解集合プログラミングを用いた配電網問題の解法

山田 健太郎 [1]，湊 真一 [2]，田村 直之 [3]，番原 睦則 [4]

[1] 名古屋大学大学院 情報学研究科
yken66@nagoya-u.jp

[2] 京都大学大学院 情報学研究科
minato@i.kyoto-u.ac.jp

[3] 神戸大学 情報基盤センター
tamura@kobe-u.ac.jp

[4] 名古屋大学大学院 情報学研究科
banbara@nagoya-u.jp

概 要　配電網の構成技術はスマートグリッドや，災害時の障害箇所の迂回構成などを支える重要な研究課題である．配電網問題は，供給ネットワークに関するトポロジ制約と，電流・電圧に関する電気制約を満たすスイッチの開閉状態を求める問題である．配電網遷移問題は，配電網問題とその 2 つの実行可能解が与えられたとき，一方から他方へ，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．

本論文では，解集合プログラミング (ASP) を用いた配電網問題の解法および配電網遷移問題への拡張について述べる．提案解法では，まず与えられた問題インスタンスを ASP のファクト形式に変換した後，そのファクトと配電網 (遷移) 問題を解くための ASP 符号化を結合した上で，高速 ASP システムを用いて解を求める．提案解法の評価として，実用規模の問題を含む問題集を用いた実験結果について述べる．

1　はじめに

電力網の構成制御は，エネルギーの節約や安定した電力供給を支える重要な研究課題である．電力網は，高電圧で発電所と変電所を結ぶ**送電網**と，低電圧で変電所と家庭や工場といった需要家を結ぶ**配電網**に分類される．配電網は変電所と需要家との間で構成される電力供給ネットワークであり，その構成技術はスマートグリッドや，災害時の障害箇所の迂回構成などを支える重要な基盤技術である．

配電網問題は，供給経路に関する**トポロジ制約**と，電流・電圧に関する**電気制約**を満たしつつ，電力の損失を最小にするスイッチの開閉状態を求めることが目的である [**?**]．トポロジ制約は，短絡 (供給経路上のループ，複数の変電所と結ばれる需要家) と停電 (変電所と結ばれない需要家) が発生しないことを保証する．電気制約は，供給経路の各区間で許容電流を超えないこと，電気抵抗による電圧降下が許容範囲を超えないことを保証する．本稿ではトポロジ制約のみの配電網問題を対象とする．

配電網問題の例を図 1 に示す．この例は，3 つの変電所 $\{s_a, s_b, s_c\}$[1]，16 個のスイッチ $\{sw_1, \ldots, sw_{16}\}$，22 箇所の配電区間 $\{s_1, \ldots, s_{22}\}$ から構成されている．また，各配電区間にかかる負荷電流は，表 1 の通りである．例えば，区間 s_1 にかかる負荷は，$I_1 = 40\ (A)$ である．配電網問題の実行可能解は閉じたスイッチの集合で表すことができる．

[1] 本稿では，変電所に直接つながっている配電線を変電所として扱う．

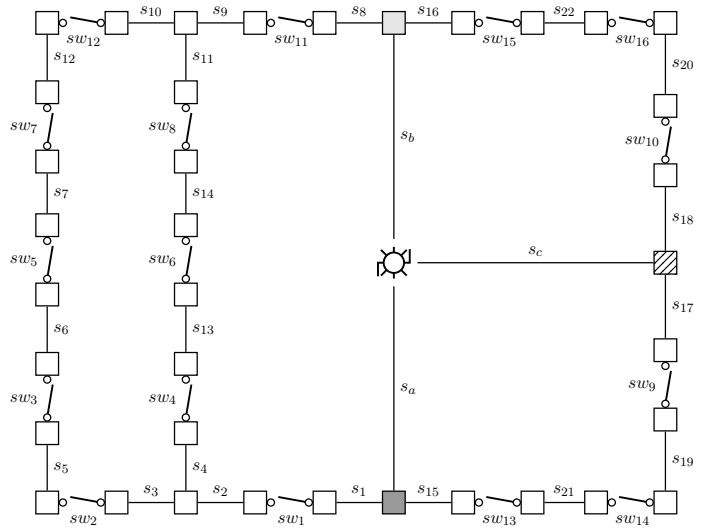

図 1. 配電網問題の例

表 1. 負荷電流の一覧 (A)

I_a	=	16	I_b	=	41	I_c	=	16	I_1	=	40	I_2	=	5
I_3	=	34	I_4	=	0	I_5	=	11	I_6	=	34	I_7	=	24
I_8	=	31	I_9	=	45	I_{10}	=	24	I_{11}	=	0	I_{12}	=	45
I_{13}	=	21	I_{14}	=	20	I_{15}	=	0	I_{16}	=	0	I_{17}	=	0
I_{18}	=	0	I_{19}	=	35	I_{20}	=	20	I_{21}	=	35	I_{22}	=	20

配電網問題は求解困難な組合せ最適化問題の一種であり，これまでフロンティア法を用いた解法等が提案されている [**?**]．トポロジ制約のみの配電網問題は，与えられた連結グラフと根と呼ばれる特殊なノードから，**根付き全域森**を求める部分グラフ探索問題に帰着できることが知られている [**?**]．以降，この問題を**根付き全域森問題**と呼ぶ．

解集合プログラミング (Answer Set Programing; ASP[**?**, **?**, **?**, **?**]) は，論理プログラミングから派生した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り，論理プログラムは ASP のルールの有限集合である．ASP システムは論理プログラムから安定モデル意味論 [**?**] に基づく解集合を計算するシステムである．近年，SAT ソルバーの技術を応用した高速な ASP システムが確立され，制約充足問題，プランニング，システム生物学，時間割問題，システム検証など様々な分野への実用的応用が急速に拡大している [**?**]．

本稿では，解集合プログラミングを用いた根付き全域森問題の解法について述べる．提案アプローチでは，まず与えられた問題インスタンスを ASP のファクト形式に変換した後，ASP ファクトと根付き全域森問題を解くための ASP 符号化と結合した上で，高速 ASP システムを用いて解を求める (図 2 参照)．

根付き全域森問題を解く ASP 符号化 (論理プログラム) として，基本符号化と改良符号化の 2 種類を考案した．特に，改良符号化は，根付き全域森問題の連結制約を ASP の個数制約で表現することにより，基礎化後のルール数を少なく抑えるよう工夫されており，大規模な問題に対する有効性が期待できる．

また，配電網における障害時の復旧予測への応用を狙いとし，根付き全域森問題の 2 つの実行可能解が，局所的な変形による遷移だけで互いに移りあえるかを問う “解の遷移問題” への拡張を行なった．この遷移問題を解くには，複数の根付き全域森問題を繰り返し解く必要がある．しかし，各問題中の制約の大部分は共通であるため，ASP システムが同一の探索空間を何度も調べることに

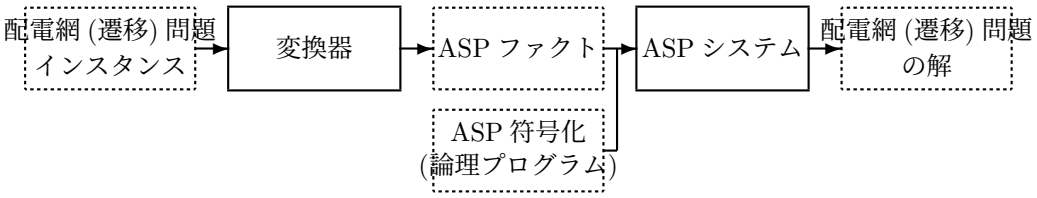

図 2. 提案アプローチの構成図

なり，求解効率が低下するという問題点がある．この問題を解決するために，マルチショット ASP 解法を用いた実装を提案する．この解法は，ASP システムが同様の探索失敗を避けるために獲得した学習節を (部分的に) 保持することで，無駄な探索を行うことなく，制約を追加した論理プログラムを連続的に解くことができる．そのため，求解性能の向上が期待できる．

提案アプローチの有効性を評価するために，DNET (Power Distribution Network Evaluation Tool) [2] に公開されている問題集と，Graph Coloring and its Generalizations [3] に公開されているグラフ彩色問題を基に生成した根付き全域森問題を用いて，実行実験を行なった．その結果，改良符号化は，基本符号化と比較して，より多くの問題をより高速に解くことができた．また，改良符号化は辺数 (スイッチ数) が 40,000 個を超える問題も解いており，大規模問題に対する ASP の有効性が確認できた．

解の遷移問題については，DNET で公開されている実用規模の問題 (fukui-tepco) をベースとした問題集を用いて実験を行った．その結果，マルチショット ASP 解法は，通常の解法と比較して，平均で 3.3 倍の高速化を実現し，マルチショット ASP 解法の優位性が確認できた．

本稿の構成は以下の通りである．2 節で根付き全域森の定義を示し，3 節で解集合プログラミングの説明を行う．4 節で根付き全域森問題の ASP 符号化を示し，**??**節で根付き全域森問題を遷移問題へ拡張を行う．6 節で評価実験とその考察を述べ，最後に，7 節で本稿をまとめる．

2 配電網問題

2.1 配電網問題

本節では，本稿で扱う配電網問題を定式化する．以下の式において，ある配電区間 $i \in \{1, \ldots, n\}$ に対して，配電網構成 X が決まったとする．このとき，各区間 i に対して，変電所と反対側の区間 (木の下流) にある自身を含む区間の集合を C_i^{down} とする．また，各区間 i には，負荷 (電力需要) I_i が与えられる．

区間 i における電流の値 J_i は，次式で与えられる．

$$J_i = \sum_{j \in C_i^{down}} I_j. \tag{1}$$

配電網問題は，以下のように定式化される．

$$subject\ to \quad X \text{ is spanning rooted forest.} \tag{2}$$

$$J_i \leq J^{max},\ i \in \{1, \ldots, n\}. \tag{3}$$

制約 (2)，(3) をそれぞれ，**トポロジ制約**と**電流制約**と呼ぶ．これらの制約を満たす閉じたスイッチの組合せが存在するかどうかの判定問題として本稿では定式化する．なお，文献 [**?**] では，電圧に関する制約などを含んだ組合せ最適化問題として定式化されている．

[2] `https://github.com/takemaru/dnet`
[3] `https://mat.tepper.cmu.edu/COLOR04/`

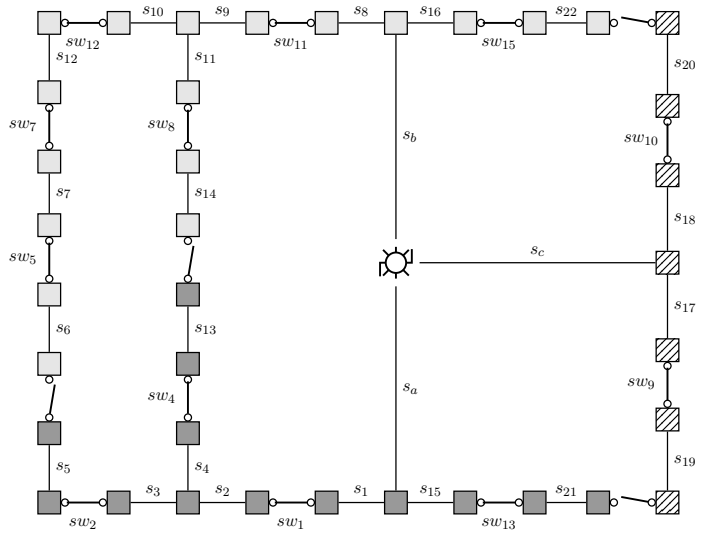

図 3. 配電網問題 (図 1) の解の一例

次に例として，図 1 で示した配電網問題の解を図 3 に示す．この解は電流制約を $V_{max} = 300$ として求められた解である．閉じたスイッチは，$\{sw_1, sw_2, sw_4, sw_5, sw_7, sw_8, sw_9, sw_{10}, sw_{11}, sw_{12}, sw_{13}, sw_{15}\}$ である．このスイッチの集合から決まる配電経路は，図 3 の通り，トポロジ制約を満たしている．また，各変電所に $\{s_a, s_b, s_c\}$ について，それぞれ，$s_a^{down} = \{s_a, s_1, s_2, s_3, s_4, s_5, s_{13}, s_{15}, s_{21}\}$，$s_b^{down}\{s_b, s_6, s_7, s_8, s_9, s_{10}, s_1$ $s_c^{down} = \{s_c, s_{17}, s_{18}, s_{19}, s_{20}\}$ である．したがって，表 1 より，各供給経路に対する電流値の合計は，$J_a = 162$，$J_b = 284$，$J_c = 71$ となり，電流制約も満たしている．

ここで，トポロジ制約を満たす配電網構成は，**根付き全域森**という部分グラフに対応することが知られている [**?**]．根付き全域森は以下のように定義される．

定義. グラフ $G = (V, E)$ と，根と呼ばれる V 上のノードの集合が与えられたとする．このとき，G 上の根付き全域森とは，以下の制約を満たす G の部分グラフ $G' = (V, E'), E' \subseteq E$ である．

1. G' はサイクルを持たない．(非閉路制約)
2. G' の各連結成分は，ちょうど 1 つの根を含む．(根付き連結制約)

本稿では，与えられたグラフ G から，根付き全域森 G' を求める部分グラフ探索問題を**根付き全域森問題**と呼ぶ．

根付き全域森問題の入力例となるグラフを図 4 に示す．図 4 は，図 1 で示した配電網に対応しており，配電区間 $\{s_i \mid 1 \leq i \leq 22\}$ は，スイッチで区切られる 1 つのまとまりごとに**ノード**に対応する．例えば，区間 $\{s_2, s_3, s_4\}$ は，ノード 5 に対応している．スイッチ $\{sw_1, \ldots, sw_{16}\}$ は，**辺**に対応する．また，図中の色付きノード $\{r_1, r_2, r_3\}$ は変電所を含むことを意味しており，**根**に対応している．

根付き全域森の例を図 5 に示す．根付き全域森は，各連結成分が必ずちょうど 1 つの根をもつ木構造を形成することで，非閉路制約と根付き連結制約を満たす．図 5 は，図 3 の配電網問題の解に対応している．

2.2 配電網遷移問題

配電網の構成制御における障害時の復旧予測への応用を狙いとし，ある初期配電網構成 (スタート状態) から目的配電網構成 (ゴール状態) へのスイッチの切替手順を求める組合せ遷移問題を考える．各ステップ t と $t+1$ の間で**遷移制約**を満たしながら，もととなる配電網問題の実行可能解のみを経由し，最短ステップ長での切替手順を求めることが目的である．この組合せ遷移問題を**配電**

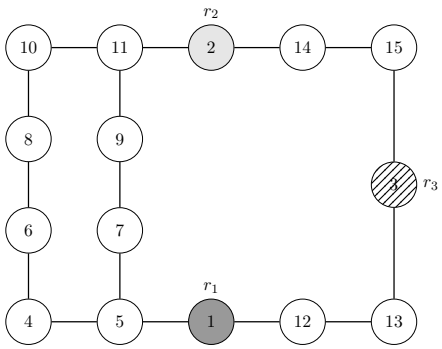

図 4. 配電網問題 (図 1) に対応する根付き全域森問題

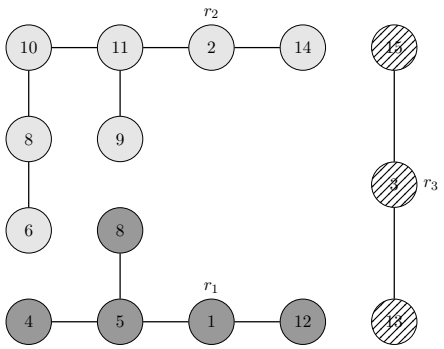

図 5. 配電網問題の解 (図 3) に対応する根付き全域森問題の解

網遷移問題と呼ぶ．本稿では，遷移制約を各ステップ t で切替可能なスイッチの個数を d 個に制限する．

3　解集合プログラミング

ASP の言語は，一般拡張選言プログラムをベースとしている [**?**]．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呼ぶ．

論理プログラムは，以下の形式の**ルール**の有限集合である．

$$a_0 \texttt{ :- } a_1\texttt{ , ... ,}a_m\texttt{ , not } a_{m+1}\texttt{ , ... ,not } a_n\texttt{ .}$$

ここで，$0 \leq m \leq n$ であり，各 a_i はアトム，`not` は**デフォルトの否定**[4]，“`,`” は連言を表す．“`:-`” の左側を**ヘッド**，右側を**ボディ**と呼ぶ．“`.`” はルールの終わりを表す終端記号である．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 `:- .`) を**ファクト**と呼び，“`:-`” を省略することができる．

ヘッドが空のルールを**一貫性制約**と呼ぶ．

$$\texttt{:- } a_1\texttt{ , ... ,}a_m\texttt{ , not } a_{m+1}\texttt{ , ... ,not } a_n\texttt{ .}$$

例えば，一貫性制約 “`:-` a_1`,`a_2`.`” は,「a_1 と a_2 が両方同時に成り立つことはない」を意味し，“`:-` a_1`, not` a_2`.`” は,「a_1 が成り立つならば，a_2 も成り立つ」を意味する．

ASP 言語には，組合せ問題を解くために便利な拡張構文が用意されている．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`; ... ;`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」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，“`:-` a_0`, not` lb `{`a_1`; ... ;`a_n`}` ub`.`” と書くと,「a_0 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」を意味する．また，**重み付き個数制約** (`#sum`) も用意されている．例えば，“`:-` a_0`, not` lb `#sum {`w_1`:`a_1`; ... ;`w_n`:`a_n`}` ub`.`” と書くと,「a_0 が成り立つならば，$a_1, \ldots, a_n$ のうち真となるアトムの重み和が lb 個以上 ub 個以下である」を意味する．項 w_i は重みを表す．

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 [**?**] に基づく解集合を計算するシステムである．本稿では，高性能かつ高機能な ASP システムとして世界中で広く使われている *clingo* [5]を使用する．*clingo* を含め最新の高速 ASP システムは，グラウンダーを用いて変数を

[4] **失敗による否定**とも呼ばれる．述語論理で定義される否定 ($\neg$) とは意味が異なる．

[5] `https://potassco.org/`

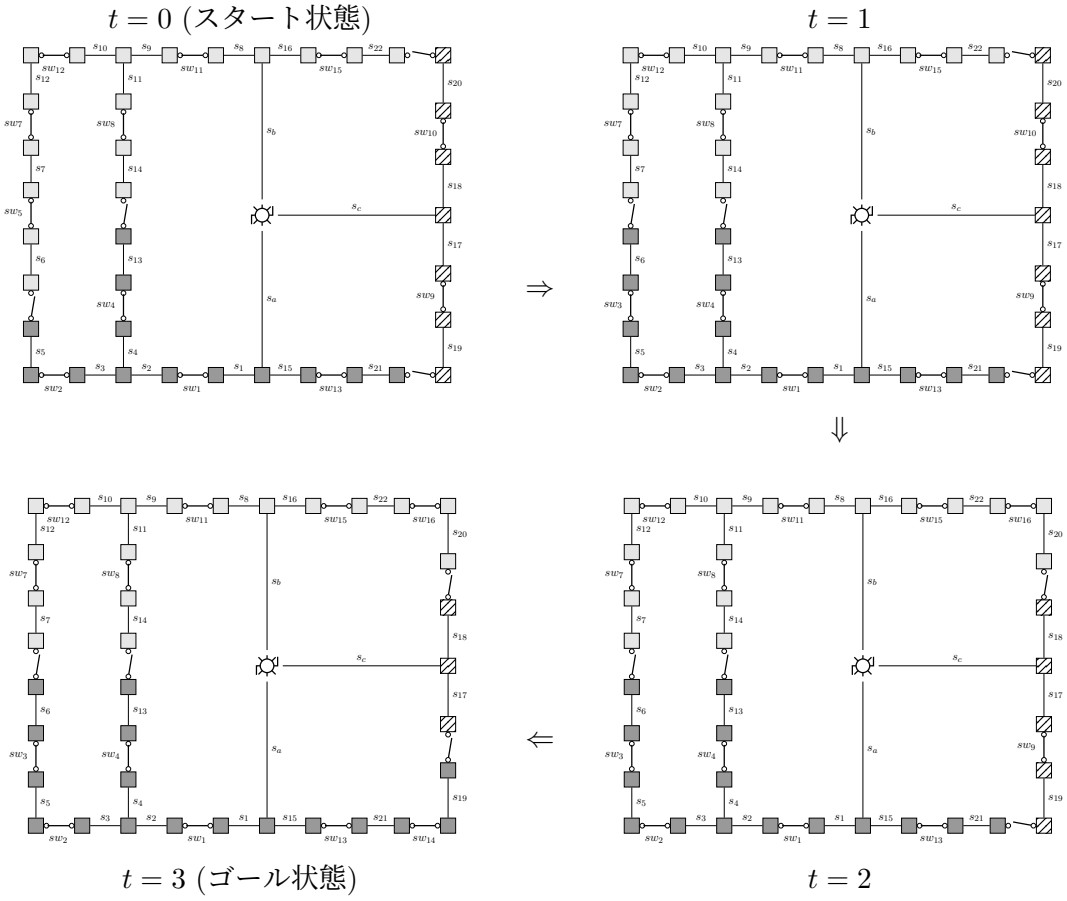

図 6. 根付き全域森遷移問題 (遷移制約 $d = 2$) の解の一例

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変換 (**基礎化**) したのち，ASP ソルバーを用いて解集合を計算する方式が主流となっている．

4　配電網問題の ASP 符号化

本節では，配電網問題の入力と制約を論理プログラムとして表現する方法について述べる．

要相談

ASP ファクト形式. 配電網問題の入力 (図 1) のファクト表現をコード 1 に示す．この問題は，区間 25 個，スイッチ 16 個から構成されている．各区間とスイッチは，隣接関係をアトム `dnet_node/2` によって表される．例えば，ファクト `dnet_node(1,2).` は，ノード 1 とノード 2 が隣接していることを表す．

前処理. 配電網問題を解くための前処理となる ASP 符号化をコード**??**に示す．1 行目のルールは，スイッチを含むノード (スイッチノード) 番号を表すアトム `swt_node(X)` を導入する．この `swt_node(X)` は，ノード番号 `X` はスイッチを含むことを意味する．反対に，2 行目のルールは，スイッチを含まないノード (ジャンクションノード) 番号を表すアトム `jct_node(X)` を導入する．この `jct_node(X)` は，ノード `X` には，スイッチが含まれないことを意味する．

4 行目のルールは，各区間がどのノード番号に属するかを表すアトム `section(S,X)` を導入する．また，5 行目で区間を意味するアトム `section(S)` を導入する．

7,8 行目のルールでは，各区間について，スイッチノードまたはジャンクションノードに含まれていることを表すアトム `swt_node(S)`,`jct_node(S)` を導入する．

10 行目のルールで，各スイッチがどの 2 つの区間を接続されているかを表すアトム `switch(SW,S,T)` を導入する．このアトム `switch(SW,S,T)` は，区間 `(S,T)` はスイッチ `SW` で接続されていることを意味する．

14 行目からのルールは根付き全域森問題の入力を生成するルールである．14,15 行目のルールは，根付き全域森問題のノードと区間を対応させるアトム `node/2` を導入する．アトム `node(X,S)` は，各区間 `S` は根付き全域森のノード `X` に含まれることを意味する．14 行目のルールで，ジャンクションノードはそのまま根付き全域森のノードとして定義され，ジャンクションノードに属する区間は，そのまま対応づけられる．15 行目のルールでは，スイッチノードに含まれ，ジャンクションノードに含まれない各区間について，属するノード番号のうち，小さい方の番号を根付き全域森問題のノードとして定義する．

18 行目は，根ノードを表すアトム `root(X)` を導入する．この `root(X)` は，`node(X,S)` のうち，区間 `S` が変電所と直接つながっているならば，根ノードであることを意味する．また，19 行目で各 `node/2` を，根付き全域森問題の入力のノードを表すアトム `node/1` とする．20 行目で，辺を表すアトム `edge(X,Y)` を導入する．この `edge(X,Y)` は任意の 2 つのノード `(X,Y)` が辺で結ばれていることを意味する．

4.1　根付き全域森問題の ASP 符号化

基本符号化. 根付き全域森問題の ASP 符号化をコード 2 に示す．2 行目のルールは，各辺 `(X,Y)` に対して，解の候補となるアトム `inForest(X,Y)` を導入する．この `inForest(X,Y)` は，辺 `(X,Y)` が根付き全域森に含まれることを意味する．各ノードの到達可能性は 5〜7 行目のルールで表され

る．アトム reached(X,R) は，ノード X は根 R から到達可能であることを意味する．5 行目のルールは，各根ノードは自分自身から到達可能であることを表す．6〜7 行目のルールは，ノード Y が根ノード R から到達可能であり，かつ，辺 (X,Y) が全域森に含まれるならば，ノード X も R から到達可能であることを表す．

非閉路制約は 10〜13 行目のルールで表される．このルールは，各連結成分のノード数と辺数の差が 1 になること (木の性質) を，ASP の重み付き個数制約を使って表している．根付き連結制約は 16〜17 行目のルールで表される．16 行目のルールは，各ノードは少なくとも 1 つの根から到達可能であることを表している (*at-least-one* 制約)．17 行目のルールは，各ノードは高々1 つの根から到達可能であることを表している (*at-most-one* 制約)．これら 2 つの一貫性制約により，各ノードはちょうど 1 つの根から到達可能であることが強制される．

改良符号化．基本符号化は，根付き全域森問題の制約を ASP のルール 7 個で簡潔に表現できる．しかし，根付き連結制約を表す *at-most-one* 制約の基礎化後のルール数は，根ノード数の 2 乗に比例するため，大規模な問題に対する求解性能が低下する可能性がある．この問題を解決するために考案した改良符号化をコード 3 に示す．基本符号化との違いは，根付き連結制約を ASP の個数制約で表している点である (16 行目)．グラフのノード数を n，根ノード数を r として，根付き連結制約の基礎化後のルール数を比較すると，基本符号化が $n(1+{}_rC_2)$ 個なのに対し，改良符号化は n 個と少なく抑えることができる．これにより，大規模な問題に対する有効性が期待できる．

発展符号化．非閉路制約を別の方法で符号化するために，根付き全域森問題を有向グラフに拡張し，各ノードの入次数に関する制約を用いて符号化を行った．考案した発展符号化をコード 4 に示す．9 行目のルールでは，各根ノードについて，入次数は 0 であることを強制している．10 行目のルールでは，根ノード以外の各ノードについて，入次数が 1 であることを保証している．その他のルールは改良符号化と同じである．

4.2 電流制約の ASP 符号化

電流制約の ASP 符号化をコード 5 に示す．3 行目のルールで，根付き全域森問題の解となるアトム inForest(X,Y) をもとに，配電網において，スイッチが閉じて区間が接続されていることを表すアトム connected(SW,S,T) を導入する．このアトムは，区間 S からスイッチ SW によって，もう 1 つの区間 T に接続していることを表す．6 行目のルールで，配電網問題の解を表すアトム closed_switch(SW) を導入する．この closed_switch(SW) は，スイッチ SW が閉じたスイッチであることを意味する．

ジャンクションノード内での上下流の関係は，8,9 行目のルールで表される．アトム entrance_section(S,X) は，区間 S がノード X 内で最も上流にあることを意味する．8 行目のルールで変電所と直接つながっている区間が上流にあることを表している．9 行目のルールで，ジャンクションノード内の区間が，スイッチによって外部のノードから接続されているならば上流であることを意味している．

また供給経路内の上下関係は，アトム downstream(S,T) で表される．この downstream(S,T) は，区間 S の下流に区間 T が接続されていることを意味する．12 行目のルールで，接続されている 2 つの区間 (S,T) がそのまま上下関係であることを意味している．13 行目のルールでは，entrance_section(S,X) が属するノード X 内にある他の区間 T が下流にあることを意味している．

15,16 行目は，変電所と区間の供給関係を表している．アトム suppliable(T,R) は，区間 T が変電所 R から供給を受けていることを意味する．15 行目のルールでは，各変電所は自身から供給を受けることを表す．16 行目のルールでは，区間 S が変電所 R から供給され，かつ，区間 T が S の下流にあるならば，T も R から供給されることを表す．

18 行目のルールで，各変電所 R について，供給する各区間 S の負荷 I の総和が入力として与えられる max_current 以下でなければならないことを表す．

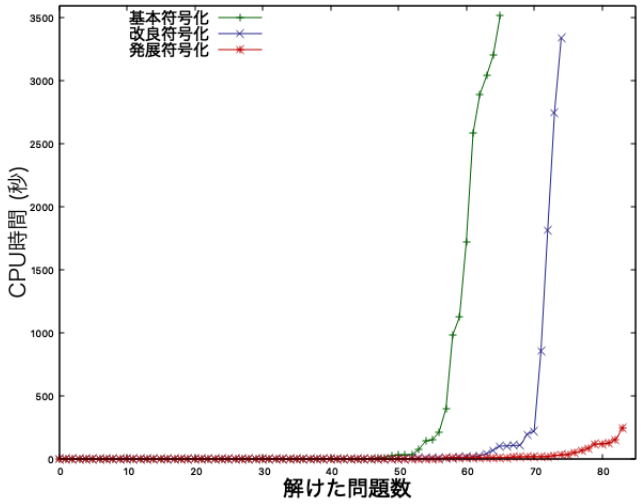

図 7. 基本符号化 (コード 2) と改良符号化 (コード 3) と発展符号化 (コード 4) の比較

5 配電網遷移問題の ASP 符号化

本節では，配電網遷移問題の入力と制約を論理プログラミングとして表現する方法について述べる．

ASP ファクト形式．配電網遷移問題は，もととなる配電網問題の入力と，新たにスタート状態とゴール状態が与えられる．配電網遷移問題のスタート状態とゴール状態の入力 (図 6) のファクト表現をコード 6 に示す．スタート状態における閉じたスイッチは，アトム `init_switch/1` によって表される．また，ゴール状態での閉じたスイッチは，アトム `goal_switch/1` によって表される．

ASP 符号化．配電網遷移問題の ASP 符号化をコード 7 に示す．この符号化は，ASP システム *clingo* のマルチショット ASP 解法ライブラリを用いており，`base`，`step(t)`，`check(t)` の 3 パートから構成される．

`base` パートには，ステップ `t=0` で満たすべき制約を記述する．ここでは 13 行目のルールで，スタート状態とステップ `0` での閉じたスイッチが一致することを強制している．また，17~37 行目のルールは，4 節で示した前処理の ASP 符号化を記述している．

次に `step(t)` パートには，各ステップ `t` において満たすべき制約を記述する．ここでは，根付き全域森問題を解く ASP 符号化のうち，比較実験 (6 節) にて良い性能を示した発展符号化をもとに拡張を行った．発展符号化と電流制約の符号化の各アトムにステップを表す項 `t` を引数として追加したルールを記述している (47~79 行目)．さらに，83~85 行目には，遷移制約を表すルールを記述している．アトム `changed(SW,t)` は，ステップ `t-1` とステップ `t` の間でスイッチ `SW` の状態が変化したことを意味する．85 行目のルールで，各ステップ `t` において，変化したスイッチの数が `d` であることを保証している．

最後に，`step(t)` パートでは，プログラムの終了条件を記述する．ここでは，93 行目でゴール状態とステップ `t` の対応を記述する．なお，`t` がインクリメントされると，一つ前の不要になった終了条件は，`query(t)` の真偽を動的に操作することにより無効化される．

6 実行実験

提案アプローチの有効性を評価するために，節 4 と節 5 の符号化に基づくソルバーを開発し，実行実験を行った．

表 2. 基本符号化 (コード 2) と改良符号化 (コード 3) と発展符号化 (コード 4) の比較 (解けた問題数)

辺の数	問題数	基本符号化	改良符号化	発展符号化
1 〜 1000	30	**30**	**30**	**30**
1001 〜 4000	20	**20**	**20**	**20**
4001 〜 7000	11	9	**10**	**10**
7001 〜 10000	8	4	6	**8**
10001 〜 20000	9	2	5	**9**
20001 〜 30000	2	1	**2**	**2**
30001 〜 40000	1	0	0	**1**
40001 〜 50000	4	0	2	**4**
計	85	66	75	**84**

表 3. 配電網遷移問題の ASP 符号化 (コード **??**) の実行結果

ステップ長 t	問題数	平均 CPU 時間
1	6	1.035
2	62	1.608
3	189	2.155
4	312	2.734
5	280	3.361
6	130	4.165
7	21	5.086
計	1000	

根付き全域森問題. ベンチマークとしては，DNET [6] で公開されている配電網問題 3 問，および，Graph Coloring and its Generalization [7] で公開されているグラフ彩色問題をもとに生成した 82 問[8]を使用した．ベンチマーク問題 (計 85 問) の規模は，ノード数 11〜1406，辺の数 16〜49629，根ノード数 1〜281 である．ASP システムには *clingo*-5.4.0 (*trendy*) を使用し，問題 1 問あたりの制限時間は 1 時間とした．実験環境は，Mac mini，3.2 GHz Intel Core i7，64GB メモリである．

基本符号化と改良符号化と発展符号化の比較結果を図 7 に示す．この図はカクタスプロットと呼ばれ，縦軸が CPU 時間，横軸が解けた問題数を表す．グラフが下に寄るほどより高速に，右に寄るほどより多くの問題を解いたことを意味する．図 7 より，発展符号化は，他 2 つの符号化と比較して，より多くの問題を高速に解いていることがわかる．

表 2 は，解けた問題数を，ベンチマーク問題に含まれる辺の数で分類したものである．発展符号化は，ほぼ全てのベンチマーク問題 (85 問中 84 問) が解けており，大規模な問題に対する有効性が確認できた．

配電網遷移問題. ベンチマークとしては，DNET で公開されている実用規模の配電網問題 (fukui-tepco，ノード数 432，根ノード数 72，電流上限 300) をベースにした．この問題の実行可能解から，スタート状態を 10 個，ゴール状態を 100 個，をランダムに選び，それらを組み合わせた計 1000 問

[6] `https://github.com/takemaru/dnet`

[7] `http://mat.tepper.cmu.edu/COLOR04/`

[8] グラフ彩色問題 127 問の中から，連結グラフで辺の数が 50,000 以下である 82 問を使用した．根については全ノードの 1/5 をランダムに選んで使用した．

の配電網遷移問題を生成した．ASP システムと実験環境は上と同じである．

配電網遷移問題の ASP 符号化 (コード 7) の実行結果を表 3 に示す．左から順に，問題名，解を求めるまでのステップ長，解けた問題数，平均 CPU 時間を示している．今回行った実行実験では，最長でステップ長が 7 の問題を解くことができた．

7　おわりに

本稿では，根付き全域森問題とその解の遷移問題について，解集合プログラミング技術を用いた解法を提案した．根付き全域森問題を解く 2 種類の ASP 符号化を考案した．特に，改良符号化は，根付き全域森問題の連結制約を ASP の個数制約で表現することにより，基礎化後のルール数を少なく抑えるよう工夫されている．解の遷移問題については，マルチショット ASP 解法を利用して解く方法を提案した．この方法は，ASP システムが同様の探索失敗を避けるために獲得した学習節を (部分的に) 保持することで，無駄な探索を行うことなく，制約を追加した問題を連続的に解くことができる．DNET の問題集，および，Graph Coloring and its Generalizations の問題を元に生成した問題集を用いた実験の結果，大規模な根付き全域森問題に対する改良符号化の有効性，遷移問題に対するマルチショット ASP 解法の優位性が確認できた．今後の課題としては，配電網問題の電気制約を，背景理論付き ASP (ASP Modulo Theories) を用いて実装することが挙げられる．

```
%% test.yaml (dnet format)
%% 37 nodes, 25 sections, 3 substations 16 switches
dnet_node(1, ("section_-001"; "section_0302"; "section_0303")).
dnet_node(2, ("section_-002"; "section_0001"; "section_0002")).
dnet_node(3, ("section_-003"; "section_0008"; "section_0309")).
dnet_node(4, ("section_0302"; "switch_0010")).
dnet_node(5, ("section_0300"; "switch_0010")).
dnet_node(6, ("section_0298"; "section_0299"; "section_0300")).
dnet_node(7, ("section_0299"; "switch_0006")).
dnet_node(8, ("section_1057"; "switch_0006")).
dnet_node(9, ("section_1057"; "switch_0004")).
dnet_node(10, ("section_0298"; "switch_0009")).
dnet_node(11, ("section_0296"; "switch_0009")).
dnet_node(12, ("section_0296"; "switch_0008")).
dnet_node(13, ("section_0294"; "switch_0008")).
dnet_node(14, ("section_0294"; "switch_0007")).
dnet_node(15, ("section_0001"; "switch_0001")).
dnet_node(16, ("section_1063"; "switch_0001")).
dnet_node(17, ("section_1061"; "section_1062"; "section_1063")).
dnet_node(18, ("section_1061"; "switch_0002")).
dnet_node(19, ("section_1059"; "switch_0002")).
dnet_node(20, ("section_1059"; "switch_0004")).
dnet_node(21, ("section_1062"; "switch_0003")).
dnet_node(22, ("section_1066"; "switch_0003")).
dnet_node(23, ("section_1066"; "switch_0005")).
dnet_node(24, ("section_1068"; "switch_0005")).
dnet_node(25, ("section_1068"; "switch_0007")).
dnet_node(26, ("section_0002"; "switch_0011")).
dnet_node(27, ("section_0004"; "switch_0011")).
dnet_node(28, ("section_0004"; "switch_0012")).
dnet_node(29, ("section_0008"; "switch_0013")).
dnet_node(30, ("section_0006"; "switch_0013")).
dnet_node(31, ("section_0006"; "switch_0012")).
dnet_node(32, ("section_0309"; "switch_0016")).
dnet_node(33, ("section_0307"; "switch_0016")).
dnet_node(34, ("section_0307"; "switch_0015")).
dnet_node(35, ("section_0303"; "switch_0014")).
dnet_node(36, ("section_0305"; "switch_0014")).
dnet_node(37, ("section_0305"; "switch_0015")).
load("section_-001", 16).
load("section_-002", 41).
load("section_-003", 16).
load("section_0001", 31).
load("section_0002", 0).
load("section_0004", 20).
load("section_0006", 20).
load("section_0008", 0).
load("section_0294", 34).
load("section_0296", 11).
load("section_0298", 34).
load("section_0299", 0).
load("section_0300", 5).
load("section_0302", 40).
load("section_0303", 0).
load("section_0305", 35).
load("section_0307", 35).
load("section_0309", 0).
load("section_1057", 21).
load("section_1059", 20).
load("section_1061", 0).
load("section_1062", 24).
load("section_1063", 45).
load("section_1066", 45).
load("section_1068", 24).
substation("section_-002").
substation("section_-001").
substation("section_-003").
switch("switch_0001").
switch("switch_0002").
switch("switch_0003").
switch("switch_0004").
switch("switch_0005").
switch("switch_0006").
switch("switch_0007").
switch("switch_0008").
switch("switch_0009").
switch("switch_0010").
switch("switch_0011").
switch("switch_0012").
switch("switch_0013").
switch("switch_0014").
```

```
%% solution candidates
{ inForest(X,Y) } :- edge(X,Y).

%% reachability
reached(R,R) :- root(R).
reached(X,R) :- reached(Y,R), inForest(Y,X).
reached(X,R) :- reached(Y,R), inForest(X,Y).

%% acyclicity constraint
:- root(R),
   not 1 #sum{ 1,X:reached(X,R) ;
               -1,X,Y:inForest(X,Y),reached(X,R),reached(Y,R)
             } 1.

%% rooted connectivity constraint
:- node(X),  not reached(X,_).
:- reached(X,R1), reached(X,R2), R1 < R2.
```

コード 2. 根付き全域森問題の基本符号化

```
%% solution candidates
{ inForest(X,Y) } :- edge(X,Y).

%% reachability
reached(R,R) :- root(R).
reached(X,R) :- reached(Y,R), inForest(Y,X).
reached(X,R) :- reached(Y,R), inForest(X,Y).

%% acyclicity constraint
:- root(R),
   not 1 #sum{ 1,X:reached(X,R) ;
               -1,X,Y:inForest(X,Y),reached(X,R),reached(Y,R)
             } 1.

%% rooted connectivity constraint
:- node(X), not 1 { reached(X,R) } 1.
```

コード 3. 根付き全域森問題の改良符号化

```
%% solution candidates
{ inForest(X,Y); inForest(Y,X) } 1 :- edge(X,Y).

%% reachability
reached(R,R) :- root(R).
reached(X,R) :- reached(Y,R), inForest(Y,X).

%% acyclicity constraint
:- root(R), inForest(_,R).
:- node(X), not root(X), not 1 { inForest(_,X) } 1.

%% rooted connectivity constraint
:- node(X), not 1 { reached(X,R) } 1.
```

コード 4. 根付き全域森問題の発展符号化

```
%%% electrical constraints

connected(SW,S,T) :- inForest(X,Y), node(X,S), node(Y,T),
                     switch(SW,S,T).

closed_switch(SW) :- connected(SW,_,_).

entrance_section(S,X) :- substation(S), section(S,X).
entrance_section(S,X) :- jct_node(X), section(S,X),
                         connected(_,_,S).

downstream(S,T) :- connected(_,S,T).
downstream(S,T) :- entrance_section(S,X), section(T,X), S!=T.

suppliable(R,R) :- substation(R).
suppliable(T,R) :- downstream(S,T), suppliable(S,R).

:- substation(R),
   not #sum{ I,S:suppliable(S,R), load(S,I) } max_current.
```

コード 5. 電流制約の符号化

```
init_switch("switch_0001").
init_switch("switch_0002").
...
init_switch("switch_0016").

goal_switch("switch_0001").
goal_switch("switch_0002").
...
goal_switch("switch_0015").
```

コード 6. 配電網遷移問題 (図 6) のファクト表現

```
%%% require: recongo/core.lp
#const core_stop = "SAT".
#const core_min = 0.

#const d = 2.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
#program base.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% init
:- not closed_switch(SW,0), init_switch(SW).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% Pre-processing
swt_node(X) :- dnet_node(X,S), switch(S).
jct_node(X) :- dnet_node(X,_), not swt_node(X).

section(S,X) :- dnet_node(X,S), not switch(S).
section(S)   :- section(S,_).

jct_section(S) :- section(S,X), jct_node(X).
swt_section(S) :- section(S,X), swt_node(X).

switch(SW,S,T) :- section(S,X), section(T,Y), S!=T,
                      dnet_node(X,SW), dnet_node(Y,SW), switch(SW).

%% spanning rooted tree
node(X,  S) :- jct_section(S), dnet_node(X,S), jct_node(X).
node(Min,S) :- swt_section(S), not jct_section(S),
               Min = #min { X : dnet_node(X,S) }.

root(X) :- node(X,S), substation(S).
node(X) :- node(X,_).

edge(X,Y) :- node(X,S), node(Y,T), X<Y, switch(SW,S,T).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
#program step(t).
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% spanning rooted forest

%% choose edge
{ inForest(X,Y,t); inForest(Y,X,t) } 1 :- edge(X,Y).

%% tree constraint
:- root(R), inForest(_,R,t).
:- node(X), not root(X), not 1 { inForest(_,X,t) } 1.

%% connectivity constraint
:- node(X), not 1 { reached(X,R,t) } 1.

%% generate reached
reached(R,R,t) :- root(R).
reached(X,R,t) :- reached(Y,R,t), inForest(Y,X,t).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% electrical constraints
connected(SW,S,T,t) :- inForest(X,Y,t), node(X,S), node(Y,T),
                       switch(SW,S,T).

closed_switch(SW,t) :- connected(SW,_,_,t).

entrance_section(S,X,t) :- substation(S), section(S,X).
entrance_section(S,X,t) :- jct_node(X), section(S,X),
                           connected(_,_,S,t).

downstream(S,T,t) :- connected(_,S,T,t).
downstream(S,T,t) :- entrance_section(S,X,t), section(T,X), S!=T.

suppliable(R,R,t) :- substation(R).
suppliable(T,R,t) :- downstream(S,T,t), suppliable(S,R,t).

%% current constraint
:- substation(R),
   not #sum{ I,S:suppliable(S,R,t), load(S,I) } max_current.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```