組合せ最適化問題の最適解を高速列挙するための ソフトウェアフレームワーク設計

小池　英勝[1]

要　旨

解法が確立していない問題を解くためのソフトウェア開発では，プログラムの仕様の変更が頻繁に起こることがある．本論文は，頻繁な仕様変更に対応しながら正しくて効率的なプログラムを開発するためのフレームワークを提案する．本研究の特徴は，プログラムの部品を書き換えルールで表現し，仕様から正しい書き換えルールを生成・集積してプログラムを構成することである．このことによってプログラムの効率と正しさの両立が可能になる．フレームワークの適用例としてコンテナ・プリマーシャリング問題の最適解を高速列挙するプログラムの開発プロセスを扱う．

キーワード：組合せ最適化，コンテナ・プリマーシャリング問題，列挙，ソフトウェアフレームワーク，ロジスティクス

1．はじめに

本研究の目的は，組合せ最適化問題の最適解を計算機で高速に列挙することである．組合せ最適化問題の最適解は複数存在することがあるが，問題のサイズや性質によっては，１つの解を得ることも困難な場合がある．既存の解法では，現状で最高性能の計算機を用いても一定のサイズ以上で最適解を得ることができない．そして，既存の解法の多くは，ヒューリスティックスや近似アルゴリズムを用いて計算量を抑えながら近似解を求めるので，最適解が求められないことがある．

個々の組合せ最適化問題は，より大きな問題の副問題になっていることがある．例えば，コンテナ・プリマーシャリング問題（Container Pre-Marshalling Problem，CPMP）は，コンテナ埠頭の敷地内の多数ある問題の１つである．コンテナ埠頭全体のパフォーマンスを最適化するには，複数の問題を統合的に扱う必要がある．近似解では，最適化の精度が落ちる．そして，一般に複数あるはずの解が１つしか求められないと局所最適化が起こり，全体の最適化が難しくなる．最適解が本質的に複数ある場合には，そこから１つだけ解を見つけて終わるのではなく，状況に応じて指定された個数の解を列挙したり，自明な形で解の集合を示したりする必要がある．

本論文は，実用的な規模の組合せ最適化問題を高速に解くためのフレームワークを構築する．現状は実装として未完成の部分があるが，このフレームワークは，仕様に関するプログラムの証明可能な正しさ，全ての解を列挙できる能力，そして，実用的な時間で解が得られる計算効率を兼ね備えるための論理的根拠を持っている．

以降，第２節で，本研究の背景と目的について述べる．第３節で，提案する問題解決のフレームワークの概要を説明する．第４節で，提案するフレームワークの適用例として扱う CPMP について述べる．第５節で，提案するフレームワークで CPMP の解を列挙するプログラムを生成する．６節で，まとめと今後の課題を述べる．

2．背景と目的

確立された解法がなく，新しい解法が次々と提案されるような研究分野で実験プログラムの開発が困難になる背景とそれに対する本研究の目的を述べる．

[1] 札幌学院大学経済学部；koike@sgu.ac.jp

2.1　背景

解法が確立していない問題を解くための理論を構築しようとするとき，実験用のプログラム開発には，一般的なプログラム開発と異なる以下のような難しさがある．

(1) 様々なアイディアを試すために変更が頻繁に起こる：これは，プログラムの仕様が変更されることである．仕様はプログラム開発において上流工程で作成されるので，その変更はコストが高い．

(2) プログラムの正しさを証明する必要がある：一般的なプログラムのテストよりも厳密さが求められるので，科学的なアプローチによる正しさの証明が必用な場合がある．

(3) リソースを効率的に利用して動作する必要がある：アルゴリズムの有効性を発揮させるには効率的な実装が必要である．もし，非効率な実装で実験すると，対象のアルゴリズムに対して評価を誤る可能性がある．

本論文で扱うプログラム開発は，既存の方法で解くことができない問題を新しい方法で解くために行うので，正しく，かつ，高速な動作が要求される．高速な動作には，計算リソースを適切に利用する技術を適用するが，その結果プログラムが複雑になるので正しさの証明が難しくなる．計算リソースを効率的に利用するために，実際に用いる計算機のスペックを考慮したプログラミングが必要になる．そのためには，計算機に搭載された装置（CPU，MB，主記憶装置，補助記憶装置，ネットワークカード，等）に関する知識と合わせて，そこで動作する OS とミドルウェアなどの関連するソフトウェアの知識が必要である．しかし，計算機もソフトウェアも陳腐化が激しいため，特定の計算リソースに特化しすぎると，これらが更新された場合に，既存の最適性が以降の改善を妨げる可能性がある．このため，想定するソフトウェア・ライフスパン中に見込まれる将来の計算リソースの改善を考慮した適切な最適化が必要である．

2.2　目的

本論文の目的は，頻繁に仕様の変更が起こる中で効率的でかつ仕様に関して正しいプログラムを開発するためのフレームワークを設計することである．このフレームワークは，上述のような特殊なケースだけではなく一般的なプログラム開発にも有用である．このフレームワーク開発の背景にある長期的な目標は，開発者が宣言的な仕様を記述するだけで，そこから効率的かつ正しいプログラムを自動生成することである．しかし，現時点での完全な実現は難しいので，以下で述べる具体的な目標に取り組む．

著者はこれまでに，組合せ最適化問題である CPMP を高速に解くための研究を行う中で実験プログラムを作成してきた．このプログラムの開発過程を当該フレームワークの視点から見直すことで，既存の実験プログラムのリファクタリングと機能の改良を効果的に行う．フレームワークという抽象的な構造と具体的な既存の CPMP のプログラム改善を同時に行いそれぞれの成果を反映することで，フレームワークは実用性を増し，CPMP のプログラムは改善されたものになることをねらう．

3．問題解決の流れとフレームワーク

本節では，提案するフレームワークを用いて組合せ最適化問題を解く際の手順について述べる．このフレームワークでは，問題解決を(1)形式的仕様の記述，(2)仕様変換，(3)手続き生成，(4)手続き最適化，(5)C++プログラム生成，そして，(6)プログラム実行という 6 つの工程で構成し，この順番で行う．ただし，必要に応じて以前の工程に移ることがある．以下で，各工程について説明する．

3.1　形式的仕様の記述

解こうとする問題の解集合を宣言的に記述する．この記述を形式的仕様とよぶ．以降，仕様は特に断りがない限り形式的仕様を意味する．仕様は，連立方程式と同様に，解集合を厳密に定義する．よって，仕様はプログラムの正しさのよりどころになる．仕様を記述する表現手段は，人間が記述したり意味を理解したりできると同時にコンピュータで扱いやすい必要がある．また，数式だけではなく，プログラムで解こうとする一般的な問題を記述できる必要がある．本研究では，仕様記述に第5.1節で示すような論理式を用いる．ただし，論理プログラムと異なり手続き的な意味は持たない．

仕様は，自然数と四則演算の関係などの一般的な公理や定理などの定義，解こうとする問題に関するルールなどの定義，そして，解こうとする具体的な問題の定義からなる．

仕様は定義部とクエリ部からなる．クエリ部には，解こうとする問題の具体例を記述する．それ以外は，定義部に記述する．この工程では，仕様の定義部を記述する．クエリ部は計算中に解が自明になるまで書き換えられるので計算中の状態を表す．

手続きは，後の工程で仕様から生成する．解を自明にするための（計算）手続きを書き換えルールで表現する．このときクエリ部を書き換えても，仕様が定義する解集合は変えないようにする．これは，連立方程式を解く際に解を変えずに式を変更するのと同じである．解集合を変えないルール集合のみを用いることで，プログラムの正当性を保証する．

3.2 仕様変換

効率的な手続きを生成するために仕様を改善する．仕様の改善の手法としては，計算パスを削減する新しい定義の導入や論理等価性に基づく仕様の変換などがある．これらの変換は，もとの仕様が表す解集合を変えない全ての変換の候補から，手続きの効率化に寄与するものを選んで適用できる．仕様変換は仕様の定義部の変換で，クエリ部の書き換えである計算とは異なる．

CPMP の問題解決における計算パス削減のための定義導入の例として，精度の高い下限（Bortfeldt *et al.*, 2012; Tanaka *et al.*, 2019）や無駄なコンテナの動きの除外（Tanaka *et al.*, 2019）がある．

3.3 手続き生成

仕様からクエリ部を変換する手続きを生成する．手続きは書き換えルールで表現する．書き換えルールの正しさは，仕様が表す解集合を変更しないことを示すことで証明できる．１つの書き換えルールの正しさは，他の書き換えルールの正しさと独立している．よって，プログラムを正しいルールのみで構成すると，各ルールの相互作用を考慮しなくてもプログラム全体の正しさが保証される．仕様に関して正しい手続きを生成する理論は文献（Akama *et al.*, 2006）にある．また，正しい書き換えルールを自動生成することができる（Koike *et al.*, 2001）．ただし，必要なルールの一部を自動生成できないことがある．また，自動生成されたルールよりも効率的なものが存在する可能性がある．ルールが不足していると計算が途中で止まるが，そこまでの計算の正しさは保証される．計算が止まっても，適切なルールを追加することで計算を再開することができる．追加されたルールが正しければ再開後の計算の正しさも保証される．

3.4 手続き最適化

書き換えルールは，上述した生成の過程である程度効率化されているが，より効率的なものに置き換えることができる可能性がある．また，手続きを並列実行させることで効率化できる場合がある（Koike *et al.*, 2012）．手続きを書き換えルールで記述しているため，１つの書き換えルールをプログラムコンポーネントとして扱える．書き換えルールをコンポーネントとみた場合，他の手続き型のコンポーネントと比較して粒度が細かい．書き換えルールの集合をプログラムとした場合，ピンポイントでコンポーネントの追加と削除が行えるしプログラムの正しさに影響しないので，最適化のための変更が低コストで行える．

3.5 C++プログラム生成

上の工程によって得られた個々の書き換えルールは手続きの各ステップを表しているので，それを手続き型プログラミング言語で記述することができる．本研究では，書き換えルール集合のプログラムから C++プログラムを生成する．C++を採用した理由は，書き換えルールを表現できるだけの高級言語としての機能をもちながら，効率化を突き詰めて行えるからである．C++プログラムを生成する際は実行環境を考慮した最適化を行う．実行環境には，計算機，OS，ミドルウェア，そして，ネットワークなどが含まれる．

3.6 プログラム実行

生成した C++プログラムから実行可能プログラムを生成し実行する．この工程は，一般的なプログラミング言語での開発と同様である．

4．コンテナ・プリマーシャリング問題

本論文が提案するフレームワークを用いたプログラム開発の具体例として，第 5 節で CPMP を扱う．そのため，本節で CPMP について説明する．

4.1 CPMP の概要

物流は経済活動の動脈ともいわれ，経済のグローバル化にともない，世界的にその重要性を増している．

物流の手段の 1 つとして海上コンテナ輸送がある．コンテナ輸送量は年々増加しており，近年，コンテナ埠頭の能力の限界を超える状況が世界中で起こっている．このような流通の混雑は，経済成長への悪影響や，輸送に携わる労働者の労働条件の悪化などを引き起こす．よって，早期に解決するべき世界的に重要な問題である．流通混雑問題の 1 つに CPMP がある．CPMP は，コンテナ埠頭などで起こるコンテナの並べ替え問題である．コンテナはトラックで陸送されて船積みされる前にコンテナ埠頭のヤードと呼ばれる敷地に一時的に積まれて待機する．この時点ではランダムに積まれるが，船積みの際は，コンテナの内容や行き先などから決まる優先度に従い積まれる．先に積むコンテナの上に後で積むコンテナがあると，まずそれを除けるための無駄な動きが必要になるので混雑の一因になる．船積みの時間を最適化する目的で，全てのコンテナの優先度が確定してから船積みが始まるまでの間に，事前に並べ替える問題が CPMP である．図 1 は，CPMP の一例を表す．図 1(a) はトラックからヤードにランダムに積まれたコンテナの集合を表し，Bay と呼ばれる．Bay はコンテナが縦に積まれた複数の Stack で構成される．Stack はコンテナが積める個数に制限がある．図 1 は，コンテナが最大 4 つ積める 6 つの Stack で構成されている Bay を表す．この Bay のサイズを 6×4 と表す．図中の四角はコンテナを表しその番号は船に積む順番を表す（0 が最も早く船積みされる）．CPMP は図 1(a) の Bay を図 1(b) のような，後で積むコンテナが先に積むコンテナの上にない状態に並べ替える最短の手順の求める問題である．解は一般に複数あり，その解の 1 つは (1, 0) (1, 3) (0, 1) (4, 3) (4, 1) (2, 4) (2, 1) (0, 1) (3, 4) である．この列は，9 回のコンテナの移動を表す．例えば，先頭の (1, 0) は図 1 の stack 1の上のコンテナを stack 0 に移すことを表す．CPMP は一見簡単な問題に見えるが，NP 困難であることが示されている．

4.2　CPMP の特性

CPMP の特性として，本論文で注目するものは以下の通りである．

(1) 1 つの解の候補の生成を低コストで行える．
(2) 1 つの解の候補の評価を低コストで行える．
(3) 一定以上の問題のサイズでは，解の候補の数が現状の計算機では扱えないほど多くなることがある．

特性(3)が実用的なサイズの問題を解くことを困難にしている．

ヒューリスティックアプローチでは，計算量の削減のため，計算パスの枝狩りを大胆に行う．これは，まだ解法が確立されていない問題で，とにかく何らかの答えを求めることが重要な場合に効果的である．しかし，最適解を得られる保証がなくなるだけでなく，特定の問題では解が得られないことがある．このことは実用的にも問題になりうる．

本研究では，計算量を削減する方法として，最適性を損なわないもののみを採用する．言い換えると，削減する解の候補のなかに最適解が含まれないことを保証する．著者はこれまでに CPMP において解の最適性を保証しながら効率的な計算ができることを示した（小池，2017；Koike, 2017）．本研究で開発するプログラムは，CPMP の最適解を全て求める機能を持つ．それと同時に，指定された個数だけ解を求める機能と，指定された近似比の解を求める機能を持つ．

4.3　前提条件

CPMP はコンテナの並べ替え問題であるが，類似の問題，例えば，コンテナ・リロケーション問題（Container Relocation Problem, CRP; Kim & Hong, 2006）などと明確に区別するために前提条件を以下に

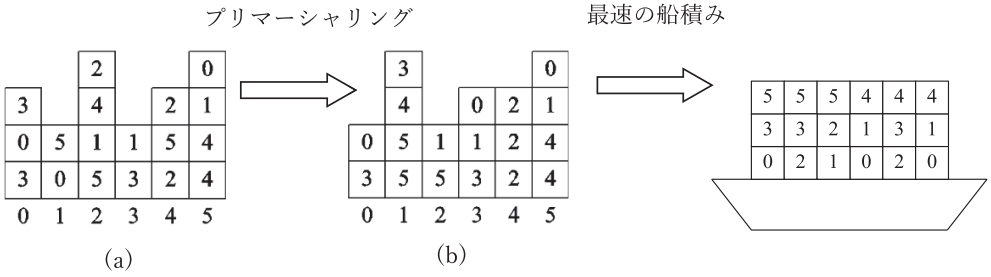

図 1　コンテナ・プリマーシャリング

明示する.

(1) コンテナは，同一の Bay 上でのみ移動可能である．コンテナを並べ替えながら船積みする CRP と異なり，並べ替え中に Bay が持つコンテナの総数は変わらない.

(2) 全てのコンテナは同じサイズである.

(3) 全てのコンテナに優先順位を示す番号が与えられている.

(4) 異なるコンテナが同じ優先順位を持つことがある.

(5) コンテナのスタック間の移動距離の差によるコストの差は無視できるほど小さいとする.

5．CPMP を解くプログラムの自動生成

本節では，提案するフレームワークを用いて CPMP の解を列挙するプログラムを生成する.

5.1 仕様記述の例

図 2 は，CPMP の仕様の例である.

この仕様は，説明のために簡単化したものだが，実際に解を求めるプログラムが生成できるという意味で完全な仕様である．本論文で扱う仕様は S 式で記述された確定節集合である．改行は見やすさ以外の意味を持たない．表現に S 式を用いることと改行の扱いは，後述の書き換えルールでも同様である.

例えば，図 2 の節の 1 つである

(as(decideOneMove *Bay *Move)：

(decideOneMove( )*Bay *Move))

```
(as (premarshalling *InitialBay *InitialBay ()) :)
(as (premarshalling *InitialBay *OptimizedBay (*Move | *Moves2)) : (move *InitialBay *NewBay
*Move)(premarshalling *NewBay *OptimizedBay *Moves2))
(as (move *InitialBay *NewBay *Move) :(decideOneMove *InitialBay *Move) (apply *Move *InitialBay
*NewBay))
(as (decideOneMove *Bay *Move) : (decideOneMove () *Bay *Move))
(as (decideOneMove *Stacks0 (*Stack1 | *Stacks) (*Top *Sn1 *Sn2)) : (nonempty *Stack1) (top *Stack1
*Top) (getStackNumber *Stack1 *Sn1) (append *Stacks0 *Stacks *RestStacks) (decideToWhere
*RestStacks *Sn2))
(as (decideOneMove *Stacks0 (*Stack1 *Stack2 | *Stacks) *Move) : (append *Stacks0 (*Stack1)
*Stacks1) (decideOneMove *Stacks1 (*Stack2 | *Stacks) *Move))
(as (decideToWhere (*Stack | *Stacks) *Sn) : (hasSpace *Stack) (getStackNumber *Stack *Sn))
(as (decideToWhere (*Stack | *Stacks) *Sn) : (decideToWhere *Stacks *Sn))
(as (nonempty (*N (*E | *List))):)
(as (top (*N (*E | *List)) *E):)
(as (getStackNumber (*N *List) *N):)
(as (apply (*Top *Sn1 *Sn2) *Bay1 *Bay2) : (removeTop *Sn1 *Bay1 *TempBay) (push *Top *Sn2 *TempBay
*Bay2))
(as (removeTop *N ((*N *List) | *Stacks) (*NewStack | *Stacks)) : (pop (*N *List) *NewStack))
(as (removeTop *N ((*N1 *List) | *Stacks) ((*N1 *List) | *NewStacks)) : (not_eq *N *N1)(removeTop
*N *Stacks *NewStacks))
(as (push *Top *N ((*N *List) | *Stacks) (*NewStack | *Stacks)) : (push_s *Top (*N *List) *NewStack))
(as (push *Top *N ((*N1 *List) | *Stacks) ((*N1 *List) |*NewStacks)) : (not_eq *N *N1)(push *Top
*N *Stacks *NewStacks))
(as (pop (*N (*L | *List)) (*N *List)) :)
(as (push_s *Top (*N *List) (*N *NewList)) :(append (*Top) *List *NewList))
(as (append () *x *x))
(as (append (*a|*x) *y (*a|*z)):(append *x *y *z))
(as (hasSpace (*N *List)) : (length *List *Len) (< *Len 4))
(as (length () 0):)
(as (length (*E | *List) *Len) : (length *List *Len2) (add *Len2 1 *Len))
(as (pm *OptimizedBay *Moves) : (premarshalling
((1 (3 2))
 (2 (5 4 1))
 (3 (7 6))) *OptimizedBay *Moves)
 (equal *OptimizedBay ((1 (4 5 7))(2 (1) )(3 (2 3 6))) )
 (equal *Moves ((? ? ?)(? ? ?)(? ? ?)(? ? ?)(? ? ?)(? ? ?))))
```

図 2　CPMP の形式的仕様

は，確定節

decideOneMove（BAY, MOVE)←decideOneMove（[ ], BAY, MOVE).

を表す．確定節集合は問題を宣言的に定義したものなので，論理プログラムと異なり手続きやプログラムとしての意味を持たない．よって，節やアトムの順序に意味は無いし，カットのような制御の記述は含まない．

本論で扱う仕様の文法の説明をする．１つの節は，(as〈アトム〉:〈アトムの集合〉) という形式で表現される．節の先頭の as は assert を意味する．〈アトム〉はシンボルで始まり，そのあとに 0 個以上の項が続くリストである．〈アトム〉に続く：は，論理式の←を表す．項は空白で区切られる．ただし，(,)，*，|, ：のような定義済みシンボルは空白が無くても区切られる．論理変数を*で始まるシンボルで表す．無名論理変数を？で表す．リストを表す項（*X | *Y）について，*X は，先頭の要素，*Y は*X より後ろの全ての項（を要素として持つリスト）をそれぞれ表す．( )は空リストを表す．例えば，３つの異なる変数を要素に持つリスト（*A *B *C）は，（*A | (*B *C)）や（*A *B *C | ( )）とも表せる．〈アトムの集合〉は 0 個以上の〈アトム〉からなる列である．無名論理変数について，(? ? ?) は (*A *B *C) と同じである．

図２の仕様では，最後の節以外が問題の一般的な定義で，最後の節が具体的な問題の定義である．最後の節は，図３の解くべき Bay の初期状態と図 4(a) の並べ替えが完了した状態をそれぞれ，以下の項で表している．

解くべき Bay：((1 (3 2)) (2 (5 4 1)) (3 (7 6)))

完了した状態：((1 (4 5 7)) (2 (1)) (3 (2 3 6)))

上の Bay を表す項について説明する．これらの項はスタックの集合を表す．スタックは

（スタック番号（〈コンテナ番号列〉））

のような構造を持つ．ここで，〈コンテナ番号列〉は，コンテナの優先度を表す整数が 0 個以上続く数列を表す．最後のアトム

(equal *Moves ((? ? ?) (? ? ?) (? ? ?) (? ? ?) (? ? ?) (? ? ?)))

は，求めるコンテナの移動が 6 回であることを要求している．リスト (? ? ?) が 1 回のコンテナの移動を表す．１番目の要素は，移動するコンテナの番号を表す．２番目の要素は，移動元のスタック番号を表す．３番目の要素は，移動先のスタック番号を表す．１番目の要素は，冗長な情報だが，人間が実行結果を確認しやすくするためにつけている．この時点でのリストは３つの無名論理変数からなる．後の工程でプログラム実行時に解が見つかると，これらの変数に具体的な値が代入される．

以上から，最後の節の全体の意味をまとめると，「Bay の初期状態からコンテナを 6 回移動して並べ替えが完了した状態が第一引数であり，その正しい手順が第二引数である」となる．本来，最適解の移動回数は計算前にはわからない．６回という具体的な移動の回数はあらかじめ解いた結果を用いている．Bay の最終状態も同様である．この例題はサイズが小さいため，C++で全ての計算パスを列挙する単純なプログラムを作成して求めた．原理的には，具体的な値を入れなくてもよい．解の長さと Bay の最終状態を与えた理由は，ルール生成の計算時間を短縮するためである．具体的な例を与えてプログラムを生成しても，そのプログラムは他の異なる問題も解くことができる．

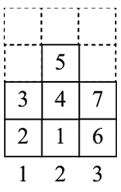

図３　Bay の初期状態

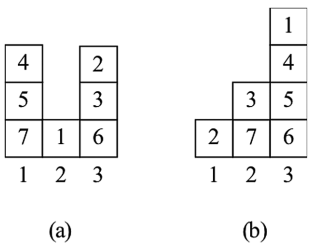

図４　コンテナを並べ替えた結果

5.2　仕様変換

元の仕様に変更を加えることで，より効率的なプログラムが生成できる場合がある．例えば解の長さの下限（Lower Bound, LB）を用いた探索空間の削減がある．この場合，仕様の３番目の確定節を

(as (move *InitialBay *NewBay (*Move | *Moves)) :
(decideOneMove *InitialBay *Move) (apply *Move *InitialBay *NewBay) (length *Moves *numMoves)
(lowerBound *NewBay *LB) (<= *LB *numMoves))

に置き換える．ここで，lowerBound/2 は，第一引数が Bay で第二引数はその LB を表す．lowerBound/2 を定義する節も追加する必要があるがここでは省略する．この lowerBound/2 で得られる LB が正確であればあるほど探索空間を削減することができる．精度の高い LB の計算方法は，例えば，文献（Tanaka et al., 2019）にある．

5.3 手続き生成

仕様から手続きを生成する．手続きは書き換えルールの集合で表現する．仕様から書き換えルールの集合を自動生成する方法が Koike *et al.*（2001）などで提案されている．図５と図６は，図２の仕様から自動生成した書き換えルールの集合である．自動生成は，Koike *et al.*（2001）の方法に基づく実験システムを用いて行った．このルールの集合は，言語処理系 ETI（Koike *et al*, 2005）で動作するプログラムでもある．よってこの工程が終了した時点で，プログラムの実行が可能である．理論的には，仕様Ｓは定義部Ｄとクエリ部Ｑで構成される（Ｓ＝Ｄ∪Ｑ）．Ｑは，実行時に入力された具体的な質問から作られる．自動生成された各書き換えルールはＱをＱ’に書き換えたとき，その宣言的意味を保存する（Ｍ(Ｄ∪Ｑ)＝Ｍ(Ｄ∪Ｑ’)）．ここで，Ｍ(Ｓ) は仕様Ｓによって決まる解集合である．ルールの正しさに関する詳細は Akama & Nantajeewarawat（2006）にある．

図７は，図５と図６のプログラムを実行した結果である．ユーザが具体的な質問として(pm？?)と入力すると，クエリ部に節（(ans(pm *A *B))←(pm *A *B)）が作られる．この節のボディ（節の←より右側のアトム集合）が，適用可能なルールで０回以上書き換えられる．ルールによって定義される動作には，例えば，

```
(Rule ruleName
(Head (pm *K12 *L12))
(Body (exec (= *K12 ((1 *M12) (2 *N12) (3 *O12))) (= *L12 ((*P12 *Q12 *R12) (*S12 *T12 *U12) (*V12 *W12 *X12)
(*Y12 *A13 *B13) (*C13 *D13 *E13) (*F13 *G13 *H13)))) (premarshalling ((1 (3 2)) (2 (5 4 1)) (3 (7 6))) *K12
*L12)))

(Rule ruleName
(Head (premarshalling *A11 *B11 (*C11 | *D11)))
(Body (move *A11 *E11 (*C11 | *D11)) (premarshalling *E11 *B11 *D11)))

(Rule ruleName
(Head (premarshalling *S1 *T1 ())) (Body (exec (= *T1 *S1)) (optimized *S1)))

(Rule ruleName
(Head (optimized (*Q1 | *R1))) (Body (optimizedStack *Q1) (optimized *R1)))

(Rule ruleName (Head (optimized ())) (Body))

(Rule ruleName (Head (optimizedStack (*I1 *J1))) (Body (ascendingOrder *J1)))

(Rule ruleName
(Head (move *D13 *E13 (*F13)))
(Body (decideOneMove *D13 *F13) (apply *F13 *D13 *E13) (numBadlyPlacedContainers *E13 *G13) (collectNBSs
*E13 *H13) (minimize *H13 *I13) (add *G13 *I13 *J13) (<= *J13 0)))

(Rule ruleName (Head (decideOneMove *X7 *Y7)) (Body (decideOneMove () *X7 *Y7)))

(Rule ruleName2
(Head (decideOneMove () *G64 (*H64 *I64 *J64)))
(Body (exec (= *G64 ((*K64 (*H64 | *L64)) | *M64))) (getStackNumber (*K64 (*H64 | *L64)) *I64) (append ()
*M64 *N64) (decideToWhere *N64 *J64))
(Body (exec (= *G64 (*O64 *P64 | *Q64))) (decideOneMove (*O64) (*P64 | *Q64) (*H64 *I64 *J64))))

(Rule ruleName (Head (getStackNumber (*X *Y) *A1)) (Body (exec (= *A1 *X))))

(Rule ruleName
(Head (numBadlyPlacedContainers (*J3 | *K3) *L3))
(Body (numBadlyPlacedContainersInStack *J3 *M3) (numBadlyPlacedContainers *K3 *N3) (add *M3 *N3 *L3)))

(Rule ruleName (Head (numBadlyPlacedContainers () *F)) (Body (exec (= *F 0))))

(Rule ruleName
(Head (numBadlyPlacedContainersInStack (*W8 *X8) *Y8))
(Body (reverse *X8 *A9) (revNumBadlyPlacedContainersInStack *A9 *Y8)))
```

図５　生成された手続きを表すルール集合（1/2）

```
(Rule ruleName (Head (collectNBSs (*J3 | *K3) *L3))(Body (exec (= *L3 (*M3 | *N3)))
(numBadlyPlacedContainersInStack *J3 *M3) (collectNBSs *K3 *N3)))
(Rule ruleName (Head (collectNBSs () *F))(Body (exec (= *F ()))))
(Rule ruleName (Head (move *M25 *N25 (*O25 | *P25)))(Body (decideOneMove *M25 *O25) (apply *O25 *M25 *N25)
(length *P25 *Q25) (numBadlyPlacedContainers *N25 *R25) (collectNBSs *N25 *S25) (minimize *S25 *T25) (add
*R25 *T25 *U25) (<= *U25 *Q25)))
(Rule ruleName2 (Head (decideOneMove *M59 ((*N59 (*O59 | *P59)) *Q59 | *R59) (*S59 *T59 *U59)))(Body (exec
(= *O59 *S59)) (getStackNumber (*N59 (*O59 | *P59)) *T59) (append *M59 (*Q59 | *R59) *V59) (decideToWhere
*V59 *U59))(Body (append *M59 ((*N59 (*O59 | *P59))) *W59) (decideOneMove *W59 (*Q59 | *R59) (*S59 *T59
*U59))))
(Rule ruleName (Head (apply (*O5 *P5 *Q5) ((*P5 (*R5 | *S5)) | *T5) *U5))(Body (push *O5 *Q5 ((*P5 *S5)
| *T5) *U5)))
(Rule ruleName (Head (apply (*J5 *K5 *L5) *M5 *N5)) (Body (removeTop *K5 *M5 *O5) (push *J5 *L5 *O5 *N5)))
(Rule ruleName (Head (removeTop *D1 ((*D1 (*E1 | *F1)) | *G1) *H1))(Body (exec (= *H1 ((*D1 *F1) | *G1)))))
(Rule ruleName (Head (removeTop *V ((*W *X) | *Y) *A1))(Cond (not (var *W)) (not (var *V)) (/== *V *W))
(Body (exec (= *A1 ((*W *X) | *B1))) (removeTop *V *Y *B1)))
(Rule ruleName2 (Head (decideToWhere ((*V1 *W1) | *X1) *Y1))(Body (exec (= *Y1 *V1)) (hasSpace (*V1
*W1)))(Body (decideToWhere *X1 *Y1)))
(Rule ruleName (Head (hasSpace (*U *V))) (Body (length *V *W) (> 4 *W)))
(Rule ruleName (Head (push *A1 *B1 ((*C1 *D1) | *E1) *F1)) (Cond (not (var *C1)) (not (var *B1)) (/== *B1
*C1))(Body (exec (= *F1 ((*C1 *D1) | *G1))) (push *A1 *B1 *E1 *G1)))
(Rule ruleName (Head (push *P1 *Q1 ((*Q1 *R1) | *S1) *T1)) (Body (exec (= *T1 (*U1 | *S1))) (push_s *P1
(*Q1 *R1) *U1)))
(Rule ruleName (Head (push_s *D1 (*E1 *F1) *G1))(Body (exec (= *G1 (*E1 *H1))) (append (*D1) *F1 *H1)))
(Rule ruleName (Head (append (*S | *T) *U *V)) (Body (exec (= *V (*S | *W))) (append *T *U *W)))
(Rule ruleName(Head (decideOneMove *R27 ((*S27 (*T27 | *U27))) (*V27 *W27 *X27)))(Body (exec (= *T27 *V27))
(getStackNumber (*S27 (*T27 | *U27)) *W27) (append *R27 () *Y27) (decideToWhere *Y27 *X27)))
(Rule ruleName (Head (reverse (*B1 | *C1) *D1))(Body (reverse *C1 *E1) (append *E1 (*B1) *D1)))
(Rule ruleName (Head (length (*O | *P) *Q))(Body (length *P *R) (add *R 1 *Q)))
(Rule ruleName (Head (length () *F)) (Body (exec (= *F 0))))
(Rule ruleName (Head (reverse () *F)) (Body (exec (= *F ()))))
(Rule ruleName (Head (revNumBadlyPlacedContainersInStack (*Y *A1 | *B1) *C1)) (Cond (number *A1) (number
*Y) (> *A1 *Y))(Body (length (*A1 | *B1) *C1)))
(Rule ruleName(Head (decideToWhere () *F))(Body (exec (false))))
(Rule ruleName (Head (revNumBadlyPlacedContainersInStack (*I) *J))(Body (exec (= *J 0))))
(Rule ruleName (Head (revNumBadlyPlacedContainersInStack () *F))(Body (exec (= *F 0))))
(Rule ruleName (Head (ascendingOrder (*Y *A1 | *B1)))(Cond (number *A1) (number *Y) (<= *Y *A1))(Body
(ascendingOrder (*A1 | *B1))))
(Rule ruleName (Head (ascendingOrder (*K)))(Body))
(Rule ruleName (Head (ascendingOrder (*Q *R | *S)))(Cond (number *R) (number *Q) (> *Q *R))(Body (exec
(false))))
(Rule ruleName(Head (revNumBadlyPlacedContainersInStack (*O *P | *Q) *R))(Cond (number *P) (number *O)(<=
*P *O))(Body (revNumBadlyPlacedContainersInStack (*P | *Q) *R)))
(Rule ruleName(Head (decideOneMove *I52 ((*J52 ()) *K52 | *L52) (*M52 *N52 *O52)))(Body (append *I52 ((*J52
())) *P52) (decideOneMove *P52 (*K52 | *L52) (*M52 *N52 *O52))))
```

図６　生成された手続きを表すルール集合 (2/2)

変数の代入，アトムの消去・書き換え，そして，１つの節から異なる０個以上の節への書き換えがある．書き換えが進み，どのルールも適用できなくなったら計算は停止する．このとき，プログラムが適切に記述されていればクエリ部は解を自明に表している．図７は実際の実行画面で，クエリ部に図８が示す２つの節が残ったことを表す．この節集合は，コンテナの移動の手順として，

(a) (7 3 2) (3 1 3) (2 1 3) (7 2 1) (5 2 1) (4 2 1)

(b) (7 3 1) (5 2 3) (4 2 3) (1 2 3) (7 1 2) (3 1 2)

という２つの解が得られ，かつ，それ以外の解がないことを表している．また，各節のアトム pm/２の第１引数は得られた手順によってコンテナを並べ替えた後の Bay の状態を表している．上の手順 (a) から，図４(a) の Bay が，手順 (b) から図４(b) の Bay がそれぞれ得られた．

プログラムを構成する書き換えルールについて説明する．詳細は Koike *et al.* (2005) にある．ルールは以下のような構文を持つ：

〈Rule〈ルール名〉(Head〈アトム列〉)(Cond〈アトム列〉)(Body (exec〈アトム列〉)〈アトム列〉))

〈ルール名〉は任意のシンボルである．ルール名を

```
[D]>load "rules/petr.etc"

[D]>n
Execution mode "N"

[N]>(pm ? ?)
------------------------N execution --------------------
------------------------------------------------------------
(ans (pm ((1 (4 5 7)) (2 (1)) (3 (2 3 6))) ((7 3 2) (3 1 3) (2 1 3) (7 2 1) (5 2
 1) (4 2 1))))<-.
(ans (pm ((1 (2)) (2 (3 7)) (3 (1 4 5 6))) ((7 3 1) (5 2 3) (4 2 3) (1 2 3) (7 1
 2) (3 1 2))))<-.

execution time: 168831 [msec]

[N]>
```

図 7　実行例

```
(ans (pm ((1 (4 5 7)) (2 (1)) (3 (2 3 6))) ((7 3 2) (3 1 3) (2 1 3) (7 2 1) (5 2
1) (4 2 1))))<-.
(ans (pm ((1 (2)) (2 (3 7)) (3 (1 4 5 6))) ((7 3 1) (5 2 3) (4 2 3) (1 2 3) (7 1
2) (3 1 2))))<-.
```

図 8　計算結果

用いて同時に適用可能なルールに対して優先順位を設定できる．〈ルール名〉の後に Head 部が続く．Head 部は 1 つ以上のアトムを持つ．Head 部の後に Cond 部が続く．Cond は Condition を意味し，Head だけでは表現できないルールの適用条件を記述する．Cond 部は省略できる．Cond 部の後に Body 部が続く．ルールは Body 部を 1 つ以上持つ．ルールの適用時に Body 部の数だけ節が複製され，それぞれの Body に従って節の書き換えが起こる．これによって複数の解がある場合に対応できるので解の完全性を保証できる．Body 部は exec 部と置き換え部からなる．exec は execution を意味する．exec 部のアトム列は組み込み述語やユーザ定義述語の実行を記述できる．exec 部に記述する例として代入がある．exec 部の実行が失敗すると変換中の節は削除される．exec 部は省略できる．exec 部の後に続くアトム列が置き換え部である．置き換え部は 0 個以上のアトム列で，ヘッド部にマッチした節のボディアトムと置き換わる．

書き換えルールは，クエリ部の節に適用され，対象の節を新しい（0 個以上の節からなる）節集合に置き換える．書き換えルールは，クエリ部の節のボディアトムと Head 部がマッチして，かつ，Cond 部の条件が満たされたとき適用可能である．書き換えルールが複数同時に適用可能な場合，どのルールから適用しても計算の正しさは保証される．ただし，適用順序の違いは計算効率の差となることがある．

5.4　手続き最適化

上記のルールベースのプログラムは C++などの手続き型言語よりも抽象的で，プログラムの大域的な最適化が容易な場合がある．自動生成されたルールは個々の正当性が証明されているし，ルールの追加や削除は計算の正当性に影響しない．よって，高速化のためにより効率的なルールを追加したり，ボトルネックとなっていたルール削除したりすることが低コストで行える．このルールベースのプログラム開発は，プログラムのどこで計算量が増えるかをルールの Body 部の数で特定することができる．一般的に，Body 部の数が多いほど計算量を増大させる可能性が高くなる．よって，複数の Body 部を持つルールを，節数を増大させないように変更できれば計算効率を改善できる．改善できるケースとして仕様の変更による改善があり，上述の LB の導入がこれにあたる．仕様を変更したら，手続き生成の工程に戻りプログラムを再び自動生成することで，仕様に関して正しいプログラムを得ることができる．別のケースとして，ルール自体の変更がある．この場合はルールの正当性はプログラマが責任を持つことになる．理論的には変更により追加されたルールが $M(D \cup Q) = M(D \cup Q')$ を満たすことを証明できれば仕様に関して正しいことを保証できる．

しかし，正しさの証明は自明な場合がある一方で，難しい場合がある．正しさの証明が難しい場合は，一般的なプログラミングのテスト手法を用いて正しさの検証を行う．

5.5　C++プログラム生成

手続き生成の工程で得られたルールベースのプログラムは，言語処理系 ETI により実行可能である．ETI は，ハイレベルプログラミング言語としての抽象性や実行時中にプログラムを変更できるなど柔軟な機能を持つ．しかし，実行効率は C++と比較すると良くない．ルールベースのプログラムは手続きを記述しているので，ある程度機械的な作業で C++などの手続き型プログラミング言語へ変換できる．ETI 自体も C++で記述されている．計算状態を表現するクエリ部の節集合と個々の書き換えルールを C++に変換できる．適用するルールの選択を行うコードも ETI から流用できる．ETI のデータ構造は安定性と汎用性を重視して設計されている．そのため，問題に特化したデータ構造を用いることで効率が改善することがある．

作成するプログラムが CPU の性能を使い切るためにはマルチスレッドで効率的な動作をする必要がある．メモリは CPU に対して遅く，不用意なメモリアクセスが計算効率低下の原因になる場合がある．よって，CPU の性能を発揮できるように効率的なメモリアクセスを行うためのプログラム設計が必要である．この工程では，プログラミング言語，利用するハードウェア，OS，そして，コンパイラなどの特性も考慮してコーディングする．

この方法により，C++で記述したプログラム（Koike, 2017）で例題を解くと，図 5 と図 6 で示したルールベースのプログラムと比較して約 1 万倍高速だった．ただし，C++のプログラムは，その時点で有効と考えられた最大限の最適化を行っている．たとえば，仕様には，マルチセットを用いた高速化（Koike, 2017）に加え，コンテキストを考慮した LB の計算方法や，コンテナの移動の制限など，計算量の削減のための様々なテクニックの定義を導入している．

5.6　プログラム実行

C++プログラムをコンパイルし実行可能ファイルを生成する．そして，実行可能ファイルを実行する．このとき，目的の問題の解を設定した時間内に得られれば成功である．しかし，解がいつまでも得られない場合がある．実行時にプロファイラなどを利用することで性能が上がる場合があるが，多くの場合，解けなかった問題が解けるようになるほどの改善はない．また，目標を達成しても，新しい目標として問題のサイズを 1 段大きくしただけで，全く解が得られなくなる場合がある．そのような場合は，仕様記述の工程に戻ってさらなる高速化を目指す．

6．まとめと今後の課題

組合せ最適化問題の 1 つである CPMP のような解法が確立していない問題を計算機で高速に解こうとすると，開発する実験プログラムを高度に最適化しながら変更に強い構造も維持しなければならない．本論文では，この難しい問題を，仕様から正しい手続きを自動生成するフレームワークを適用することで解決しようとした．このフレームワークでは，仕様に関して正しい手続きをルール集合として得ることができる．そのルール集合をガイドとして，大域的な最適化が可能になる．より効率的なルールを生成するという観点から，仕様を改善することでプログラムの正しさを維持しながらより高度な高速化が実現できる可能性がある．

今後の課題を以下に列挙する．

- 仕様の表現力の改善：現在のフレームワークでは確定節のみを入力できるが，関連研究でこのフレームワークが，より表現力の高い論理表現を扱えることが示されている．よって，本研究のフレームワーク上で仕様の表現力を高めることができる．結果として，より複雑な問題への取り組みが可能になる．
- 書き換えルールの表現力の改善：書き換えルールの表現力を拡張することで，より効率的な手続きを記述できるようにする．これには，データ構造や計算状態の表現力の改善も含まれる．
- 書き換えルールの処理系の改善：現状の書き換えルール実行用言語処理系は，安定性重視の設計であるため，大幅な効率改善の余地がある．ルールベースのプログラムを効率的に実行することは，本フレームワークでのプログラム開発の効率化につながる．
- 書き換えルールからの C++プログラム自動生成：現在手作業で行っている C++プログラムの記述を

できるだけ自動化する．そのことによって，人的エラー混入の可能性を減少させながら開発の効率化を実現する．

特に，書き換えルールからのC++プログラムの自動生成は，優先的に実現したい．

謝辞

本研究は，2018年度札幌学院大学研究奨励金（C）SGU-CS2018-01の助成を受けた．

参考文献

[1] Akama, K., Nantajeewarawat, E. and Koike, H. (2006). Program Generation in the Equivalent Transformation Computation Model using the Squeeze Method, Perspectives of System Informatics, Lecture Notes in Computer Science, 4378, Springer Verlag, Heidelberg, 41–54, DOI: 10.1007/978-3-540-70881-0_7.

[2] Akama, K. and Nantajeewarawat, E. (2006). Formalization of the Equivalent Transformation Computation Models, Journal of Advanced Computational Intelligence and Intelligent Informatics 10, 245–259, DOI: 10.20965/jaciii.2006.p0245.

[3] Bortfeldt, A. and Forster, F. (2012). A Tree Search Procedure for The Container Pre-Marshalling Problem, European Journal of Operational Research, 217(3), 531–540, DOI: 10.1016/j.ejor.2011.10.005.

[4] Kim, K. H. and Hong, G. P. (2006). A heuristic rule for relocating blocks, Computers & Operations Research, 33, 940–954, DOI: 10.1016/j.cor.2004.08.005.

[5] Koike, H., Akama, K. and Miura, K. (2012). Generation Method for Correct Parallel Programs Based on Equivalent Transformation, Proc. of The 2nd International Conference on Information and Communication Technologies and Applications (ICTA 2012), 116–121.

[6] 小池英勝（2017）. コンテナプリマーシャリング問題における探索空間の削減, 札幌学院大学総合研究所紀要, 4, 9–15.

[7] Koike, H. (2017). Search Space Reduction with Multiset for Effectively Solving the Container Pre-Marshalling Problem, 2017 International Conference on Mathematical Methods & Computational Techniques in Science & Engineering (AIP Conference Proceedings), 1872(1), 020026, DOI: 10.1063/1.4996683.

[8] Koike, H., Akama, K. and Mabuchi, H. (2005). A Programming Language Interpreter System Based on Equivalent Transformation, In 2005 IEEE 9th International Conference on Intelligent Engineering Systems (INES 2005), 283–288, DOI: 10.1109/INES.2005.1555174.

[9] Koike, H., Akama, K., and Boyd, E. (2001). Program Synthesis by Generating Equivalent Transformation Rules, in Proceedings of the Second International Conference on Intelligent Technologies (InTech'01), Bangkok, Thailand, 250–259.

[10] Tanaka, S., Tierney, K., Parreño-Torres, C., Alvarez-Valdes, R., and Ruiz, R. (2019). A branch and bound approach for large pre-marshalling problems, European Journal of Operational Research, 278(1), 211–225, DOI: 10.1016/j.ejor.2019.04.005.

The Proceedings of the Research Institute of Sapporo Gakuin University Vol.8, 67–78 (2021)

Software Framework Design for Fast Enumeration of Optimal Solutions to Combinatorial Optimization Problems

Hidekatsu KOIKE[1]

Abstract

Frequent revision of program specifications can occur in software development to solve problems of which well-established solutions have yet to be found. This paper proposes a framework for software development to efficiently enumerate optimal solutions to combinatorial optimization problems; the framework copes with the frequent specification changes. A peculiarity of this study is that we represent each program component as a correct rewriting rule, which is generated from a specification and accumulates in a program. The peculiarity enables us to manage both efficiency and correctness of a program. This paper demonstrates a software development process to efficiently enumerate optimal solutions to a container pre-marshalling problem as an application example of the framework.

Keywords: Combinatorial Optimization, Container Pre-Marshalling Problem, Enumeration, Logistics, Software Framework.

[1] Department of Economics, Sapporo Gakuin University; koike@sgu.ac.jp.