解集合プログラミングに基づく組合せ遷移ソルバーの実装方式に関する考察

山田 悠也　湊 真一　番原 睦則

組合せ遷移問題とは，基となる組合せ問題とその 2 つの実行可能解が与えられたとき，一方の実行可能解から他方の実行可能解へ，遷移制約を満たしつつ，実行可能解のみを経由して到達できるかを判定する問題である．計算困難性について，基の組合せ問題が NP 完全であるとき，その遷移問題の多くは PSPACE 完全になることが知られている．本稿では，解集合プログラミング (ASP) を用いた組合せ遷移問題の解法として，有界組合せ遷移を提案し，そのソルバーの実装方法について述べる．有界組合せ遷移は，組合せ遷移問題に対して，制限された長さの (すなわち有界の) 遷移系列を論理プログラムとして記号的に表現し，その論理プログラムを ASP システムで実行することにより，到達可能性の検査を行う．有界組合せ遷移の有効性を評価するために，代表的な組合せ遷移問題の 1 つである k 彩色遷移問題をベンチマークとして，提案ソルバーの性能評価を行った．

1　はじめに

組合せ遷移問題 (Combinatorial Reconfiguration Problems; CoRe [12] [14] [16]) とは，基となる組合せ問題とその 2 つの実行可能解が与えられたとき，一方の実行可能解から他方の実行可能解へ，遷移制約を満たしつつ，実行可能解のみを経由して到達できるかどうかを判定する問題である．

計算困難性について，基の組合せ問題が NP 完全であるとき，その遷移問題の多くは PSPACE 完全になることが知られている．代表的なものとしては，命題論理の充足可能性判定問題 (SAT)，集合被覆問題，独立点集合，グラフ点彩色問題を基とする組合せ遷移問題などがある [5] [6] [10] [12].

組合せ遷移問題の研究は，理論計算機科学の分野を中心に，最近 10 年で急速に発展し，理論的な基盤が整備されつつある．また，実社会においても，配電網の切替えなど持続的システムへの実用的応用が期待されている．しかしながら，一般の組合せ遷移問題を解く汎用ソルバーのアルゴリズム，ヒューリスティックスなどの実践的な研究開発はまだ始まったばかりである．著者らの知る限り，実用的な汎用ソルバーはまだ存在していない．

解集合プログラミング (Answer Set Programming; ASP [2] [9] [11] [13]) は，論理プログラミングから派生した宣言的プログラミングパラダイムである．ASP 言語は一階論理に基づく知識表現言語の一種である．ASP システムは安定モデル意味論 [9] に基づく解集合を計算するシステムである．近年，SAT の技術を応用した高速な ASP システムが開発され，プランニング，有界モデル検査，システム生物学など様々な分野への実用的応用が急速に拡大している [7].

組合せ遷移問題に対して ASP を用いる利点としては，ASP 言語の高い表現力により，記号制約を簡潔に記述できる点，インクリメンタル ASP 解法により，学習節を再利用した効率的な解探索が可能である点，探索ヒューリスティックスを簡単にカスタマイズできる点，などが挙げられる．

本稿では，解集合プログラミング (ASP) を用いた組合せ遷移問題の解法として，有界組合せ遷移を提案

A Study on ASP-based Implementation of Combinatorial Reconfiguration Solver

Yuya Yamada, Mutsunori Banbara, 名古屋大学 大学院情報学研究科, Graduate School of Informatics, Nagoya University.

Shin-ichi Minato, 京都大学 大学院情報学研究科, Graduate School of Informatics, Kyoto University.

し，そのソルバーの実装方法について述べる．有界組合せ遷移は，SAT 技術を用いたモデル検査手法の 1 つである有界モデル検査 [1] [3] [4] を組合せ遷移に応用した解法といえる．

有界組合せ遷移は，組合せ遷移問題に対して，制限された長さの (すなわち有界の) 遷移系列を論理プログラムとして記号的に表現し，その論理プログラムを ASP システムで実行することにより，到達可能性の検査を行う．ASP システムが充足可能と判定した場合，制限された長さの到達可能な遷移系列が存在することを意味する．逆に充足不能と判定した場合，制限された長さでは到達不能であることを意味する．この場合，遷移系列の長さを増加させた論理プログラムを構成し，再び ASP システムによる実行を繰り返す．有界組合せ遷移は，到達可能な遷移系列を探すだけで，到達不能の証明は行わない不完全な手続きであるが，検査すべき遷移系列の長さに上限が存在する場合には完全な手続きとなる．

有界組合せ遷移の有効性を評価するために，代表的な組合せ遷移問題の 1 つである k 彩色遷移問題をベンチマークとして，提案ソルバーの性能評価を行った．その結果，インクリメンタル ASP 解法を用いた実装方法がより多くの問題を高速に解くことができることを確認した．

2 組合せ遷移問題と k 彩色遷移問題

組合せ遷移問題 (Combinatorial Reconfiguration Problems; CoRe [12] [14] [16]) は，基となる組合せ問題の実行可能解が形成する解空間グラフを用いて定義される．解空間グラフの頂点は実行可能解を表し，辺は遷移制約に関する実行可能解の間の隣接関係を表す．つまり，組合せ問題が実行可能解が存在するかを判定するのに対し，組合せ遷移問題は，解空間グラフにおいて 2 つの頂点間の到達可能性を判定することが目的となる．

基となる組合せ問題の例として，色数 k のグラフ点彩色問題を考える．この問題は，与えられた有限無向グラフについて，隣接する頂点が同色にならないように各頂点を彩色するとき，グラフが k 色以下で彩色可能かどうかを判定する問題である．

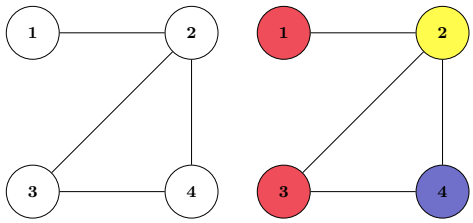

図 1　色数 $k = 3$ のグラフ点彩色問題の入力例 (左) と実行可能解 (右)

グラフ点彩色問題を基とする組合せ遷移問題は，**k 彩色遷移問題** (k-coloring Reconfiguration Problems) ともよばれ盛んに研究されている [5] [6]．k 彩色遷移問題は，色数 k のグラフ点彩色問題と 2 つの実行可能解が与えられたとき，一方の実行可能解から他方の実行可能解へ，1 回の遷移において色が変化する頂点はただ 1 つのみという遷移制約を満たしつつ，実行可能解のみを経由して到達できるかを判定する問題である．k 彩色遷移問題では，グラフの形や色数に制限を加えることで多項式時間で解ける問題が存在するが，一般に $k \geq 4$ において PSPACE 完全であることが知られている．

図 1 に，色数 $k = 3$ のグラフ点彩色問題の例とその実行可能解を示す．図 2 は，到達可能な $k = 4$ 彩色遷移問題の例である．この例は，スタート状態 ($t = 0$) からゴール状態 ($t = 3$) へ，3 ステップの遷移で到達している．また，各状態 t はグラフ点彩色問題の制約を満たしており，1 回の遷移においてちょうど 1 個の頂点の色が変化していることがわかる．

3 解集合プログラミング

解集合プログラミング (Answer Set Programming; ASP [2] [9] [11] [13]) は，演繹データベース，否定を含む論理プログラミング，非単調推論，制約充足 (特に，SAT) を起源にもつ，宣言的プログラミングパラダイムである．ASP の言語は，一般拡張選言プログラムに基づいている．本稿では説明の簡略化のため，そのサブクラスである標準論理プログラムについて説明する．以降，標準論理プログラムを単に論理プログラムとよぶ．

論理プログラムは，以下の形式をした**ルール**の有限

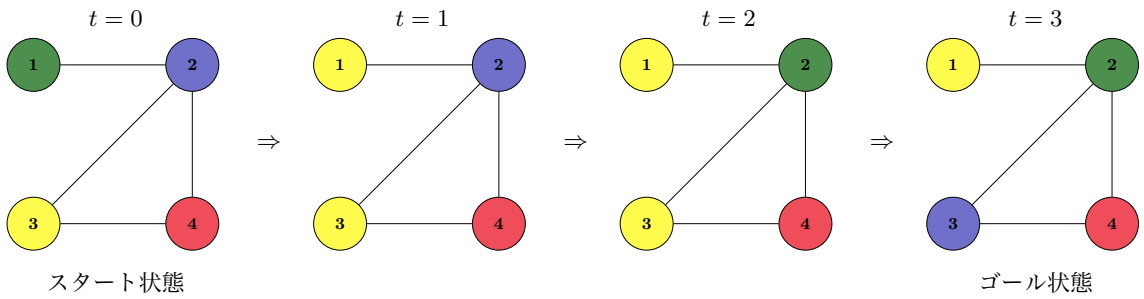

図 2　到達可能な $k = 4$ 彩色遷移問題の例

集合である．

$$a_0 \leftarrow a_1, \ldots, a_m, \sim a_{m+1}, \ldots, \sim a_n$$

ここで，$0 \leq m \leq n$ であり，各 a_i はアトム，$\sim$ はデフォルトの否定 †1，“,” は連言を表す．$\leftarrow$ の左側をヘッド，右側をボディとよぶ．ルールの直観的な意味は，「$a_1, \ldots, a_m$ がすべて成り立ち，$a_{m+1}, \ldots, a_n$ のそれぞれが成り立たないならば，a_0 が成り立つ」である．ボディが空のルール (すなわち $a_0 \leftarrow$) を**ファクト**とよび，$\leftarrow$ を省略してよい．

ヘッドが空のルールを**一貫性制約**とよぶ．

$$\leftarrow a_1, \ldots, a_m, \sim a_{m+1}, \ldots, \sim a_n$$

直観的には，$\leftarrow a_1$ は，a_1 ではないという禁止を表し，$\leftarrow \sim a_1$ は，a_1 でなければならないという強制を表す．また，$\leftarrow a_1, \sim a_2$ は，a_1 が成り立つならば，a_2 が成り立つことを意味する．

ASP 言語は，組合せ問題を解くための便利な拡張構文を備えている．その代表的なものが**選択子**と**個数制約**である．例えば，選択子 $\{a_1; \ldots; a_n\}$ をファクトとして書くと，アトム集合 $\{a_1, \ldots, a_n\}$ の任意の部分集合が成り立つことを意味する．個数制約は選択子の両端に選択可能な個数の上下限を付けたものである．例えば，$lb\ \{a_1; \ldots; a_n\}\ ub \leftarrow Body$ と書くと，$Body$ が成り立つならば，$a_1, \ldots, a_n$ のうち，lb 個以上 ub 個以下が成り立つことを意味する．

ASP システムは，与えられた論理プログラムから，安定モデル意味論 [9] に基づく解集合を計算するソフトウェアである．ASP システムの多くは，変数を含む論理プログラムを変数を含まない論理プログラムに**基礎化**したのち，基礎ソルバーを用いて解集合を計算する．本稿で使用する高速 ASP システム *clingo* †2 は，基礎化のためのグラウンダー *gringo* と基礎ソルバー *clasp* をシームレスに結合したものである．以降で示す論理プログラムのソースコードはすべて *clingo* 言語で書かれており，表記上の対応については表 1 の通りである．

表 1　論理プログラムとソースコードの対応

論理プログラム	$\leftarrow$	,	;	$\sim$
ソースコード	`:-`	`,`	`;`	`not`

ASP を用いた基本的な問題解法は，最初に解きたい問題を論理プログラムとして表現 (ASP 符号化) する．つぎに ASP システムを用いて論理プログラムの解集合を計算する．最後に解集合を解釈してもとの問題の解を得る，という 3 つのステップからなる．以下，グラフ点彩色問題を例にとって，各ステップごとに説明する．

コード 1 に，図 1 のグラフを ASP のファクト形式で表したもの示す．アトム `n/1` は頂点数，`e/1` は辺数，`node/1` は頂点，`edge/2` は辺を表している．ピリオド (“.”) はルールの終わりを表す終端記号である．

グラフ点彩色問題を解く論理プログラムを，コード 2 に示す．コード中の `c` は色数を表す定数であり，実際の値は実行時に *clingo* のオプションとして与えられる．1 行目のアトム `col/1` は色を表し，`col(1..c).` は `col(1).`, `col(2).`, ..., `col(c).` と書くことに等しい．2 行目のルールは，頂点 `X` が色 `C` で塗られることを意味するアトム `color(X,C)` を導入し，個数制約

†1 “失敗による否定” ともよばれる．述語論理で定義される否定 ($\neg$) とは意味が異なる．

†2 `https://potassco.org/`

```
n(4).      e(4).
node(1).   node(2).   node(3).   node(4).
edge(1,2). edge(2,3). edge(2,4). edge(3,4).
```

コード 1　図 1 のグラフの ASP ファクト表現 (graph.lp)

```
1 col(1..c).
2 1 { color(X,C): col(C) } 1 :- node(X).
3 :- not { color(X,C); color(Y,C) } 1, edge(X,Y), col(C).
```

コード 2　色数 c のグラフ点彩色問題を解く論理プログラム (color.lp)

```
% clingo -c c=3 graph.lp color.lp
clingo version 5.4.0
Reading from graph.lp ...
Solving...
Answer: 1
color(1,1) color(2,3) color(3,1) color(4,2)...
SATISFIABLE

Models       : 1+
Calls        : 1
CPU Time     : 0.002s
```

コード 3　*clingo* の実行例

を用いて「各頂点は 1 つの色で塗られる」という制約を表している．3 行目のルールは，一貫性制約と個数制約を使って「辺で結ばれた頂点 X と Y が同じ色 C で塗られることはない」という制約を表している．

ASP システムは，コード 1 のファクトとコード 2 の論理プログラムから解集合を計算する．コード 3 に *clingo* の実行例を示す．色 1 を赤，色 2 を青，色 3 を黄とすると，得られた解集合 { color(1,1), color(2,3), color(3,1), color(4,2)} は，図 1 の右側の彩色を表す．

論理プログラムの基礎化について，もう少し詳しく説明する．*clingo* は，グラウンダー *gringo* を用いてコード 1 のファクトとコード 2 の論理プログラムを基礎化した後，*clasp* を用いて基礎化された論理プログラムの解集合を計算する．基礎化されたルール集合

```
1  node(1).   node(2).   node(3).   node(4).
2  edge(1,2). edge(2,3). edge(2,4). edge(3,4).
3  col(1).    col(2).    col(3).
4  1 { color(1,1) ; color(1,2) ; color(1,3) } 1.
5  1 { color(2,1) ; color(2,2) ; color(2,3) } 1.
6  1 { color(3,1) ; color(3,2) ; color(3,3) } 1.
7  1 { color(4,1) ; color(4,2) ; color(4,3) } 1.
8  :- not {color(1,1) ; color(2,1) } 1.
9  :- not {color(1,2) ; color(2,2) } 1.
10 :- not {color(1,3) ; color(2,3) } 1.
11 :- not {color(2,1) ; color(3,1) } 1.
12 :- not {color(2,2) ; color(3,2) } 1.
13 :- not {color(2,3) ; color(3,3) } 1.
14 :- not {color(2,1) ; color(4,1) } 1.
15 :- not {color(2,2) ; color(4,2) } 1.
16 :- not {color(2,3) ; color(4,3) } 1.
17 :- not {color(3,1) ; color(4,1) } 1.
18 :- not {color(3,2) ; color(4,2) } 1.
19 :- not {color(3,3) ; color(4,3) } 1.
```

コード 4　基礎化された論理プログラム

をコード 4 に示す．4–7 行目がコード 2 の 2 行目に，8–19 行目がコード 2 の 3 行目に対応している．

4　ASP を用いた組合せ遷移問題の解法

ASP 技術を用いた組合せ遷移問題の解法として，**有界組合せ遷移** (Bounded Combinatorial Reconfiguration) を提案し，そのソルバーの実装方法について述べる．

4.1　有界組合せ遷移

有界組合せ遷移では，組合せ遷移問題に対して，制限された長さの遷移系列を論理プログラムとして記号的に表現し，その論理プログラムを ASP システムで実行することにより，到達可能性の検査を行う．ASP システムが充足可能と判定した場合，制限された長さの到達可能な遷移系列が存在することを意味する．逆に充足不能と判定した場合，制限された長さでは到達不能であることを意味する．この場合，遷移系列の長さを増加させた論理プログラムを構成し，再び ASP システムによる実行を繰り返す．有界組合せ遷移は，到達可能な遷移系列を探すだけで，到達不能の証明は行わない不完全な手続きであるが，検査すべき遷移系列の長さに上限が存在する場合には，完全

な手続きとなる．例えば，基の組合せ問題の直径 (任意の 2 つの実行可能解の最短経路のうちの最大長) を上限とすればよい †3．

組合せ遷移問題が与えられたとき，基の組合せ問題の変数集合 $\boldsymbol{x} = \{x_1, x_2, \ldots, x_n\}$，各遷移ステップ $t \geq 0$ に対して，ステップ t での各変数の値を表す変数集合 $\boldsymbol{x}^t = \{x_1^t, x_2^t, \ldots, x_n^t\}$ を導入する．ここで，スタート状態の制約を表す論理式を $S(\boldsymbol{x})$，基の組合せ問題の制約を表す論理式を $C(\boldsymbol{x})$，遷移制約を表す論理式を $T(\boldsymbol{x}, \boldsymbol{x}')$，ゴール状態の制約を表す論理式を $G(\boldsymbol{x})$ とする．スタート状態から ℓ ステップ遷移した後の各変数の値 $\boldsymbol{x}^\ell$ は，$S(\boldsymbol{x}^0) \wedge \bigwedge_{t=0}^{\ell} C(\boldsymbol{x}^t) \wedge \bigwedge_{t=1}^{\ell} T(\boldsymbol{x}^{t-1}, \boldsymbol{x}^t)$ を満たす．そこで，ゴール状態の制約を満たすかどうか判定するため，$G(\boldsymbol{x}^\ell)$ を付け加えた論理式 φ_ℓ を構成する．

$$\begin{aligned}\varphi_\ell = {} & S(\boldsymbol{x}^0) \\ & \wedge \bigwedge_{t=0}^{\ell} C(\boldsymbol{x}^t) \wedge \bigwedge_{t=1}^{\ell} T(\boldsymbol{x}^{t-1}, \boldsymbol{x}^t) \\ & \wedge G(\boldsymbol{x}^\ell)\end{aligned}$$

与えられたステップ長 ℓ に対し φ_ℓ が充足可能であれば，到達可能な遷移系列が得られる．

4.2 有界組合せ遷移の基本ソルバー

ASP システムを用いて有界組合せ遷移を行う手続きは以下のようになる．

1. ステップ長 ℓ を $\ell = 0$ とする．
2. φ_ℓ を論理プログラムとして記述し，ファクト形式の CoRe 問題インスタンスとともに，ASP システムに入力として与える．
3. ASP システムの出力結果が充足可能であれば終了する．充足不能であれば ℓ の値を 1 増加させ，2 に戻り手続きを繰り返す．ただし，ℓ が基の問題の直径を超えたところで繰り返しを停止できる．

この手続きは，与えられた組合せ遷移問題に対して，到達可能な最短の遷移系列を探索する．一般に，到達不能の証明は行えない不完全な手続きであるが，検査すべきステップ長 ℓ に上限が存在する場合には完全な手続きとなり，到達不能も判定できる．この手続きを，単純に実装したプログラムを**基本ソルバー**とよぶことにする．

†3 理論的には，問題の直径を超えるまで手続きを繰り返せばよいが，実用的な問題ではしばしば直径が大きすぎ，現実的には適用できない場合もある．

```
1 n(4).       e(4).
2 node(1).    node(2).    node(3).    node(4).
3 edge(1,2).  edge(2,3).  edge(2,4).  edge(3,4).
4 start(1,4). start(2,2). start(3,3). start(4,1).
5 goal(1,3).  goal(2,4).  goal(3,2).  goal(4,1).
```

コード 5　$k = 4$ 彩色遷移問題のファクト表現
(graph_reconfig.lp)

k 彩色遷移問題を例にとって，基本ソルバーを用いた問題解法を説明する．図 2 の $k = 4$ 彩色遷移問題の ASP ファクト表現をコード 5 に示す．コード 1 のグラフ点彩色問題のファクト表現と比較して，スタート状態を表す `start/2` とゴール状態を表す `goal/2` が追加されている．ここで，色 1 は赤，色 2 は青，色 3 は黄，色 4 は緑とする．例えば，`start(1,4).` は，スタート状態 ($t = 0$) において頂点 1 が色 4 (緑) で塗られることを表す．

コード 6 に，k=c 彩色遷移問題を解く論理プログラムとして `changed` 符号化を示す．コード中の `c` は色数，`length` はステップ長を表す．1 行目のアトム `t/1` はステップ数を表す．アトム `color(X,C,T)` は，ステップ `T` において，頂点 `X` が色 `C` で塗られることを意味する．3 行目はスタート状態の制約を表す．5–7 行目は基のグラフ点彩色問題の制約を表す．9–11 行目は遷移制約を表す．9 行目の補助アトム `changed(X,T)` は，ステップ `T-1` と `T` の間で頂点 `X` の色が変化したことを意味する．11 行目のルールは，個数制約を用いて「各ステップ `T` において色が変化する頂点はただ 1 つのみ」という制約を表している．

コード 7 に，遷移制約の表現方法が異なる `unchanged` 符号化を示す．`changed` 符号化との違いは，9–10 行目のルールだけである．9 行目の補助アトム `unchanged(X,T)` は，ステップ `T-1` と `T` の間で頂点 `X` の色が変化しないことを意味する．10 行目のルールは，個数制約を用いて「各ステップ `T` において

```
col(1..c).    t(0..length).
% start
:- not color(X,C,0), start(X,C).
% constraints
1 { color(X,C,T): col(C) } 1 :- node(X), t(T).
:- not { color(X,C,T); color(Y,C,T) } 1,
   edge(X,Y), col(C), t(T).
% transition
changed(X,T) :- color(X,C1,T), color(X,C2,T-1),
                C1 != C2, T >= 1.
:- not 1  { changed(X,T) } 1, t(T), T >= 1.
% goal
:- not color(X,C,length), goal(X,C).
```

コード 6　k =c 彩色遷移問題を解く changed符号化 (changed.lp)．定数 length はステップ長を表す．

```
col(1..c).    t(0..length).
% start
:- not color(X,C,0), start(X,C).
% constraints
1 { color(X,C,T): col(C) } 1 :- node(X), t(T).
:- not { color(X,C,T); color(Y,C,T) } 1,
   edge(X, Y), col(C), t(T).
% transition
unchanged(X,T) :- color(X,C,T), color(X,C,T-1), T >= 1.
:- not N-1 { unchanged(X,T) } N-1 , n(N), t(T), T >= 1.
% goal
:- not color(X,C,length), goal(X,C).
```

コード 7　k =c 彩色遷移問題を解く unchanged符号化 (unchanged.lp)．定数 length はステップ長を表す．

色が変化しない頂点は N-1 個」という制約を表している．この unchanged 符号化の特徴は，changed 符号化と比較して，基礎化後のルール数が少なく抑えられる点である．

基本ソルバーは，CoRe 問題インスタンスのファクト表現と，CoRe 問題を解く符号化を入力として，到達可能性を検査する．コード 8 に，k 彩色遷移問題に対する基本ソルバーの動作を示す．この例より，ステップ長 length を 0 から 3 まで変化させ，*clingo* を合計 4 回実行することにより，最短の遷移系列が得られることがわかる．length=3 で得られた解集合は，図 2 に示した遷移系列に対応している．

```
% clingo graph_reconfig.lp changed.lp -c c=4 -c length=0
clingo version 5.4.0
Reading from graph_reconfig.lp ...
Solving...
UNSATISFIABLE

Models       : 0
Calls        : 1
CPU Time     : 0.003s

% clingo graph_reconfig.lp changed.lp -c c=4 -c length=1
clingo version 5.4.0
Reading from graph_reconfig.lp ...
Solving...
UNSATISFIABLE

Models       : 0
Calls        : 1
CPU Time     : 0.003s

% clingo graph_reconfig.lp changed.lp -c c=4 -c length=2
clingo version 5.4.0
Reading from graph_reconfig.lp ...
Solving...
UNSATISFIABLE

Models       : 0
Calls        : 1
CPU Time     : 0.005s

% clingo graph_reconfig.lp changed.lp -c c=4 -c length=3
clingo version 5.4.0
Reading from graph_reconfig.lp ...
Solving...
Answer: 1
color(1,4,0) color(2,2,0) color(3,3,0) color(4,1,0)
color(1,3,1) color(2,2,1) color(3,3,1) color(4,1,1)
color(1,3,2) color(2,4,2) color(3,3,2) color(4,1,2)
color(1,3,3) color(2,4,3) color(3,2,3) color(4,1,3)...
SATISFIABLE

Models       : 1+
Calls        : 1
CPU Time     : 0.005s
```

コード 8　*clingo* の実行例

4.3　ソルバーの改良

有界組合せ遷移は，ステップ長 ℓ を増やしながら，複数の φ_ℓ を繰り返し解く必要がある．しかし，各 φ_ℓ に含まれる制約の大部分は同じであるため，実際に

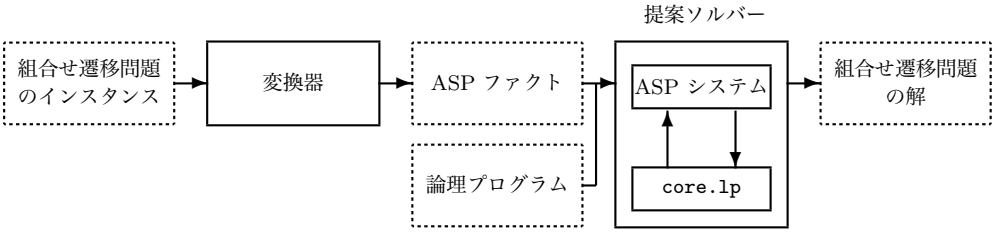

図 3　インクリメンタル ASP 解法を用いた改良ソルバーの構成図

$\varphi_{\ell-1}$ と φ_ℓ の差をみてみると，
$\varphi_{\ell-1} \oplus \varphi_\ell = \{C(\boldsymbol{x}^\ell), T(\boldsymbol{x}^{\ell-1}, \boldsymbol{x}^\ell), G(\boldsymbol{x}^{\ell-1}), G(\boldsymbol{x}^\ell)\}$
であり，多くの制約は共通していることがわかる．そのため，各 φ_ℓ に対して毎回 ASP システムを実行する基本ソルバーは，同一の探索空間を何度も調べることになり求解効率が低下する．また，毎回 φ_ℓ 全体を基礎化するオーバーヘッドも無視できない．

この問題を解決するために，有界モデル検査やプランニングの実装で使われるインクリメンタル ASP 解法を用いる [8]．この解法は，ルールを追加・削除した一連の論理プログラムを，同様の探索失敗を防ぐために獲得した学習節を再利用しながら，連続的に解くための手法である．

インクリメンタル ASP 解法を用いた改良ソルバーは，ステップ長 ℓ を増やしながら，段階的に φ_ℓ を構成し，獲得した学習節を再利用した効率のよい到達可能性の検査を可能とする．改良ソルバーの構成を図 3 に示す．

改良ソルバーにおける有界組合せ遷移を行う手続き (図 3 の `core.lp`) は以下のようになる．

1. ASP システムを起動する．
2. ファクト形式の CoRe 問題インスタンスと，S を表すサブプログラム，C と T を表すサブプログラム，G を表すサブプログラムから構成される論理プログラムを与える．
3. ステップ長 ℓ を $\ell = 0$ とする．
4. $\ell > 0$ であれば，$G(\boldsymbol{x}^{\ell-1})$ を表すルールを削除する．
5. $\ell = 0$ であれば，$S(\boldsymbol{x}^0)$ を表すルールを追加する．
6. $C(\boldsymbol{x}^\ell)$ と $T(\boldsymbol{x}^{\ell-1}, \boldsymbol{x}^\ell)$ を表すルールを追加する．
7. $G(\boldsymbol{x}^\ell)$ を表すルールを追加する．
8. ASP システムを用いて到達可能性を検査する．
9. ASP システムの出力結果が充足可能であれば終了する．充足不能であれば ℓ の値を増加させ，4 に戻り手続きを繰り返す．ただし，ℓ が基の問題の直径を超えたところで繰り返しを停止できる．
10. ASP システムを終了する．

到達不能の証明については，4.2 節で述べた基本ソルバーの手続きと同じである．

図 4 に，*clingo* の Python インターフェースを用いて上記の手続きを実装した改良ソルバーを示す．変数 `core_max` はステップ長 ℓ の上限値を表す．`core_stop` はプログラムの停止条件を表す変数であり，基本的には `SAT` を指定する．11 行目で定義されている `step` は ℓ に対応する．16–18 行目は手続き 4 に対応している．19–20 行目は手続き 5 に対応している．21 行目は手続き 6 に対応している．22 行目は手続き 7 に対応している．23 行目でステップ数 ℓ で追加されるルールの基礎化を行い，25 行目の `prg.solve()` で到達可能性を検査している．

改良ソルバーの入力となる論理プログラムは，`base`, `step(t)`, `check(t)` の 3 つのパートから構成される．`base` パートには，スタート状態の制約 S を記述する．`step(t)` パートには，各ステップ $0 \leq \mathtt{t} \leq \ell$ において満たすべき基の組合せ問題の制約 C および遷移制約 T を記述する．`check(t)` パートには，ゴール状態の制約 G を記述する．コード 9 とコード 10 に，k=c 彩色遷移問題を解く 2 種類の論理プログラムを示す．これらは，それぞれコード 6 とコード 7 の論理プログラムに対応している．改良ソルバーの実行例をコード 11 に示す．得られた解集合は，図 2 の遷移系列に対応している．

```
#script (python)
import clingo

def get(val, default):
    return val if val != None else default

def main(prg):
    core_stop  = get(prg.get_const("core_stop"), clingo.String("SAT"))
    core_max   = prg.get_const("core_max")

    step, ret = 0, None
    while ((core_max is None or step <= core_max.number) and
            (step == 0 or (core_stop.string == "SAT" and not ret.satisfiable) or
            (core_stop.string == "UNSAT" and not ret.unsatisfiable))):
        parts = []
        if step > 0:
            prg.release_external(clingo.Function("query", [step-1]))
            prg.cleanup()
        else:
            parts.append(("base", []))
        parts.append(("step", [step]))
        parts.append(("check", [step]))
        prg.ground(parts)
        prg.assign_external(clingo.Function("query", [step]), True)
        ret, step = prg.solve(), step+1

    print('c step: %s' % str(step-1))
    if (step != 0 and
        (core_stop.string != "SAT" or ret.satisfiable) and
        (core_stop.string != "UNSAT" or ret.unsatisfiable)):
        print('s REACHABLE')
    else:
        print('s UNREACHABLE')

#end.
#program check(t).
#external query(t).
```

図 4　*clingo* の Python API を用いた改良ソルバーの実装 (core.lp)

5　実行実験

提案した有界組合せ遷移の有効性を評価するために，k 彩色遷移問題をベンチマークとして開発したソルバーの性能評価を行った．ベンチマーク問題には，独自に作成した k 彩色遷移問題 (90 問) を使用した．作成方法は以下の通りである．

- Graph Coloring and its Generalization †4 で公開されているグラフ点彩色問題のインスタンス 127 個のうち，彩色数が判明している 44 個 [15] をベンチマーク問題の候補とした．
- 44 個のグラフに対して，彩色数での実行可能解の総数を計算できるか調査した．総数が得られた 9 個のグラフを表 2 に示す．
- 総数が得られた 9 個のグラフについて，実行可能解をランダムに 2 つ選びベンチマーク問題を各 10 問ずつ作成した (計 90 問)．

k 彩色遷移問題を解く論理プログラムには，changed 符号化，unchanged 符号化，changed_inc 符号化，unchanged_inc 符号化の 4 つを使用した．ステップ

†4 https://mat.tepper.cmu.edu/COLOR04/

```
#program base.
col(1..c).
color(X,C,0) :- start(X,C).

#program step(t).
1 { color(X,C,t): col(C) } 1 :- node(X).
:- not { color(X,C,t); color(Y,C,t) } 1, edge(X,Y), col(C).

changed(X,t) :- color(X,C1,t), color(X,C2,t-1),
                C1 != C2, t >= 1.
:- not 1 { changed(X,t) } 1, t >= 1.

#program check(t).
:- not color(X,C,t), goal(X,C), query(t).
```

コード 9　k=c 彩色遷移問題を解く changed_inc 符号化 (changed_inc.lp)

```
#program base.
col(1..c).
:- not color(X, C, 0), start(X, C).

#program step(t).
1 { color(X,C,t): col(C) } 1 :- node(X).
:- not { color(X,C,t); color(Y,C,t) } 1, edge(X,Y), col(C).

unchanged(X,t) :- color(X,C,t), color(X,C,t-1), t >= 1.
:- not N-1 { unchanged(X,t) } N-1 , n(N), t >= 1.

#program check(t).
:- not color(X,C,t), goal(X,C), query(t).
```

コード 10　k=c 彩色遷移問題を解く unchanged_inc 符号化 (unchanged_inc.lp)

長の上限値には，解の総数から 1 ひいた値を使用した †5．ASP システムには，*clingo* のバージョン 5.4.0 (jumpy) を使用した．計算環境は Mac OS 3.2GHz 6 コア Intel Core i7，64GB メモリである．1 問あたりの制限時間は 1 時間とした．

表 3 に実験結果を示す．左から順に問題名，到達可能/到達不能の別，ステップ長，各符号化の CPU 時間を表している．解けた問題の数は，changed 符号化が 21 問，unchanged 符号化が 21 問，changed_inc 符号化が 67 問，unchanged_inc 符号化が 71 問である．unchanged_inc が最も多くの問題を解き，平均 CPU 時間も最も短い．図 5 にカクタスプロットを示す．図 5 の縦軸は CPU 時間，横軸は解けた問題数を表す．図より，unchanged_inc 符号化が，他の符号化と比較して，より多くの問題をより高速に解いていることが確認できる．以上のことから，インクリメンタル ASP 解法を用いた改良ソルバーの優位性が確認できた．

†5　ただし，グラフ myciel4 から作成した問題については，*clingo* が対応している数よりも解の総数が大きかったため，ステップ長の上限値を $2^{31}-1$ とした．

```
% clingo graph_reconfig.lp changed_inc.lp core.lp -c c=4
clingo version 5.4.0
Reading from graph_reconfig.lp ...
Solving...
Solving...
Solving...
Solving...
Answer: 1
color(1,4,0) color(2,2,0) color(3,3,0) color(4,1,0)
color(1,3,1) color(2,2,1) color(3,3,1) color(4,1,1)
color(1,3,2) color(2,4,2) color(3,3,2) color(4,1,2)
color(1,3,3) color(2,4,3) color(3,2,3) color(4,1,3)...
SATISFIABLE
c step: 3
s REACHABLE

Models       : 1+
Calls        : 4
CPU Time     : 0.029s
```

コード 11　提案ソルバーの実行例

表 2　独自に生成した k 彩色遷移問題のベンチマーク問題

グラフ名	頂点数	辺数	彩色数	解の総数
1-FullIns_3	30	100	4	50,693,280
le450_5a	450	5,714	5	3,840
le450_5c	450	9,803	5	120
le450_5d	450	9,757	5	960
myciel3	11	20	4	12,480
myciel4	23	71	5	2,845,658,400
queen5_5	25	160	5	240
queen6_6	36	290	7	100,800
queen7_7	49	476	7	20,160

表 3　実験結果: CPU 時間

問題名	到達可能/到達不能	ステップ長	changed	unchanged	changed_inc	unchanged_inc
			基本ソルバー		改良ソルバー	
1-FullIns_3_col4_1	REACHABLE	11	1.374	1.071	1.291	0.756
le450_5a_col5_1	UNREACHABLE	3,839	TO	TO	3.761	2.991
le450_5a_col5_2	UNREACHABLE	3,839	TO	TO	2.943	11.673
le450_5a_col5_3	UNREACHABLE	3,839	TO	TO	4.662	22.048
le450_5a_col5_4	UNREACHABLE	3,839	TO	TO	TO	7.776
le450_5a_col5_5	UNREACHABLE	3,839	TO	TO	TO	26.723
le450_5a_col5_6	UNREACHABLE	3,839	TO	TO	3.618	3.220
le450_5a_col5_7	UNREACHABLE	3,839	TO	TO	1318.024	8.698
le450_5a_col5_8	UNREACHABLE	3,839	TO	TO	3.367	18.226
le450_5a_col5_9	UNREACHABLE	3,839	TO	TO	2.943	2.847
le450_5a_col5_10	UNREACHABLE	3,839	TO	TO	2.968	3.196
le450_5c_col5_1	UNREACHABLE	119	TO	TO	2.189	2.165
le450_5c_col5_2	UNREACHABLE	119	TO	TO	150.018	2.148
le450_5c_col5_3	UNREACHABLE	119	TO	TO	150.283	2.701
le450_5c_col5_4	UNREACHABLE	119	TO	TO	152.434	2.102
le450_5c_col5_5	UNREACHABLE	119	TO	TO	2.209	2.114
le450_5c_col5_6	UNREACHABLE	119	TO	TO	2.219	2.141
le450_5c_col5_7	UNREACHABLE	119	TO	TO	151.252	2.150
le450_5c_col5_8	UNREACHABLE	119	TO	TO	2.878	2.176
le450_5c_col5_9	UNREACHABLE	119	TO	TO	2.827	2.146
le450_5c_col5_10	UNREACHABLE	119	TO	TO	2.246	2.166
le450_5d_col5_1	UNREACHABLE	959	TO	TO	2.680	5.653
le450_5d_col5_2	UNREACHABLE	959	TO	TO	4.097	2.621
le450_5d_col5_3	UNREACHABLE	959	TO	TO	3.421	2.600
le450_5d_col5_4	UNREACHABLE	959	TO	TO	155.850	6.960
le450_5d_col5_5	UNREACHABLE	959	TO	TO	3.294	2.618
le450_5d_col5_6	UNREACHABLE	959	TO	TO	199.336	25.273
le450_5d_col5_7	UNREACHABLE	959	TO	TO	3.308	12.001
le450_5d_col5_8	UNREACHABLE	959	TO	TO	2.695	15.224
le450_5d_col5_9	UNREACHABLE	959	TO	TO	3.351	13.121
le450_5d_col5_10	UNREACHABLE	959	TO	TO	333.183	7.974
myciel3_col4_1	REACHABLE	11	0.179	0.148	0.065	0.065
myciel3_col4_3	REACHABLE	11	0.206	0.164	0.082	0.080
myciel3_col4_4	REACHABLE	13	0.486	0.445	0.303	0.583
myciel3_col4_7	REACHABLE	8	0.090	0.072	0.042	0.036
myciel3_col4_8	REACHABLE	9	0.119	0.094	0.048	0.044
myciel4_col5_2	REACHABLE	16	42.482	48.721	20.308	134.056
myciel4_col5_3	REACHABLE	14	17.098	21.211	5.286	80.848
myciel4_col5_4	REACHABLE	7	0.256	0.193	0.086	0.072
myciel4_col5_6	REACHABLE	10	1.525	1.391	1.052	1.558
myciel4_col5_10	REACHABLE	17	212.682	73.646	29.999	200.068
queen5_5_col5_1	UNREACHABLE	239	439.863	302.994	0.073	0.082
queen5_5_col5_2	UNREACHABLE	239	437.481	301.194	0.074	0.057
queen5_5_col5_3	UNREACHABLE	239	441.161	303.136	0.073	0.589
queen5_5_col5_4	UNREACHABLE	239	443.299	304.547	0.061	0.057
queen5_5_col5_5	UNREACHABLE	239	441.616	302.366	0.074	0.067
queen5_5_col5_6	UNREACHABLE	239	446.811	301.201	0.061	0.149
queen5_5_col5_7	UNREACHABLE	239	441.071	303.449	0.074	0.056
queen5_5_col5_8	UNREACHABLE	239	442.539	303.821	0.073	0.137
queen5_5_col5_9	UNREACHABLE	239	443.353	302.952	0.074	0.066
queen5_5_col5_10	UNREACHABLE	239	446.027	305.350	0.073	0.058
queen6_6_col7_1	UNREACHABLE	100,799	TO	TO	5.375	5.488
queen6_6_col7_2	UNREACHABLE	100,799	TO	TO	5.337	5.292
queen6_6_col7_3	UNREACHABLE	100,799	TO	TO	5.345	4.390
queen6_6_col7_4	UNREACHABLE	100,799	TO	TO	5.714	4.358
queen6_6_col7_5	UNREACHABLE	100,799	TO	TO	4.274	5.178
queen6_6_col7_6	UNREACHABLE	100,799	TO	TO	5.403	5.129
queen6_6_col7_7	UNREACHABLE	100,799	TO	TO	5.219	4.418
queen6_6_col7_8	UNREACHABLE	100,799	TO	TO	4.300	4.389
queen6_6_col7_9	UNREACHABLE	100,799	TO	TO	5.365	5.407
queen6_6_col7_10	UNREACHABLE	100,799	TO	TO	TO	4.412
queen7_7_col7_1	UNREACHABLE	20,159	TO	TO	2804.714	949.947
queen7_7_col7_2	UNREACHABLE	20,159	TO	TO	1.342	907.100
queen7_7_col7_3	UNREACHABLE	20,159	TO	TO	1.586	1.499
queen7_7_col7_4	UNREACHABLE	20,159	TO	TO	TO	2.768
queen7_7_col7_5	UNREACHABLE	20,159	TO	TO	1.622	6.052
queen7_7_col7_6	UNREACHABLE	20,159	TO	TO	1.803	1626.628
queen7_7_col7_7	UNREACHABLE	20,159	TO	TO	1.343	1.590
queen7_7_col7_8	UNREACHABLE	20,159	TO	TO	1.481	3.652
queen7_7_col7_9	UNREACHABLE	20,159	TO	TO	1224.680	3.760
queen7_7_col7_10	UNREACHABLE	20,159	TO	TO	1.557	1.349
		平均 CPU 時間	223.796	151.341	101.758	59.095
		解けた問題数	21	21	67	71

6　おわりに

本稿では，解集合プログラミング (ASP) を用いた組合せ遷移問題の解法として，有界組合せ遷移を提案し，そのソルバーの実装方法について述べた．有界組合せ遷移の有効性を評価するために，代表的な組合せ遷移問題の 1 つである k 彩色遷移問題をベンチマークとして，提案ソルバーの性能評価を行った．その結果，インクリメンタル ASP 解法を用いた改良ソルバーがより多くの問題を高速に解くことができる

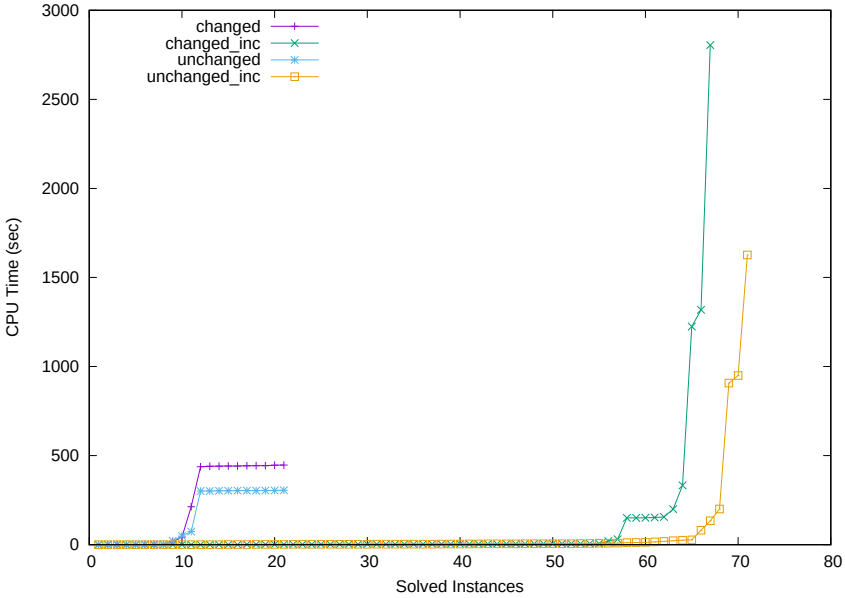

図 5　カクタスプロット

ことを確認した．

組合せ遷移問題を解く汎用ソルバーの研究開発はまだ始まったばかりである．今後の課題として，改良ソルバーの高速化があげられる．高速化の手法としては，アルゴリズムの刷新や探索ヒューリスティックスの改良を考えている．集合被覆問題や独立点集合を基とする組合せ遷移問題，15 パズルなどの問題記述例の蓄積も重要な課題の一つである．また，k 彩色遷移問題の ASP 符号化の改良も考えている．

参 考 文 献

[1] 番原睦則, 田村直之: SAT によるシステム検証, 人工知能学会誌, Vol. 25, No. 1(2010), pp. 122–129.

[2] Baral, C.: *Knowledge Representation, Reasoning and Declarative Problem Solving*, Cambridge University Press, 2003.

[3] Biere, A.: Bounded Model Checking, *Handbook of Satisfiability*, IOS Press, 2009, pp. 457–481.

[4] Biere, A., Cimatti, A., Clarke, E. M., and Zhu, Y.: Symbolic Model Checking without BDDs, *Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS 1999), LNCS 1579*, 1999, pp. 193–207.

[5] Bonsma, P. S. and Cereceda, L.: Finding Paths between graph colourings: PSPACE-completeness and superpolynomial distances, *Theoretical Computer Science*, Vol. 410, No. 50(2009), pp. 5215–5226.

[6] Cereceda, L., van den Heuvel, J., and Johnson, M.: Finding paths between 3-colorings, *Journal of Graph Theory*, Vol. 67, No. 1(2011), pp. 69–82.

[7] Erdem, E., Gelfond, M., and Leone, N.: Applications of ASP, *AI Magazine*, Vol. 37, No. 3(2016), pp. 53–68.

[8] Gebser, M., Kaminski, R., Kaufmann, B., and Schaub, T.: Multi-shot ASP solving with clingo, *Theory and Practice of Logic Programming*, Vol. 19, No. 1(2019), pp. 27–82.

[9] Gelfond, M. and Lifschitz, V.: The Stable Model Semantics for Logic Programming, *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, MIT Press, 1988, pp. 1070–1080.

[10] Gopalan, P., Kolaitis, P. G., Maneva, E. N., and Papadimitriou, C. H.: The Connectivity of Boolean Satisfiability: Computational and Structural Dichotomies, *SIAM Journal on Computing*, Vol. 38, No. 6(2009), pp. 2330–2355.

[11] 井上克巳, 坂間千秋: 論理プログラミングから解集合プログラミングへ, コンピュータソフトウェア, Vol. 25, No. 3(2008), pp. 20–32.

[12] Ito, T., Demaine, E. D., Harvey, N. J. A., Papadimitriou, C. H., Sideri, M., Uehara, R., and Uno, Y.: On the complexity of reconfiguration problems, *Theoretical Computer Science*, Vol. 412, No. 12-14(2011), pp. 1054–1065.

[13] Niemelä, I.: Logic Programs with Stable Model Semantics as a Constraint Programming Paradigm, *Ann. Mathematics and Artificial Intel-*

ligence, Vol. 25, No. 3–4(1999), pp. 241–273.

[14] Nishimura, N.: Introduction to Reconfiguration, *Algorithms*, Vol. 11, No. 4(2018), pp. 52.

[15] Tamura, N., Taga, A., Kitagawa, S., and Banbara, M.: Compiling Finite Linear CSP into SAT, *Constraints*, Vol. 14, No. 2(2009), pp. 254–272.

[16] van den Heuvel, J.: The complexity of change, *Surveys in Combinatorics 2013*, Blackburn, S. R., Gerke, S., and Wildon, M.(eds.), London Mathematical Society Lecture Note Series, Vol. 409, Cambridge University Press, 2013, pp. 127–160.