プログラミング-言語・基礎・実践- 16－20
(1994. 3. 10)

C-daemonでのリングの方向付けのための自己安定アルゴリズムについて

桝田 秀夫, 片山 喜章, 増澤 利光, 都倉 信樹

大阪大学基礎工学部情報工学科

〒560 大阪府豊中市待兼山町 1-1

E-mail: h-masuda@ics.es.osaka-u.ac.jp

あらまし ネットワークの取り得る全状況集合をSとし, そのうち望ましい状態とみなせる状態を正当な状況といい, $\mathcal{L}$と表す($\mathcal{L} \subseteq S$). 任意のSの状態から出発し有限時間内に$\mathcal{L}$に含まれる状態に達する分散アルゴリズムを自己安定アルゴリズムという. 本論文では, リング上のネットワークでリングの方向付け問題を解く自己安定分散アルゴリズムを考える. つまり, 正当な状況$\mathcal{L}$を, すべてのプロセッサが同一方向に方向付けられている状態とする. また, $\mathcal{L}$からわずかの状態変化で, $\mathcal{L}$に含まれない状態になることを小変動と呼ぶものとする. 小変動に対して回復のはやいアルゴリズムが一般に有効であると考えられる. 本論文では2つのアルゴリズムについて, 小変動からの回復速度の観点から比較評価した.

和文キーワード 分散アルゴリズム, 自己安定, 小変動, リングネットワーク, リングネットワークの方向付け問題

Self-Stabilizing Algorithms for Ring Orientation Problem under C-daemon

Hideo Masuda, Yoshiaki Katayama, Toshimitsu Masuzawa, Nobiki Tokura

Faculty of Engineering Science, Osaka University

Toyonaka, Osaka 560, Japan

E-mail: h-masuda@ics.es.osaka-u.ac.jp

Abstract Let S be the set of all network configurations and let $\mathcal{L}$ be the set of all legal configurations($\mathcal{L} \subseteq S$). A distributed algorithm is said to be self-stabilizing if starting from an arbitrary configuration of S, it can leads into the configuration in $\mathcal{L}$ within some finite number of steps, and keep it in. The ring orientation problem(ROP) is to orient the direction of each processor, clockwise or counter-clockwise in the ring network. Two $O(n^2)$ algorithms for ROP have been designed where n is the number of processors. In general, such an algorithm which can quickly recover from a small disturbance should be desirable. Two algorithms are compared in this respect.

key words Distributed algorithm, Self-Stabilizing, Small disturbance, Ring network, Ring orientation problem

1. はじめに

通信用リンクで接続されたネットワーク上の複数のプロセッサが，メッセージを交換しながら協調して問題を解くアルゴリズムを，分散アルゴリズム (distributed algorithm) という．これに対し，任意のネットワーク状況から始めても解を求めて安定する分散アルゴリズムを自己安定アルゴリズム (self-stabilizing algorithm) という．自己安定アルゴリズムの長所には次の3つがあげられる．

1. 各プロセッサ，各通信リンクに対して，いかなる初期化も必要としない．
2. アルゴリズム実行中にプロセッサの持つデータの破壊 (プログラムカウンタの値の破壊も含む) など「一時故障 (transient failure)」(ローカルには検出できない) が起こっても，その後十分に長い間故障が起こらなければ問題を解くことができる．
3. アルゴリズム実行中にネットワーク形状が変化しても，その後十分長い間形状変化が起こらなければ問題を解くことができる．

プロセッサ間の通信を各リンクの各通信方向に用意された通信用レジスタを介して行なうモデル (レジスタ通信モデル) を考える場合，自己安定アルゴリズムを特徴付ける要因として，プロセッサの原子動作，および同時に動作するプロセッサの個数に対する仮定によって，C デーモン (Central daemon)，D デーモン (Distributed daemon)，R/W デーモン (Read/Write daemon) の 3 つのデーモンが考えられている．

- C デーモン (Central daemon)
  「同時に 1 つのプロセッサしか動作しない」かつ「1 原子動作で，全隣接レジスタから情報を読み出し，自分の状態を変化させ，必要があれば全隣接レジスタへの書き込みを行なう」．もっとも強いモデルである．
- D デーモン (Distributed daemon)
  「同時に複数のプロセッサが動作できる」かつ「1 原子動作で，全隣接レジスタから情報を読み出し，自分の状態を変化させ，必要があれば全隣接レジスタへの書き込みを行なう」．
- R/W デーモン (Read/Write daemon)
  「同時に 1 つのプロセッサしか動作できない」かつ「1 原子動作で，1 つのレジスタからの読み込みと内部状態の変化，あるいは，1 つのレジスタへの書き込みと内部状態の変化ができる」．もっとも弱いモデルである．

リングの方向付け問題 (ROP) とは，リング (ネットワーク) 上のすべてのプロセッサを同一方向 (例えば時計回り方向) に方向付ける問題である．これまでに，偶数サイズの均一な (すべてのプロセッサが対等で，識別子も存在しない) リングに対しては，D デーモンの下で ROP を解く決定性アルゴリズムが存在しないことが知られている [1]．また，D デーモンの下で，任意サイズの均一なリングで ROP を解く確率的な自己安定アルゴリズムが提案されている [1]．さらに，C デーモンの下で，任意サイズの均一なリングで ROP を解く決定性アルゴリズム，R/W デーモンの下で，奇数サイズの均一なリングで ROP を解く決定性自己安定アルゴリズムが [2] で示されている．

一般の分散アルゴリズムは，解を求めて停止するが，自己安定アルゴリズムは，その性質から，解を求めてもアルゴリズムの実行は停止しない．このような特徴により，一度解を求めた状態から一時故障やネットワーク形状の変化が生じても解を求めることができる．つまり，長期に渡ってネットワーク状況を安定に保ち，一時故障やネットワーク形状の変化に柔軟に対応することの求められるシステムを動作させる場合に適しており，その場合，安定状態からかけはなれた状態からの動作ではなく，安定状態から少し状態変化した時点からのアルゴリズムの動作が主に行なわれると考えられる．本論文では，自己安定アルゴリズムの安定性の指標として，安定状態から少し (ある 1 つのプロセッサだけが) 変化した状態になること (**小変動 (small Disturbance)**) を提案し，小変動に対して有効な，C デーモンの下で，任意サイズの均一なリング上で ROP を解く決定性自己安定アルゴリズムを提案した．これは，[2] と比較して，同時に複数のレジスタの値を比べることによって，オーダ的には変わらないが，小変動に対して実質的に速く安定するアルゴリズムになっている．

2. 諸定義

ここでは，本論文で扱うネットワークやアルゴリズムなどのモデルとその定義について述べる．

2.1 ネットワークに関する定義

本論文では，n 個のプロセッサが通信リンクによってリング状に接続された均一な (uniform) リングネットワーク N_n を扱う．均一な (uniform) ネットワークとは，すべてのプロセッサが識別子を持たない同一の状態機械 (同一のプログラムを実行する) であり，識別子等をプログラムで用いることのできないネットワークである．

リングネットワーク　N_n 中のプロセッサを仮に $P_0, P_1, \ldots, P_{n-1}$ とし，その集合を $\mathcal{P}$ とする．N_n において，プロセッサ P_i は $P_{i-1}, P_{i+1} (i-1, i+1$ は mod n で考える) との間に通信リンク $L(i, i-1), L(i, i+1)$ をもつ．リンクでつながれた 2 つのプロセッサは**隣接している**という．

各プロセッサで動作させるプログラムを**アルゴリズム**という．本論文では，均一なネットワークを扱うので，すべてのプロセッサで同一のアルゴリズムを実行する．以降，便宜上プロセッサを，$P_0, P_1, \ldots, P_{n-1}$ と表しているが，添字を識別子としてアルゴリズムで用いることはない．また，各プロセッサは，2 つの隣接プロセッサのどちらか一方を**第 1 プロセッサ** $P_i(1)$，他方を**第 2 プロセッサ** $P_i(2)$ として区別し，アルゴリズム実行中は変化しないものとする．ただし，$P_i(1)$ が実際には P_{i-1} あるいは P_{i+1} なのかは，ここで考えるアルゴリズムでは知り得ないものとする．

動作するアルゴリズムは，隣接プロセッサと通信リンクを用いて情報交換を行なうが，その際の通信モデルとしては，レジスタ通信モデルで考える．このモデルは，各リンクの各方向に対して 1 つの通信用レジスタが存在し，そこに隣接プロセッサへのメッセージを書き込み，それを他方のプロセッサが読み出すことによって通信を実現するもので

ある．つまり，N_n中のリンク $L(i,j)$ は2つの通信用レジスタ $R(i,j), R(j,i)$ をもち，P_iが $R(i,j)$ に書き込んだ値を P_jが読み出すことによって，P_iから P_jへの通信が行なえ，同様に $R(j,i)$ を用いて P_jから P_iへの通信が行なえる．均一なネットワークでは，各プロセッサ P_iは，隣接プロセッサの識別子が分からないので，$P_i(1)$ に対する読み出し (書き込み) レジスタを $RD(1)(WR(1))$，同様に，$P_i(2)$ に対するレジスタを，$RD(2)(WR(2))$ と参照する．

2.2 アルゴリズムの実行に関する定義

アルゴリズムの実行を表すために以下のものを考える．

スケジュール 空ではないプロセッサの部分集合の無限系列．つまり，プロセッサ集合 $\mathcal{P}$に対して，

$$S = \mathcal{Q}_0, \mathcal{Q}_1, \ldots, \mathcal{Q}_i, \ldots (\forall i, \mathcal{Q}_i \subseteq \mathcal{P}, \mathcal{Q}_i \neq \phi)$$

ネットワーク状況 プロセッサ P_iの取り得るプログラムカウンタの値と内部変数の値からなる状態の集合を Q_i とする．本論文では，均一なネットワークを考えるので，$\forall i, Q_i = Q$ としてよい．ある時点で N_nを見たとき，プロセッサ P_iの状態が $q_i \in Q$，レジスタ $R(i,j)$ の状態が $r_{ij} \in R$(Rはレジスタが取り得るすべての状態) となるとき，

$$c = (q_0, q_1, \ldots, q_{n-1}, r_{01}, r_{0(n-1)}, r_{12}, r_{10}, \ldots, r_{(n-1)0}, r_{(n-1)(n-2)})$$

を，その時点での N_nのネットワーク状況と呼ぶ．N_n の取り得るすべてのネットワーク状況の集合を Cとすると，$C = Q^n \times R^{2n}$ である．

定義 1 (アルゴリズムの実行) $\mathcal{A}$ を分散アルゴリズム，プロセッサ集合 $\mathcal{P}$の任意の部分集合を $\mathcal{Q}$ とする．$\mathcal{Q}$ に属するすべてのプロセッサが $\mathcal{A}$ によって決まる状態遷移関数にしたがって同時に状態を変えて，ネットワークの状況が c_iから c_jに変わることを，$c_i \rightarrow (\mathcal{Q}, \mathcal{A})c_j$ と表す．$S = \mathcal{Q}_0, \mathcal{Q}_1, \mathcal{Q}_2, \ldots$ を任意のスケジュールとする．このとき，ネットワーク状況の無限系列 $E = c_0, c_1, c_2, \ldots$ が，各 $i(0 \leq i)$ について，$c_i \rightarrow (\mathcal{Q}_i, \mathcal{A})c_{i+1}$ を満たすならば，Eを「初期状況 c_0，スケジュール Sに対するアルゴリズム $\mathcal{A}$ の実行」と呼ぶ． □

スケジュール Sに，ネットワークのすべてのプロセッサが無限回現れるとき，Sは**公平 (fair)** であるという．また，実行 Eのスケジュール S が公平なとき，実行 Eは公平であるという．

2.3 自己安定性に関する定義

定義 2 (自己安定アルゴリズム) $\mathcal{L} \subseteq C$ をネットワーク状況の任意の集合とする．また，X デーモンによって決まるスケジュールの集合を $T(X)$ と表す．次の *1*, *2*の条件を満たすとき，「アルゴリズム $\mathcal{A}$ は，X の下で $\mathcal{L}$ に関して自己安定である」といい，$SS(\mathcal{A}, \mathcal{L}, X)$ と書く．また，$SS(\mathcal{A}, \mathcal{L}, X)$ が成立するとき，$\mathcal{L}$ を「$(\mathcal{A}, X)$ に関して正当な状況」という．ただし，$\mathcal{A}, X$ が明らかな場合，単に正当な状況，または解状況という．

1. 到達可能性
   任意の初期ネットワーク状況 $c_0 \in C$と c_0から始まる，アルゴリズム $\mathcal{A}$ によるスケジュール $T \in T(X)$ の実行 $E_i(\mathcal{A}, T, C)$ の列中に，いつかはネットワーク状況 $c \in \mathcal{L}$ があらわれる．すなわち，実行 $E_i = c_0, c_1, \ldots, c_j, \ldots$ について，$c_j \in \mathcal{L}$ なる $j \geq 0$ が存在する．
2. 閉包性
   任意のネットワーク状況 $c \in \mathcal{L}$ から始まる任意の実行 $E_i = c_0, c_1, \ldots$ の要素であるすべてのネットワーク状況 c_jについて，$\forall j(j \geq 0), c_j \in \mathcal{L}$ である．

つまり任意の初期状況から開始しても，アルゴリズム $\mathcal{A}$ による任意の公平なスケジュールの実行は，有限時間内に正当な状況に到達し，かつ一度正当な状況に達すると，それ以降正当な状況のままということである． □

仮定 1 本論文では，公平なスケジュール，かつ，Cデーモンの下で動作する自己安定アルゴリズムを扱う． □

スケジュールが公平でない場合，ある時点以降全く動作しないプロセッサが存在し，アルゴリズムが解を求められないことがある．このため，スケジュールの公平さは最低限満たさなければならない条件である．

2.4 リングの方向付け問題に関する定義

均一なリングネットワーク N_nにおいて，リングの方向付け問題 (ROP) を考える．

定義 3 (方向付けられたリングネットワーク) 各プロセッサ P_iは，アルゴリズムを実行して第1プロセッサか第2プロセッサのいずれか一方を *head*，他方を *tail* と決める．P_iによって *head*,*tail* と指定されたプロセッサをそれぞれ $head(P_i)$,$tail(P_i)$ と表す．このとき，すべてのプロセッサ $P_i \in P$について，

$$tail(head(P_i)) = P_i$$

が成立するとき，リングネットワークは**方向付けられた**という． □

アルゴリズム A が，正当な状況に属する任意のネットワーク状況，すなわち $c \in \mathcal{L}$ なる任意のネットワーク状況 c が定義 3 の条件を満たし，かつ，定義 2を満たすとき，アルゴリズム A を「リングの方向付け問題を解く自己安定アルゴリズム」という．

3. アルゴリズム

C デーモンの下で任意の N_nについてROP を解く決定性自己安定アルゴリズム A_Cを示す．

3.1 利用する変数

A_Cにおいて，各プロセッサ P_iは，以下の変数を持つ．

$D(P_i)$: プロセッサの現在の向きを表し，1,2 のいずれかの値を取る．第1プロセッサ側を向いている場合は1，第2プロセッサ側を向 いている場合は2 が入る．

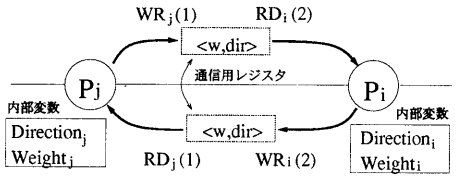

図 1: 通信用レジスタの例

$W(P_i)$: プロセッサに付けられた自然数の重み.

プロセッサ P_iに対する読み出しレジスタを $RD_i(1)$, $RD_i(2)$, 書き込みレジスタを $WR_i(1), WR_i(2)$ とする (図1). P_i は, $WR_i(1), WR_i(2)$ に対して, $\langle w, dir\rangle$ を書き込む. ただし, w, dirは以下に示す値である.

w : プロセッサが最後に計算した $W(P_i)$ の値.

dir : プロセッサがどちらを向いているかのフラグ. プロセッサ P_iは, $D(P_i)$ を元に,

$$D(P_i) = 1 \Rightarrow WR_i(1).dir = 1, WR_i(2).dir = 2$$

$$D(P_i) = 2 \Rightarrow WR_i(1).dir = 2, WR_i(2).dir = 1$$

と書き込む.

ここで, これらの変数の値によってトークンを定義する.

定義 4 (トークン) 任意のプロセッサ Pと, $Q = head(P)$ を考える. プロセッサ Pは, 以下の条件 *1,2*のいずれかを満たすとき, *(*Q への*)* トークンを持つという.

1. $head(Q) = P$ かつ $W(P) \geq W(Q)$
2. $tail(Q) = P$ かつ $W(P) > W(Q)$ □

プロセッサ Pがプロセッサ Q へのトークンをもつとき, そのトークンが Q に移ることを, トークンが前進するという.

また, 証明や効率の評価のために, 関数を導入する.

定義 5 (全有効距離 $f(c)$) あるネットワーク状況 c を考える. 各プロセッサ Pi に対して, $Pi_0(= Pi), Pi_1, Pi_2, \ldots, Pi_k(Pi_1 = head(Pi_0), Pi_1, Pi_2, \ldots, Pi_k$ は, リングネットワーク上で順につながっているものとする*)* を考える. このとき, 有効距離 $f_{Pi}(c)$ を以下のように定義する.

$$f_{Pi}(c) = k(\forall j(0 < j \leq k), W(Pi_0) > W(Pi_j)$$

f_{Pi}は, プロセッサ Pi からトークンが前進していったと仮定するとき, そのトークンが消滅せずに進むと期待できるプロセッサ数と考えられる. そして, すべてのプロセッサでの有効距離 $f_{Pi}(c)$ の総和をネットワーク状況 c に対する**全有効距離** $f(c)$ とする.

$$f(c) = \sum_{\forall Pi \in \mathcal{P}} f_{Pi}(c)$$

□

3.2 状態遷移関数

プロセッサ P_iの状態遷移を表 1の A_Cで示す.

プロセッサ P_iが $RD(1), RD(2)$ を読み出し, 動作前の状況を判断する. そして, 表 1の動作前のパターンに合う場合を探し, 動作後 (A_C) へ状態を変化させ, 結果を $WR(1), WR(2)$ に書き込む.

表 1において, プロセッサ $P_i(h) = head(P_i)$, $P_i(t) = tail(P_i)$ とする. 表 1の説明で, 記号の上についている矢印はそのプロセッサの向いている方向とし, また, プロセッサの重みの間に関する条件を, プロセッサとプロセッサの間に不等号をもって表す. さらに, プロセッサが [] で囲まれているものは, トークンを持つ. 表中において, トークン数と $f(c)$ 値は, それぞれ, プロセッサ P_iの動作前と動作後のネットワーク状況 c_b, c_aの間で, トークン数の変化, および, $f(c_b)$ と $f(c_a)$ の値を比較した結果を示す. また, 非動作となる場合は $W(P_i), D(P_i)$ の値を変更しないことを示す.

3.3 アルゴリズム A_C

プロセッサ $P_i(0 \leq i < n)$ の動作を以下に示す.

```
/* Algorithm A_C for Ring Orientation Problem */
/*                under C-daemon                */
do forever
begin /* 原子動作の開始 */
      r_f := read(RD(head(P_i)));
            /* head(P_i) の情報を読み出す */
      r_b := read(RD(tail(P_i)));
            /* tail(P_i) の情報を読み出す */
      表 1の状態遷移表に従い,
            W(P_i), D(P_i) を変更する.
      write (WR(head(P_i)), ⟨W(P_i), 1⟩);
            /* head(P_i) に情報を書き込む */
      write (WR(tail(P_i)), ⟨W(P_i), 2⟩);
            /* tail(P_i) に情報を書き込む */
end /* 原子動作の終了 */
```

4. アルゴリズム A_Cの正当性

3.節で述べたアルゴリズム A_Cが, ROP を解く自己安定アルゴリズムであることを示す.

事実 1 任意のプロセッサ P_iがアルゴリズム A_Cの原子動作を 1 回以上実行すると, 次の条件が成り立つ.

- $D(P_i) = 1 \Rightarrow$
  $WR(1) = \langle W(P_i), 1\rangle, WR(2) = \langle W(P_i), 2\rangle$
- $D(P_i) = 2 \Rightarrow$
  $WR(1) = \langle W(P_i), 2\rangle, WR(2) = \langle W(P_i), 1\rangle$ □

以下では事実 1の条件が成立しているものとする.

次に, アルゴリズム A_Cに関する解状況を定義する.

定義 6 解状況の集合 $\mathcal{L}_{ROP}$を, 以下の *2*条件を満たすすべてのネットワーク状況からなる集合とする.

1. 各プロセッサ $P_i(0 \leq i < n)$ で, $tail(head(P_i)) = P_i$ が成立する. つまり, 定義 *3*を満たす.

	動作前			動作後 (A_C)	動作後 (A_{mk})	トークン数	$f(c)$ 値
1.	$P_i(h) < [P_i] < [P_i(t)]$		$\Rightarrow$	$P_i(h) < [P_i] = P_i(t)$	左に同じ	$-1/-1$	減少/減少
2.	$P_i(h) \geq P_i < [P_i(t)]$	$w(P_i(h)) < w(P_i(t))$	$\Rightarrow$	$P_i(h) < [P_i] = P_i(t)$	左に同じ	$\pm 0/\pm 0$	減少/減少
3.	$P_i(h) \geq P_i < [P_i(t)]$	$w(P_i(h)) \geq w(P_i(t))$	$\Rightarrow$	$P_i(h) = P_i \geq P_i(t)$	左に同じ	$-1/-1$	減少/減少
4.	$P_i(h) < [P_i] < [P_i(t)]$		$\Rightarrow$	$P_i(h) < [P_i] = P_i(t)$	左に同じ	$-1/-1$	減少/減少
5.	$[P_i(h)] = [P_i] < [P_i(t)]$	$w(P_i)+1 = w(P_i(t))$ かつ, $P_i(1) = P_i(h)$	$\Rightarrow$	$P_i(h) < [P_i] = P_i(t)$	$P_i(h) < [P_i] > P_i(t)$ $w(P_i) \leftarrow w(P_i(h))+1$	$-2/-2$	減少/減少
6.	$[P_i(h)] = [P_i] < [P_i(t)]$	上記以外の場合	$\Rightarrow$	$P_i(h) < [P_i] = P_i(t)$	左に同じ	$-2/-2$	減少/減少
7.	$[P_i(h)] > P_i < [P_i(t)]$	$w(P_i(h)) < w(P_i(t))$	$\Rightarrow$	$P_i(h) < [P_i] = P_i(t)$	左に同じ	$-1/-1$	減少/減少
8.	$[P_i(h)] > P_i < [P_i(t)]$	$w(P_i(h)) = w(P_i(t))$	$\Rightarrow$	$P_i(h) < [P_i] > P_i(t)$ $w(P_i) \leftarrow w(P_i(h))+1$	左に同じ 左に同じ	$-1/-1$	$+O(n)/+O(n)$
9.	$[P_i(h)] > P_i < [P_i(t)]$	$w(P_i(h)) > w(P_i(t))$	$\Rightarrow$	$P_i(h) = [P_i] > P_i(t)$	左に同じ	$-1/-1$	減少/減少
10.	$P_i(h) < [P_i] \geq P_i(t)$	$w(P_i(h)) < w(P_i(t))$	$\Rightarrow$	$P_i(h) < [P_i] \geq P_i(t)$	左に同じ	非動作	
11.	$P_i(h) < [P_i] \geq P_i(t)$	$w(P_i(h)) \geq w(P_i(t))$	$\Rightarrow$	$P_i(h) = P_i \geq P_i(t)$	$P_i(h) < [P_i] \geq P_i(t)$	-1/非動作	$+O(n^2)$/非動作
12.	$P_i(h) \geq P_i \geq P_i(t)$		$\Rightarrow$	$P_i(h) \geq P_i \geq P_i(t)$	左に同じ	非動作	
13.	$P_i(h) < [P_i] \geq P_i(t)$		$\Rightarrow$	$P_i(h) < [P_i] \geq P_i(t)$	左に同じ	非動作	
14.	$[P_i(h)] = [P_i] \geq P_i(t)$		$\Rightarrow$	$P_i(h) < [P_i] > P_i(t)$ $w(P_i) \leftarrow w(P_i(h))+1$	左に同じ 左に同じ	$-1/-1$	$+O(n)/+O(n)$
15.	$[P_i(h)] > P_i \geq P_i(t)$		$\Rightarrow$	$P_i(h) = [P_i] > P_i(t)$	左に同じ	$\pm 0/\pm 0$	減少/減少
16.	$P_i(h) < [P_i] < P_i(t)$		$\Rightarrow$	$P_i(h) < P_i = P_i(t)$	$P_i(h) < [P_i] < P_i(t)$	-1/非動作	減少/非動作
17.	$P_i(h) \geq P_i < P_i(t)$		$\Rightarrow$	$P_i(h) \geq P_i < P_i(t)$	左に同じ	非動作	
18.	$P_i(h) < [P_i] < P_i(t)$		$\Rightarrow$	$P_i(h) < P_i = P_i(t)$	$P_i(h) < [P_i] < P_i(t)$	-1/非動作	減少/非動作
19.	$[P_i(h)] = [P_i] < P_i(t)$		$\Rightarrow$	$P_i(h) < P_i = P_i(t)$	$P_i(h) < P_i \leq P_i(t)$ $w(P_i) \leftarrow w(P_i(h))+1$	$-2/-2$	減少/減少
20.	$[P_i(h)] > P_i < P_i(t)$	$w(P_i(h)) \leq w(P_i(t))$	$\Rightarrow$	$P_i(h) \leq P_i = P_i(t)$	$P_i(h) = P_i \leq P_i(t)$	$-1/-1$	減少/減少
21.	$[P_i(h)] > P_i < P_i(t)$	$w(P_i(h)) > w(P_i(t))$	$\Rightarrow$	$P_i(h) = [P_i] > P_i(t)$	左に同じ	$\pm 0/\pm 0$	減少/減少
22.	$P_i(h) < [P_i] \geq P_i(t)$	$w(P_i(h)) \leq w(P_i(t))$	$\Rightarrow$	$P_i(h) < [P_i] \geq P_i(t)$	左に同じ	非動作	
23.	$P_i(h) < [P_i] \geq P_i(t)$	$w(P_i(h)) > w(P_i(t))$	$\Rightarrow$	$P_i(h) = P_i > P_i(t)$	$P_i(h) < [P_i] \geq P_i(t)$	-1/非動作	$+O(n^2)$/非動作
24.	$P_i(h) \geq P_i \geq P_i(t)$		$\Rightarrow$	$P_i(h) \geq P_i \geq P_i(t)$	左に同じ	非動作	
25.	$P_i(h) < [P_i] \geq P_i(t)$	$w(P_i(h)) \leq w(P_i(t))$	$\Rightarrow$	$P_i(h) \leq P_i = P_i(t)$	$P_i(h) < [P_i] \geq P_i(t)$	-1/非動作	$+O(n^2)$/非動作
26.	$P_i(h) < [P_i] \geq P_i(t)$	$w(P_i(h)) > w(P_i(t))$	$\Rightarrow$	$P_i(h) < [P_i] \geq P_i(t)$	左に同じ	非動作	
27.	$[P_i(h)] = [P_i] > P_i(t)$	$w(P_i(h)) > w(P_i(t))$	$\Rightarrow$	$P_i(h) = [P_i] > P_i(t)$ $w(P_i) \leftarrow w(P_i(h))+1$	$P_i(h) < [P_i] > P_i(t)$ 左に同じ	$-1/-1$	$+O(n)/+O(n)$
28.	$[P_i(h)] = [P_i] = P_i(t)$	$w(P_i(h)) = w(P_i(t))$	$\Rightarrow$	$P_i(h) = P_i = P_i(t)$	$P_i(h) < [P_i] > P_i(t)$ $w(P_i) \leftarrow w(P_i(h))+1$	$-2/-1$	減少/$+O(n)$
29.	$[P_i(h)] > P_i \geq P_i(t)$		$\Rightarrow$	$P_i(h) = [P_i] > P_i(t)$	左に同じ	$\pm 0/\pm 0$	減少/減少

表 1: アルゴリズム A_C と A_mk の状態遷移表

2. すべてのプロセッサ Pの内部変数 $W(P)$ の値が等しい. □

$\mathcal{L}_{ROP}$に属する任意のネットワーク状況は, 定義 3 を満たすので ROP の解である. 従って, 以下では $SS(A_C, \mathcal{L}_{ROP}, C)$ が成立すること, すなわち, アルゴリズム A_C が $\mathcal{L}_{ROP}$に関して自己安定であることを示す.

ネットワーク状況 c_iに対して, ネットワーク中のトークン数 $T(c_i)$ と全有効距離 $f(c_i)$ の 2 項組関数 $\langle T(c_i), f(c_i)\rangle$ を考える. この 2 項組関数 $\langle T(c_i), f(c_i)\rangle$ に対して, 以下の補題が成立する.

補題 1 ネットワーク状況 cが, 解状況であることの必要十分条件は, $\langle T(c), f(c)\rangle = \langle 0, 0\rangle$ である.

(証明) ネットワーク状況 c_sが解状況ならば, 定義 6より, 各プロセッサ P_iで, $tail(head(P_i)) = P_i$であり, かつ, すべてのプロセッサの内部変数 $W(P_i)$ の値が等しい. また, 定義 4のトークンの定義より, 各プロセッサ P_i で $tail(head(P_i)) = P_i$であれば, $W(P_i) > W(head(P_i))$ であるときのみトークンを持つ. しかし, すべてのプロセッサの内部変数 $W(P_i)$ の値が等しいので, トークンを持つ場合はあり得ない. 従って, トークン数は 0 である. さらに, 定義 5より, すべての内部変数 $W(P_i)$ の値が等しいのであれば, 各プロセッサ P_iの有効距離は $f_{P_i}(c) = 0$ である. 従って, その和である全有効距離も $f(c) = 0$ である. 以上より, ネットワーク状況が解状況ならば, $\langle T(c), f(c)\rangle = \langle 0, 0\rangle$ である.

逆に, $\langle T(c), f(c)\rangle = \langle 0, 0\rangle$ ならば, トークンが存在しないので, 定義 4より, 任意のプロセッサ Pと, $Q = head(P)$ を考えたとき, 定義にある条件 1,2のいずれも満たさない. つまり, すべてのプロセッサ Pと, $Q = head(P)$ について, 以下の式 1,2の両方を満たす.

$$head(Q) = P\text{ならば}\, W(P) < W(Q) \quad (1)$$

$$tail(Q) = P\text{ならば}\, W(P) \leq W(Q) \quad (2)$$

また, 定義 5より, $f(c) = 0$ ならば, すべてのプロセッサ Pで $f_P(c) = 0$ である. 従って,

$$\forall P, W(P) \geq W(Q) \quad (3)$$

である.

ここで, あるプロセッサ P_i と, $Q_i = head(P_i)$ について, $head(Q_i) = P_i$であるとする. 条件1より, $W(P_i) < W(Q_i)$ かつ, $W(Q_i) < W(P_i)$ でなければならないが, そういう場合は起こり得ない. 従って, すべてのプロセッサ P_iと, $Q_i = head(P_i)$ について, $tail(Q_i) = P_i$である. すなわち, 式 2がすべてのプロセッサ Pについて成立していることになる.

従って, 式 2,3より,

$$\forall\, P, tail(head(P)) = P\text{かつ}\, W(P) = W(head(P)) \quad (4)$$

となる. これは, 定義 6を満たすので, 解状況である. □

また, この 2 項組関数 $\langle T(c), f(c)\rangle$ の値について次の補題が成り立つ.

補題 2 ネットワーク状況 c_iに対して, 2 項組関数 $\langle T(c), f(c_i)\rangle$ を考える. $c_i \rightarrow (\exists Q, A_C)c_{i+1}$かつ $c_i \neq c_{i+1}$ であるとき, $\langle T(c), f(c_i)\rangle$ と $\langle T(c), f(c_{i+1})\rangle$ は辞書式順で単調減少する.

(証明) 表 1より, 動作する場合に関してはすべて, トークンが減少するか, トークンが減少しない場合は, $f(c_i)$ の値が減少している. また新たにトークンが生まれることもない. 従って補題は正しい. □

従って, アルゴリズム A_Cに従って動作してネットワーク状況が変化し続ければ, 補題 2より $\langle T, f(c)\rangle$ は単調減少し, 補題 1より $\langle 0, 0\rangle$ となるネットワーク状況は, 求める解状況であるので, 途中でデッドロックが起こらない限り解状況に到達するといえる.

以下では, 解状況に至るまでにデッドロックが生じないことを示す.

補題 3 任意のネットワーク状況 c_iに対して, トークンが存在すれば, 必ず動作できるプロセッサが存在し, c_iとは異なるネットワーク状況 c_jへ変化できる.

(証明) 表 1より, $P_i(1)$ または, $P_i(2)$ のいずれかが P_i へのトークンを持つ場合は, 明らかに動作しネットワーク状況が変化している. P_i自身がトークンをもつ場合は, P_iは,$P_i(1), P_i(2)$ のいずれかに向くトークンを持つので, $P_i(1), P_i(2)$ の少なくともいずれかは, 動作できる. □

以上より, 解状況でないネットワーク状況でデッドロックは生じないことがわかる. 以上の補題 1,2, 3, より, 以下の補題が成立する.

補題 4 (アルゴリズム A_Cの到達可能性) 任意のネットワーク状況 c_0からのアルゴリズム A_Cの実行により, 必ず解状況 $\mathcal{L}_{ROP}$に達する. つまり $\mathcal{L}_{ROP}$に関して到達可能性を満たす. □

以上で, アルゴリズム A_Cは, 自己安定アルゴリズムの定義 2の条件 1(到達可能性) を満たすことが示せた. 以下では, 定義 2の条件 2(閉包性) も満たすことを示す.

補題 5 任意のネットワーク状況 $c_{nt} \in \mathcal{L}_{ROP}$に対して, アルゴリズム A_Cは, ネットワーク状況を変更させる動作をしない.

(証明) 表 1と定義 6より, $c_{nt} \in \mathcal{L}_{ROP}$ならば, どのプロセッサが動作しても, 表 1の 12の状態遷移にあたり, ネットワーク状況を変えない. □

以上より, ネットワーク状況が解状況であるとき, ネットワーク状況は $\mathcal{L}_{ROP}$を逸脱しないので, 以下の補題が成立する.

補題 6 (アルゴリズム A_Cの閉包性) 任意のネットワーク状況 $c_{nt} \in \mathcal{L}_{ROP}$に対して, アルゴリズム A_Cを動作させても, $\mathcal{L}_{ROP}$に含まれないネットワーク状況になることはない. つまり, $\mathcal{L}_{ROP}$に関して閉包性を満たす. □

補題 4,6より, 以下の定理が成立する.

定理 1 (アルゴリズム A_C) アルゴリズム A_Cは, C デーモンの下で, リングの方向付け問題を解く自己安定アルゴリズムである. □

5. アルゴリズム A_Cの評価について

以下では, アルゴリズム A_Cの最悪時間計算量, つまり, 任意の初期状況から解状況に至るまでの実行ステップ数が, $O(n^2)$ であることを示す.

アルゴリズム A_Cの表 1より, 次の補題がいえる.

補題 7 ネットワーク中にトークンが存在する時, アルゴリズムによってプロセッサがネットワーク状況を変化させるように動作すれば (以降, 単に「動作する」という), トークンは必ず前進するか消える (ネットワーク全体のトークン数が減少する). □

トークン数が減少する (トークンが消える) のは, アルゴリズム A_Cの表 1において, 1,4 のように, より大きなトークンと一緒になる場合 (**トークンの吸収**と呼ぶ), 5,14, 19,27, 28 のように, 向かい合うトークンと一緒になって, 一方が消える場合 (**トークンの衝突**と呼ぶ), 3,11, 16,18, 20,23, 25 のように, (そばにある別のトークンの直接の影響ではなく) 消える場合 (**トークンの消滅**と呼ぶ), のそれぞれの場合がある.

補題 8 互いに逆方向を向くトークンは, いずれ一方は消える.

(証明) 補題 7より, トークンは消えなければ必ず前進する. 従って, 途中で消えなければ, いつかは隣合うプロセッサ P, Q に移動してくる. このとき, $P = head(Q), Q = head(P)$ が成立している. アルゴリズム A_C の表 1より, P, Q のうち, 重みのより小さなプロセッサが持つトークン (重みが等しければ動作したプロセッサの持つトークン) が消える. □

補題 9 ネットワーク中に存在するトークンの向きがすべて同一方向になるまでにかかる実行ステップ数は, 高々$O(n^2)$ である.

(証明) 補題 8より, 互いに逆を向くトークンは, いずれ衝突して消えるか, 衝突する前に消える. N_nのサイズは n であるので, 互いに逆を向くトークンは, 補題 7より高々n ステップ動作すれば, トークンが 1 つは減少する. また, 互いに逆を向くトークンの数も高々n 個であるので, 高々$O(n^2)$ ステップ動作すれば, 互いに逆を向くトークンはなくなり, トークンが残っていれば, それはすべて同一の方向を向いている. □

補題 10 ネットワーク中に存在するトークンがすべて同一方向になってからトークンがなくなるまで, すなわち解状況に到達するまでの実行ステップ数は高々$O(n^2)$ である.

(証明) 補題 7より, プロセッサが動作すれば, 必ずトークンは前進するか消える. アルゴリズム A_Cの表 1より, 逆方向を向くトークンがなければ, トークンの重みがネットワーク中に存在しない重みへ変化する動作を行なうことはない (8, 14は実行されない). つまり, プロセッサの動作は, ネットワークに存在するいずれかのトークンの重みに合わせる動作しか存在しないので, いずれかのトークンの重みにすべてのプロセッサが揃い, ネットワークは解状況に達するはずである. N_nのサイズは, n であるので, 各トークンは高々n ステップ動作すれば消える. また, トークンの数も高々n 個であるので, 高々$O(n^2)$ ステップ動作すればトークンはすべて消えて解状況に達する. □

以上より, 以下の定理が成り立つ.

定理 2 (アルゴリズム A_Cの時間計算量) アルゴリズム A_Cは, 任意のネットワーク状況 c_0から, 高々$O(n^2)$ ステップで解状況に達する.

6. 既知のアルゴリズムとの比較

6.1 アルゴリズム A_{mk}について

過去のアルゴリズムの比較のため, [2] で提案された, C デーモンの下で ROP を解く自己安定分散アルゴリズム A_{mk}を紹介する.

状態遷移表は, アルゴリズム A_Cと同様に場合分けで表すと, 表 1の A_{mk}で示される通りになる.

6.2 小変動からの回復

あるネットワーク状況 $c \in \mathcal{L}_{ROP}$を考える (動作するプロセッサが存在しない). ネットワーク状況 cから, なんらかの原因で, あるプロセッサ (本論文では 1 つ) の内部変数に狂いが生じた (一時故障が生じた) 状態を**小変動 (small disturbance)** と呼ぶ. 自己安定アルゴリズムは, その性質から, 解を求めてもアルゴリズムの実行は停止しない. この特徴により, 一度解を求めた状態から一時故障やネットワーク形状の変化が生じても解を求めることが出来る. つまり, 長期に渡ってネットワーク状況をできるだけ安定に保ち, 一時故障やネットワーク形状の変化に柔軟に対応することの求められるアルゴリズムを動作させる場合に適している. その場合, 安定状態からかけはなれた状態からの動作ではなく, 安定状態から少し変化した状態からの動作が主に行なわれると考えられる. 従って, 任意の状況から安定するまでの動作ステップ数の改良もさることながら, 小変動が生じてから安定するまでの動作ステップ数を改良することは意義がある.

ここでは, 小変動の場合別に, A_Cと A_{mk}で安定するまでに要する実行ステップ数を, 具体的に評価, 比較する. 以下では説明のため, N_nは時計回り方向に方向付けされていたと仮定し, 小変動が起こったプロセッサを P_0とする. また, ネットワーク上のプロセッサは, $P_0, P_1, \ldots, P_{n-1}$の順に時計回りに接続されていると仮定する.

1. 内部変数 $W(P_0)$ の値が大きくなったとき ($\overrightarrow{P_{n-1}} < [\overrightarrow{P_0}] > \overrightarrow{P_1}$) このとき, ネットワーク状況に対する関数である, トークン数と全有効距離 f(c) の 2 項組は, $\langle 1, n-1 \rangle$ である.

A_{mk}: 動作可能なプロセッサは P_1 のみである. 動作ステップ数は $n-1$ で解状況になる.

A_C: 動作可能なプロセッサは P_0, P_1 のいずれか 1 つである.
- P_1 が先に動作した場合. 解状況に至るまでの動作ステップ数は A_{mk} と同じく $n-1$ である.
- P_0 が先に動作した場合. 動作ステップ数は 1 で解状況になる.

したがって, 解状況に至るまでの動作ステップ数は, 最悪値評価をすれば, $n-1$ ステップ, どのプロセッサが最初に動作するかの確率が均等であれば, 平均値評価で, $\frac{n}{2}$ ステップ, 最良値評価をすれば, 1 ステップである.

従って, この小変動の場合, A_C は, A_{mk} より優っている.

2. 内部変数 $W(P_0)$ の値が大きくなり, $D(P_0)$ も変わったとき ($\overrightarrow{P_{n-1}} < [\overleftarrow{P_0}] > \overrightarrow{P_1}$) このとき, $\langle 1, n-1 \rangle$ である.

A_{mk}: 動作可能なプロセッサは P_{n-1} のみである. 動作ステップ数は $n-1$ で解状況になる.

A_C: 動作可能なプロセッサは P_0, P_1 のいずれか 1 つである.
- P_1 が先に動作した場合. 動作ステップ数は $n-1$ で解状況になる.
- P_0 が先に動作した場合. ステップ数は 1 である.

したがって, 解状況に至るまでの動作ステップ数は, 最悪値評価をすれば, $n-1$ ステップ, 平均値評価をすれば, $\frac{n}{2}$ ステップ, 最良値評価をすれば, 1 ステップである.

従って, この小変動の場合も, A_C は, A_{mk} より優っている.

3. 内部変数 $D(P_0)$ の値が変わったとき ($\overrightarrow{P_{n-1}} = [\overleftarrow{P_0}] = (\overrightarrow{P_1})$) このとき, $\langle 2, 2 \rangle$ である.

A_{mk}: 動作可能なプロセッサは, P_0, P_{n-1} である.
- P_0 が先に動作した場合. 動作ステップ数は $1 + (n-1) = n$ で解状況になる.
- P_{n-1} が先に動作した場合. 動作ステップ数は $1 + (n-1) = n$ で解状況になる.

したがって, 動作ステップ数は, n である.

A_C: 動作可能なプロセッサは, P_0, P_{n-1} である.
- P_{n-1} が先に動作した場合. 動作ステップ数は, $1 + (n-1) = n$ で解状況になる.
- P_0 が先に動作した場合. 動作ステップ数は 1 で解状況になる.

従って, 解状況に至るまでの動作ステップ数は, 最悪値評価をすれば, n ステップ, 平均値評価をすれば, $\frac{n+1}{2}$ ステップ, 最良値評価をすれば,1 ステップである.

従って, この小変動の場合も, A_C は, A_{mk} より優っている.

4. 内部変数 $W(P_0)$ の値が減少したとき (方向には無関係) ($\overrightarrow{P_{n-1}} \overset{?}{>} \overrightarrow{P_0} < [\overrightarrow{P_1}]$) このとき, $\langle 1, 1 \rangle$ である.

A_{mk}: 動作可能なプロセッサは P_0 のみである. 動作ステップ数は 1 で解状況になる.

A_C 動作可能なプロセッサは P_0 のみである. 動作ステップ数は 1 で解状況になる.

従って, この小変動の場合は, A_C は, A_{mk} と同等である.

まとめると, 以下のようになる.

小変動の種類	解状況に至るまでの A_C	ステップ数 A_{mk}
$W(P)$ の増大	最悪値: $n-1$ 最良値: 1 平均値: $\frac{n}{2}$	$n-1$
$W(P)$ の増大と $D(P)$ の変化	最悪値: $n-1$ 最良値: 1 平均値: $\frac{n}{2}$	$n-1$
$D(P)$ の変化	最悪値: n 最良値: 1 平均値: $\frac{n+1}{2}$	n
$W(P)$ の減少	1	1

A_C と A_{mk} では, 最悪値評価での動作ステップ数は変わらないが, 平均値評価や最良値評価では A_C の方が改善されている. したがって, 小変動に対して, 安定するまでの動作ステップ数が改善されていると言える.

7. まとめ

本論文では, C デーモンの下で, 任意サイズの均一なリング上で ROP を解く決定性アルゴリズムを提案し, ネットワーク状況の状態変化の数を数えることで, アルゴリズムの時間計算量を評価した. [2] のアルゴリズムと比較して, オーダ的には変わらないが, 小変動 (small Disturbance) 状態から速く収束するアルゴリズムであることを示した.

また, 本論文でのアルゴリズムは, [2] のアルゴリズムにおいて, ネットワーク状況に対して単調減少する関数を考え, その関数がより早く収束するように考慮して作成したものである.

今後の課題として, 他の自己安定アルゴリズムについても, 同様の関数を考えることで, 評価および改良を行なうこと, また, 小変動に強くなるアルゴリズムを考案することが考えられる.

参考文献

[1] A.Israeli and M.Jalfon: "Self-stabilizing ring orientation", Proc. of 4th International Workshop on Distributed Algorithms(LNCS 486), pp. 1–13 (1990).

[2] 片山, 増澤, 都倉 : "リングの方向付け問題を解く自己安定アルゴリズム", 情報処理アルゴリズム研究会, 92-AL-25-8 (1992).

盛光印刷所