む人口, また避難所とその容量が与えられたとき, それぞれの避難所にどの地域が割当されるか決定する問題である. 避難所割当問題は文献**??**のステップ 1 のとおり重み付き Exact Cover 問題に定式化することができる.

避難所を含む全ての地域を列集合 J，各避難所に割当可能な地域の組合せを行集合 I として行列で表す．この問題では一つの地域が同時に複数の避難所に割当されることはない. よって避難所を示す地域の要素を分割要素として考えることができる. この問題の目的関数は

$$minimize(dis(x), cap(x)) \tag{3}$$

の２つである. $dis(x)$ は各地域から各避難所への距離の総和である. できるだけ近い避難所に避難するのが望ましいという考えからきている. $cap(x)$ は各避難所の容量と, そこに割当されている地域の人口の合計の和の比の総和である. 特定の避難所だけ他の避難所に比べ多く割当されると比の総和は大きくなる.

文献 [2] では京都市上京区のデータを用いた実験がされている. 本研究では大阪市住吉区の町丁目データを用いている.

6. 実験結果

実験環境は以下のとおりである.

6.1 敷き詰め問題

敷き詰め問題に対して３つの手法を適用した結果を表2,1 に示す. 一番左の列, 立体は Exact Cover に定式化した元の立体を示す. 立体の番号がどの立体を示すかは, 図に示す. その右の列, および行は, 行列に定式化したときの列数, 行数である. その右はそれぞれの手法が ExactCover を列挙するためにかかった時間 [s] である. 表 1 では, 盤面の数が 27 個, ピースの数が 7 個の立体 easy cube を扱い,2 では少し大きめで, 盤面の数が 64 個, ピースの数が 15 個の立体 Dee cube を扱う.

表 1 easycube の実験結果

立体	行数	列数	解の個数	AlgorithmX	標準演算	frontier
1	417	34	396	0.05	t	4.41
2	365	34	410	0.02	t	2.72
3	313	34	150	0	t	1.57
4	450	34	748	0.09	t	5.48
5	368	34	446	0.02	t	2.8
6	522	34	5328	0.54	t	4.53
7	305	34	230	0.02	t	1.01
8	305	34	230	0.02	t	1.69
9	299	34	8	0	t	0.14
10	407	34	8	0.03	t	2.49
11	378	34	4	0.03	t	2.4
12	368	34	8	0	t	0.13

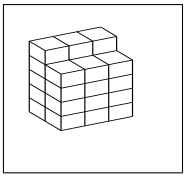

図 11 立体 1

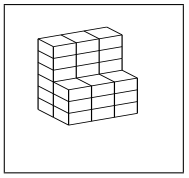

図 12 立体 2

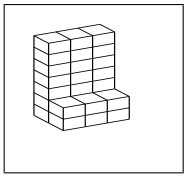

図 13 立体 3

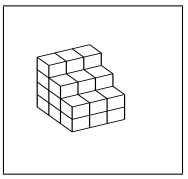

図 14 立体 4

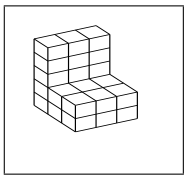

図 15 立体 5

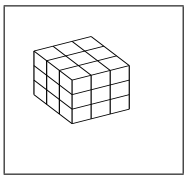

図 16 立体 6

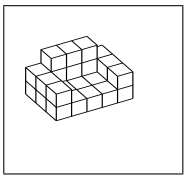
図 17 立体 7

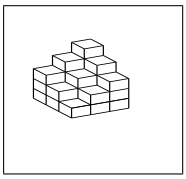
図 18 立体 8

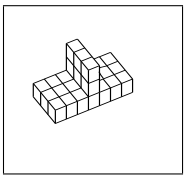
図 19 立体 9

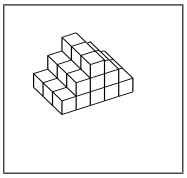
図 20 立体 10

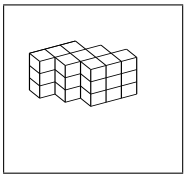
図 21 立体 11

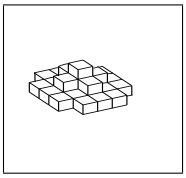
図 22 立体 12

表 2 Deecube の実験結果

立体	行数	列数	解の個数	AlgorithmX
14	1884	79	407260	538.43
15	1578	79	148971	0.0002
17	2650	79	27161	299.86
19	2498	79	2546653	1801
20	1959	79	338611	131.08
21	2243	79	703636	689.24
22	2469	79	350	1.38
23	1967	79	31870	38.14
24	1181	79	34	1.91
25	2140	79	103600	957.21
26	1703	79	6134	26.05
27	1911	79	886196	1709
28	1884	79	333313	398.74

表 2 では，標準演算とフロンティア法の欄がない．これは全ての立体に対してタイムアウトであったためである. 2 つの表をみると, いずれの立体でも AlgorithmX が高速に動作していることがわかる. 一方,ZDD に基づいた 2 つの手法では解を求めることができていない.

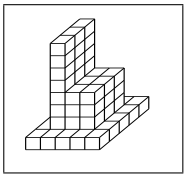
図 23 立体 14

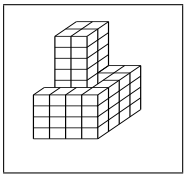
図 24 立体 15

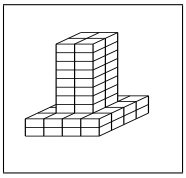
図 25 立体 16

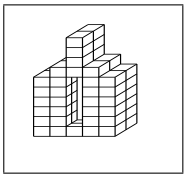
図 26 立体 17

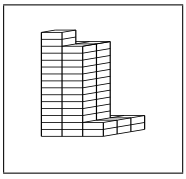
図 27 立体 18

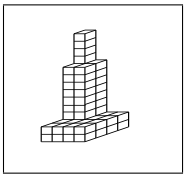
図 28 立体 19

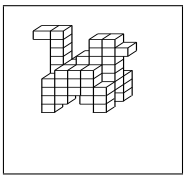
図 29 立体 20

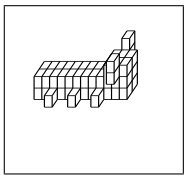
図 30 立体 21

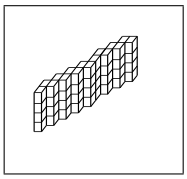
図 31 立体 22

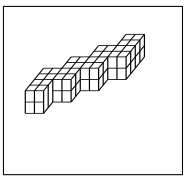
図 32 立体 23

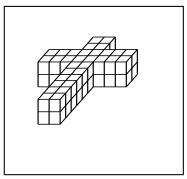
図 33 立体 24

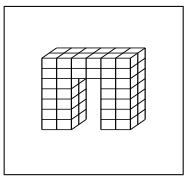
図 34 立体 25

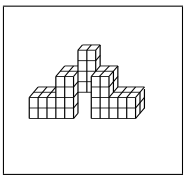
図 35 立体 26

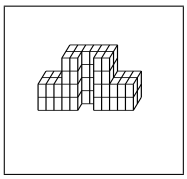
図 36 立体 27

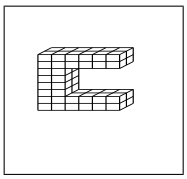
図 37 立体 28

6.2 避難所割り当て問題

この問題ではある避難所についての地域割り当て (行列では行にあたる) を制約を用いて分割している．表 ¥tb:evacuation の列を左から順に説明する. 一番左の列が, 制約を表す. その次が, 入力行列の行, 次が列でありその右が解の数である. そこから右は各手法の計算時間 [s] を示す. 制約が $1300m_CC25$ と記述されていると, 距離制約が $1500m$, 容量制約が $CC15$ ということを示している. 距離制約 $1500m$ は, ある避難所は自身を中心とした $1500m$ 圏内の地域を割り当てることが可能であるということを示す.$CC15$ は避難所の最大容量の何倍の人数を割当可能とするかを示す. これらの値が大きいほど, 入力行列のサイズが小さくなる.

敷き詰め問題のインスタンスと比べると, 入力行列のサイズが大きいことがわかる. このインスタンスに対しては ZDD の標準演算が唯一解を列挙することができた. 他の手法では解を最後まで求めることができなかった.

6.3 パレート最適解の列挙手法の比較

ParatoX を図 4, 及び図 5 を用いた AlgorithmX とおく.ParatoX とパレートサーチを用いてパレート最適を列挙する手法を実装した結果を表 4, 表 5 に示す．それぞれの立体に対して，行数及び列数は表 1，表 2，に示しているため, 省略している. パレート解の列はパレート最適解の数を，ParatoX 解は ParatoX で求めた解である.

入力行列のサイズが小さい easycube では差はほとんどない．若干遅くなっているように見えるものもあるが誤差の範囲である.Deecube では easycube と違い，2 つの手法に差があることがわかる.

表 3 避難所割り当ての実験結果

制約	行数	列数	解の個数	標準演算
1000m	2553	104	304843362072	0.47
1100m	6058	104	1360769050260	0.79
1200m	9293	104	10678290580576	0.98
1250m		104	1372620853490	1.57
1300m	12497	104	11879253967870	1.36
1300m_CC15	1563	104	882305033912	1.66
1300m_CC20	2922	104	4587702792236	3.75
1300m_CC25	1563	104	6994561316999	3.33
1500m	28702	104	19014990877396	1.91
1500m_CC15	1991	104	1080696991741	1.69
1500m_CC20	3979	104	6627301374420	5.23
2000m_CC8.0	615	104	0	0
2000m_CC9.0	770	104	33773984	0.34
2000m_CC10.0	975	104	10638067446	0.89

表 4 easycube のパレート最適解の列挙

立体	解	パレート解	ParatoX 解	AlgorithmX	paratoX
1	396	5	41	0.05	0.05
2	410	6	73	0.02	0.03
3	150	7	24	0	0.01
4	748	12	75	0.09	0.1
5	446	6	239	0.02	0.03
6	5328	6	1502	0.54	0.55
7	230	4	21	0.02	0.02
10	8	1	6	0.03	0.04
12	8	3	5	0	0.01

表 5 Deecube のパレート最適解の列挙

立体	解	パレート解	ParatoX 解	AlgorithmX	paratoX
14	407260	24	33174	538.43	301.72
21	27161	17	5932	299.86	265.44
24	2546653	33	393324	1801	1105.02
25	338611	11	105180	131.08	124.3
26	703636	16	174262	689.24	858.5
27	350	6	71	1.38	1.7
30	31870	15	1894	38.14	33.46
31	34	5	26	1.91	2.88

7. あとがき

本論文では,Exact Cover の列挙手法として 3 つの手法を挙げ, それらを実装し, 計算時間を比較した. その結果, 敷き詰め問題では AlgorithmX が勝り, 避難所割り当て問題では ZDD に基づいた 2 つの手法が勝った. 避難所割り当て問題で AlgorithmX がタイムアウトになった理由は, 入力が大きくなり, 解の個数が多くなったことが挙げられる.AlgorithmX は一つ一つ解を出力するため, 解の数が多いと時間がかかってしまう.

一方で,ZDD に基づいた 2 つの手法がこの結果になった理由としては, 敷き詰め問題では同型の部分グラフが少なかったため, ノードの共有があまり起きなかったと考えることができる. これは敷き詰め問題のピースは盤面のあらゆる場所に存在する可能性があり, 似たような列に 1 を持つ行が少なかったので, 同型の部分グラフも少なかったと考えられる. 逆に, 避難所割り当て問題は, 制約の大きさは違うが, 基本的にある避難所に割り当てされる領域は, その避難所から一定の距離内の地域だけである. なので似たような列に 1 をもつ行がたくさんあるため, 共有が上手くおきたと考えられる. また,easycube の問題では標準演算を用いた手法では解を列挙することができなかったが, フロンティア法では求めることができた. これは問題のサイズがもともと小さく,ZDD の削除ルールを適用しなくても ZDD を構築できたと考える. 標準演算を用いた方法は,ZDD の削除ルールを厳密に適用しようとしたため, 多くの組み合わせを計算し, 時間がかかってしまったのだと考えられる.

参考文献

[1] 藤澤 克樹, 梅谷 俊治, 応用に役立つ 50 の最適化問題, 朝倉書店, 2009.

[2] A. Takizawa, Y. Takechi, A. Ohta, N. Katoh, T. Inoue, T. Horiyama, J. Kawahara, S. Minato Enumeration of Region Partitioning for Evacuation Planning Based on ZDD, ISORA 2013, pp.64–71, 2013.

[3] 中山 弘隆, 谷野 哲三, 多目的最適化の理論と応用, コロナ社, 1994.

[4] D.E. Knuth, Dancing Links, Millenial Perspectives in Computer Science, pp. 187–214, 2000.

[5] 鈴木 拡, 湊 真一, BDD/ZDD を用いたペントミノパズルの解の列挙, 電子情報通信学会技術研究報告, COMP 109(54), 1–7, 2009.

[6] 川原 純, 斎藤 寿樹, 湊 真一, ZDD を用いた新たな列挙手法, 電子情報通信学会誌,95 巻, 6 号, pp. 505–511, 2012.

[7] 今井 浩, 今井 桂子, BDD による計算代数 · 計算幾何の不変量計算, 1041 巻, pp. 12–18, 1998.

[8] Shin-ichi Minato, Zero-suppressed BDDs and their applications, International Journal on STTT (2001) vol. 3, Issue 2, pp. 156—-170, 2001.