

プログラムの複雑性評価

平野行芳, 大場 充, 谷津行穂
(日本アイ・ビー・エム(株) 製品保証)

1. はじめに

作成されたプログラムに潜在するエラーおよびそのデバッグ・修正後さらに混入するエラーの総数は、基本的にそのプログラムの複雑性に依存する。従って、ソフトウェアの開発においては各モジュールごとにプログラムの複雑性を測定し、管理していくことが重要である。そのためには、プログラムの複雑性を定量的に評価する方法が必要となる。

このようなプログラムの複雑性評価法として既に知られているものには、Halsteadの測度¹⁾、McCabeのCyclomatic Number(以下、 C^N と表記する)²⁾、およびプログラム中の実行経路(Path)の総数による方法がある。³⁾ Halsteadの方法は、測定が比較的容易な利点がある反面、プログラムのアルゴリズムや制御構造に依存する意味論上の複雑性に対する考慮がないため、プログラムの物理的な長さ(行数)に直接影響され、複雑性の評価尺度としては不完全であるという欠点がある。また、 C^N は、プログラムの物理的な長さや繰り返し実行節(ループ)の判定節(IF節)との意味論上の差異を抽象するため、多分岐(SELECT文等)や分岐を過大評価する傾向がある。このため、プログラムの構造によっては、実感と C^N との間に差がある場合が生じるという欠点がある。これらの方法に比較して、プログラム中の実行経路の総数による方法は、意味論上の複雑性を増大させる最大の原因となる実行経路の数を基礎としているため、現実のエラー数との間に高い相関があることが報告されている。しかし、著者らの実験例では、特に多分岐(SELECT文)が過大評価され、

必ずしも実感に適合しない場合がある。これは多分岐による実行経路数の増加が、IF文による分岐と同様な重みで評価されるためである。

本小論では、特にプログラムの意味論上の複雑性に着眼し、より実感に適合する複雑性評価尺度として、 E^* を提案する。

2. 複雑性尺度 E^* と複雑度係数 C^*

現実のプログラムの複雑性を示す尺度として、そのプログラムに混入したエラーの総数がある。著者らの実験によれば、プログラムに混入したエラーの総数とプログラムの実行ステップ数との関係は図1のようであった。図1

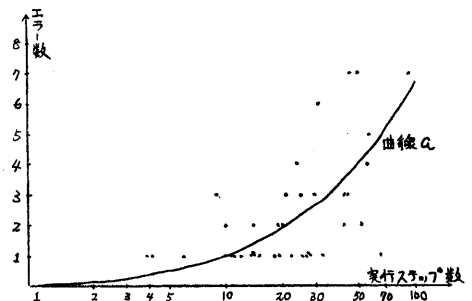


図1. エラー数と実行ステップ数

に示されるエラーの総数とプログラムの実行ステップ数との関係は、巨視的には図1の曲線Aに従っていると考えられる。この曲線Aは、実行ステップ数を m とする時、次式で与えられる:

$$e(m) \doteq C \sqrt{m} \log_{10} m, \quad (1)$$

ただし、 $e(m)$ はプログラムに混入するエラー数、 C は比例定数とする。また C はプログラムの論理的な構造や意味論上の複雑性に比例すると考えられる。

以上の考察より，ここではプログラムの複雑性評価尺度 E^* を以下のように定義する：

$$E^*(n) = C^* \cdot E(n), \quad (2)$$

ただし， n はプログラムの物理的長さ， C^* をプログラムの複雑度係数， $E(n)$ をプログラムの基本複雑度とする。プログラムの基本複雑度 $E(n)$ は以下のように定義される：

$$E(n) = \sqrt{n} \cdot \log_{10} n, \quad (3)$$

ただし， n はプログラムの物理的長さである。プログラムの物理的長さとは，プログラム中に存在する全ての実行文の中から，実行制御文を除いた全ての文の数である。ただし，CALL文や関数マクロを起動する文（FORK, JOIN, INPUT, OUTPUT等）は実行制御文とは考えないものとする。すなわち，実行制御文とは，繰り返し文（DO文および対応するEND文），判定文（IF文およびそれに従属する文），多分岐文（SELECT文等）と定義する。

複雑度係数 C^* は，プログラムの制御の流れに着目し，プログラムの物理的長さ n と，論理的長さ $\lambda(n)$ との比として次式で与えられる：

$$C^* = \frac{\lambda(n)}{n}, \quad (4)$$

プログラムの論理的長さとは，プログラム中にある任意の実行制御文から次の実行制御文まで，順次的に実行される一連の実行文を1つの実行節とする時，各実行節の物理的長さ n_i に比例定数 ρ_i を乗じたものを各実行節の論理的長さとして定義し，その各実行節の論理的長さの総和である。図8の(a)に示されるアルゴリズムをもつプログラムの論理的長さは：

$$\lambda(n_0) = \rho_0 \cdot n_0, \quad (5)$$

で与えられる。ただし， ρ_0 は実行制御

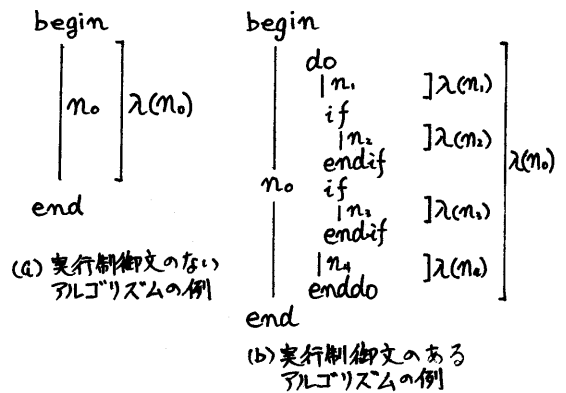


図8. アルゴリズムの例

文を含まないアルゴリズムの定数とする。図8の(b)に示されるアルゴリズムをもつプログラムの論理的長さは，各実行節の論理的長さの総和として次式で与えられる：

$$\lambda(n_0) = \rho_0 \{ \lambda(n_1) + \lambda(n_2) + \lambda(n_3) + \lambda(n_4) \}, \quad (6)$$

ただし，

$$\lambda(n_i) = \rho_i \cdot (n_i), \quad (i=1, 2, 3, 4)$$

とする。

(7)

ここでは， ρ を Implication factor と呼ぶ。(5) 式から明らかなように，実行制御文のないアルゴリズムの複雑度係数 C^* は，その Implication factor に等しい。Implication factor は実行制御文の型式および実行制御文で評価される内部（制御）変数の数によって定まる量であり，その実行制御文に含意される不確定情報の量である。また，(6) 式の例から明らかなように，複雑なネスト構造をもつプログラムほど Implication factor が多重に集じられ，かつ，その値が1以上であることから，プログラムの論理的長さは長くなる。

3. Implication factor ρ

Implication factor ρ は，以下のように定義される：

$$\rho = \frac{1}{a_p} + \log_2(a_n + 1), \quad (8)$$

ただし, a_p は選択的に実行される可能性のある実行経路(節)の数であり, a_n は実行経路の選択に際して評価される内部(制御)変数の数である。すなわち, Implication factor P は, 特定の実行節がどの程度の不確定要素をもって実行されるのかを評価するものである。

具体的に, 実行制御文を含まない一連の順次的に実行される実行文を1つの順次実行節とするとき, その順次実行節に対する Implication factor P_s は, 選択可能な実行経路が単一で, かつ, 内部(制御)変数を含まないことから次式で与えられる:

$$P_s = \frac{1}{1} + \log_2(0+1) = 1, \quad (9)$$

次に, 条件実行節として, IF文に従属した2つの実行節(then以下およびelse以下)における Implication factor P_{IF} を考える。このIF文に代表される判定節では, 選択可能な実行経路が2つ存在する。従って, その実行経路の一方が選択された時点で不確定要素は $1/2$ に減少する。一般に, 1つの内部(制御)変数をもつIF文の Implication factor P_{IF} は:

$$P_{IF} = \frac{1}{2} + \log_2(1+1) = 1.5, \quad (10)$$

となる。thenまたはelse以下の片方の実行節のないIF文の場合でも, null文から成る実行節があると考え, 通常のIF文と同じ取扱いとする。このとき,

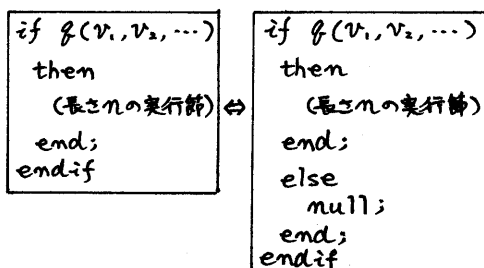


図3. else節のないif文の取扱い

null文の長さはプログラムの物理的長さに含まれないものとする。

さらに条件実行節として, SELECT文またはCASE文に代表される多分岐選択節を考える。この多分岐選択節における Implication factor P_c は, 判定節の場合と同様に, 次式で与えられる:

$$P_c = \frac{1}{K} + \log_2(1+1), \quad (11)$$

ただし, K は分岐の数である。 $K=2$ の場合は, IF文による判定節と同じとなる。

条件実行節の特殊な例としてDO文に代表される繰り返し節の Implication factor P_{DO} を考える。繰り返し節の実行は, 基本的には再帰性の問題を含むため, 他の条件節に比較して不確定要素が高い。これは, 繰り返し節のアルゴリズムが繰り返し実行されるため, その物理的長さに比較して実行時の長さが長いことに帰因する。また, 繰り返しの停止はその内部(制御)変数によって決定されるため, 停止条件を評価するための内部(制御)変数が多くなればなるほど, アルゴリズムの意味論上の複雑性は増大する。このことから, P_{DO} は定性的に次の条件を満足しなければならない:

$$P_{DO} > P_{IF} > P_s, \quad (12)$$

具体的に, 繰り返し節では選択可能な実行経路は単一であることから, P_{DO} は:

$$P_{DO} = 1 + \log_2(a_n+1), \quad (13)$$

で与えられる。従って, 内部(制御)変数が1つの場合, P_{DO} は2となる。これは(12)式を満足する。

4. アルゴリズムの論理的長さ

実行制御文を含まない, 順次的に実行される一連の実行文から成る順次実行節の論理的長さは, そのアルゴリズム

μの Implication factor が1であることから、その物理的長さに等しい。すなわち、その論理的長さ λ_s は:

$$\lambda_s = n, \quad (14)$$

であり、 n はその物理的長さである。

次に、IF文に代表される判定節の論理的長さは、IF文によって評価される内部(制御)変数の数を m とするとき次式で与えられる:

$$\lambda_{IF} = \left\{ \frac{1}{2} + \log_2(m+1) \right\} \{ \lambda(n_{THEN}) + \lambda(n_{ELSE}) \}, \quad (15)$$

ただし、 $\lambda(n_{THEN})$ は THEN 以下の実行節の論理的長さであり、 $\lambda(n_{ELSE})$ は ELSE 以下の実行節の論理的長さである。図4に示すアルゴリズムの例では、IF文

```

if A=1 ∧ B=1
  then
    } (物理的長さ2)
  end;
  else
    } (物理的長さ3)
  end;
endif;

```

図4. 判定節の論理的長さ

がネスト構造をもたないので、THENおよびELSE以下の実行節の論理的長さは物理的長さに等しい。従って、図4における論理的長さは次式で与えられる:

$$\lambda = \left\{ \frac{1}{2} + \log_2(2+1) \right\} (2+3) = 11.1, \quad (16)$$

また、SELECT文に代表される多分岐選択節の論理的長さは、一般に1変数の場合:

$$\lambda_c = \left\{ \frac{1}{K} + \log_2(1+1) \right\} \sum_{i=1}^K \lambda(n_i), \quad (17)$$

で与えられる。ただし、 K は選択される実行節の総数であり、 $\lambda(n_i)$ は各実行節の論理的長さである。図5の例の場合、各実行節はネスト構造をもたないので、その論理的長さは物理的長さ

```

select (A);
  when (A=1)
    } (物理的長さ2)
  end;
  when (A=2)
    } ; (物理的長さ1)
  when (A=3)
    } ; (物理的長さ1)
  other
    } ; (物理的長さ1)

```

図5. 多分岐選択節の例

に等しい。従って、全体の論理的長さは次式で与えられる:

$$\lambda = \left\{ \frac{1}{4} + \log_2(1+1) \right\} (2+1+1+1) = 6.25 \quad (18)$$

最後に、DO文に代表される繰り返し節の論理的長さは、一般に m 変数の場合、

$$\lambda_{DO} = \{ 1 + \log_2(m+1) \} \lambda(n_{DO}), \quad (19)$$

で与えられる。ただし、 $\lambda(n_{DO})$ はDO文以下の実行節の論理的長さとする。ここで、繰り返し数を指定する、DO $i=L, M, N$ 型のアルゴリズムについては、図6に示されるような書換えを行う。

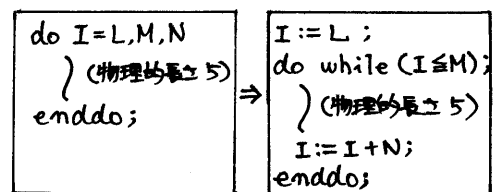


図6. DO文の書換え

従って、図7の繰り返し節の例の場合、その論理的長さは、ネスト構造をもたないことから:

$$\lambda = \{ 1 + \log_2(1+1) \} \times 5 = 10, \quad (20)$$

となる。また、図8の例の場合、同様にネスト構造をもたないので、その論理的長さは:

$$\lambda = \{1 + \log_2(1+1)\}(5+1) + 1 = 13, \quad (21)$$

となる。

```
do while (A < 10);
  }
  A := A + 1;    (物理的長さ 5)
  }
enddo;
```

図7. 繰り返し節の例(I)

```
do I = 1, 10, 1;
  }    (物理的長さ 5)
enddo;
```

図8. 繰り返し節の例(II)

5. 計算例

ここでは、3つの簡単なモジュールについて、複雑度係数 C^* 、複雑性尺度 E^* を計算する。

(1) 計算例1

図9にPL/Iを基礎とした擬似コードによるアルゴリズムの概要を示す。また、そのグラフを図10に示す。

```
AKBENT: PROC;
  ENDSW = 0;
  do until (ENDSW = 1);
    if INTERACTIVE_MODE_FLAG is ON
      then set DISPLAY_MODE to be 1;
      call READ_CONSOLE
        (INPUT_MSG_LEN, CMD);
    if INPUT_MSG_LEN = 0
      then do;
        set DISPLAY_MODE to be 1;
        if INTERACTIVE_MODE_FLAG is ON
          then call EXECUTER;
        else ERROR_RETRY; */
      end;
    else do;
      select (CMD);
```

```
when ('GO')
  do;
    LOOP_COUNTER = 0;
    set DISPLAY_MODE to be 1;
    call EXECUTER
      (LOOP_COUNTER);
  end;
when ('CLEAR')
  call INITIALIZER;
when ('SELECT')
  call CONTROL_MASK_MENU;
when ('FORMAT')
  call FORMAT_MENU;
when ('LOG')
  call LOG_CONTROL_MENU;
when ('DISPLAY')
  call STATUS_DISP_MENU;
when ('TERMINATE')
  ENDSW = 1;
otherwise ERROR_RETRY; */
end;
end;
endproc;
```

図9. 計算例1のアルゴリズム

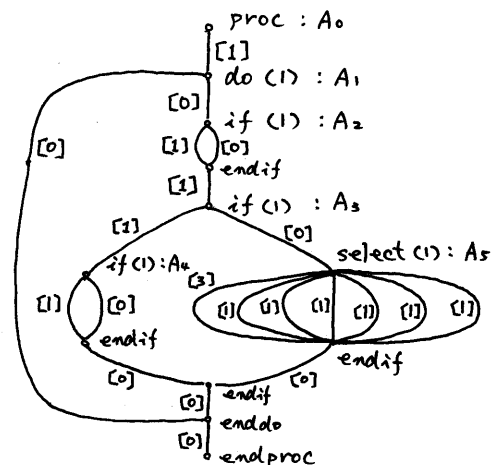


図10. 計算例1のアルゴリズムのグラフ

図10において、()内の数字は内部(制御)変数の数を示し、[]内の数字は実

行節の物理的長さを示す。図10より、アルゴリズムの論理的長さは以下のように計算される。

$$\begin{aligned}\lambda(A_0) &= 1 + \lambda(A_1), \\ \lambda(A_1) &= \{\lambda(A_2) + 1 + \lambda(A_3)\} \{1 + \log_2(1+1)\}, \\ \lambda(A_2) &= 1 \times \left\{\frac{1}{2} + \log_2(1+1)\right\} = 1.5, \\ \lambda(A_3) &= \{1 + \lambda(A_4) + \lambda(A_5)\} \left\{\frac{1}{2} + \log_2(1+1)\right\}, \\ \lambda(A_4) &= 1 \times \left\{\frac{1}{2} + \log_2(1+1)\right\} = 1.5, \\ \lambda(A_5) &= 9 \times \left\{\frac{1}{7} + \log_2(1+1)\right\} = 10.3, \\ \therefore \lambda(A_0) &= 1 + 2\{1.5 + 1 + 1.5(1.5 + 1 + 10.3)\} \\ &\doteq 44.4, \quad (22)\end{aligned}$$

ここで、アルゴリズム全体の物理的長さが14であることから、その複雑度係数 C^* 、および複雑性尺度 E^* は以下のようになる：

$$C^* = \frac{\lambda(A_0)}{14} \doteq \frac{44.4}{14} \doteq 3.17, \quad (23)$$

$$E^* = C^* \cdot (\sqrt{14} \log_{10} 14) \doteq 13.63, \quad (24)$$

(2) 計算例2

図11に擬似コードによるアルゴリズム、図12にそのグラフを示す。

SDTDLT: proc;

```
EVT_POINTER=EVT_POINTER_OF_SVT;
do while (EVT_POINTER≠∅);
  using EVT_LEN based (EVT_POINTER);
  CCB_POINTER=CCB_POINTER_OF_EVT;
  do while (CCB_POINTER≠∅);
    using CCB_LEN based (CCB_POINTER);
    OCB_POINTER=OCB_POINTER_OF_CCB;
    LUB_POINTER=LUB_POINTER_OF_CCB;
    do while (OCB_POINTER≠∅);
      using OCB_LEN based (OCB_POINTER);
      using LUB_LEN based (LUB_POINTER);
      OCB_OLD_POINTER=OCB_POINTER;
      LUB_OLD_POINTER=LUB_POINTER;
      OCB_POINTER=OCB_NEXT_POINTER;
```

```
LUB_POINTER=LUB_NEXT_POINTER;
freemain (OCB_OLD_POINTER,OCB_LEN);
freemain (LUB_OLD_POINTER,LUB_LEN);
end;
CCB_OLD_POINTER=CCB_POINTER;
CCB_POINTER=CCB_NEXT_POINTER;
freemain (CCB_OLD_POINTER,CCB_LEN);
end;
EVT_OLD_POINTER=EVT_POINTER;
EVT_POINTER=EVT_NEXT_POINTER;
freemain (EVT_OLD_POINTER,EVT_LEN);
end;
endproc;
```

図11. 計算例1のアルゴリズム

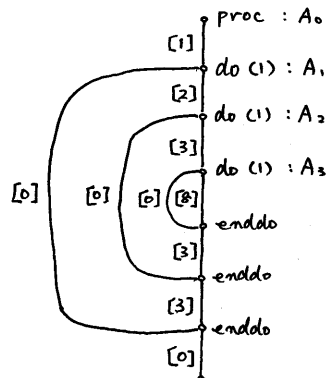


図12. 計算例2のアルゴリズムのグラフ

図12より、アルゴリズムの論理的長さは以下のように計算される。

$$\begin{aligned}\lambda(A_0) &= 1 + \lambda(A_1), \\ \lambda(A_1) &= \{2 + \lambda(A_2) + 3\} \{1 + \log_2(1+1)\}, \\ \lambda(A_2) &= \{3 + \lambda(A_3) + 3\} \{1 + \log_2(1+1)\}, \\ \lambda(A_3) &= 8 \times \{1 + \log_2(1+1)\} = 16, \\ \therefore \lambda(A_0) &= 1 + 2\{2 \cdot (16 + 6) + 5\} \\ &= 99.0, \quad (25)\end{aligned}$$

ここで、アルゴリズムの全体の物理的長さが20であることから、その複雑度係数 C^* 、および複雑性尺度 E^* は以下のようになる：

$$C^* = \frac{\lambda(A_0)}{20} = \frac{99.0}{20} = 4.95, \quad (26)$$

$$E^* = C^* \cdot \sqrt{20} \log_{10} 20 \approx 28.80, \quad (27)$$

(3) 計算例 3

図 13 に擬似コードによるアルゴリズム
ム, 図 14 にそのグラフを示す。

```
ERRDISP: proc;
  if RETURN_CODE ≠ ∅
  then
    if ERROR_FLAG is OFF
    then
      if THRESHOLD_FLAG is OFF
      then do;
        set THRESHOLD_FLAG to be ON;
        set SENSE_CODE to be X'FFFF';
        call ABEND(THRESHOLD_FLAG,
                   SENSE_CODE);
      end;
      else set END_OF_TEST_FLAG
            to be ON;
    else do;
      call ERROR_LOG;
      RETURN_CODE = 1;
      if (SENSE_CODE ∧ SENSE_MARK = ∅)
      ∧ (ERROR ≠ DATA_ERROR ∨
         PRINT_ERROR)
      then do;
        call MEDIA_EJECT;
        set CLEAR_REQUEST to be ON;
        if LOG_FULL_FLAG is OFF
        then do;
          reset ERROR_FLAG OFF;
          RETURN_CODE = ∅;
        end;
      end;
    end;
  end;
endproc;
```

図 13. 計算例 3 のアルゴリズム

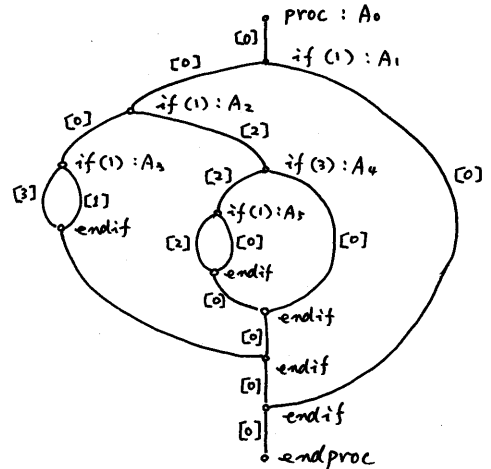


図 14. 計算例 3 のアルゴリズムのグラフ

ここで, アルゴリズム全体の物理的長さ
が 10 であることから, その複雑度
係数 C^* , および複雑性係数 E^* は以
下のようになる。

$$\lambda(A_0) = \lambda(A_1),$$

$$\lambda(A_1) = \lambda(A_2) \cdot \left\{ \frac{1}{2} + \log_2(1+1) \right\},$$

$$\lambda(A_2) = \{ \lambda(A_3) + 2 + \lambda(A_4) \} \left\{ \frac{1}{2} + \log_2(1+1) \right\},$$

$$\lambda(A_3) = (3+1) \left\{ \frac{1}{2} + \log_2(1+1) \right\} = 6.0,$$

$$\lambda(A_4) = \{ 2 + \lambda(A_5) \} \left\{ \frac{1}{2} + \log_2(3+1) \right\},$$

$$\lambda(A_5) = 2 \cdot \left\{ \frac{1}{2} + \log_2(1+1) \right\} = 3.0,$$

$$\begin{aligned} \therefore \lambda(A_0) &= \{ 6 + 2 + 2.5(2+3) \} \times 1.5 \times 1.5 \\ &\approx 46.13, \end{aligned} \quad (28)$$

$$C^* = \frac{\lambda(A_0)}{10} = \frac{46.13}{10} \approx 4.61, \quad (29)$$

$$E^* = C^* \cdot \sqrt{10} \log_{10} 10 \approx 14.58, \quad (30)$$

6. 比較評価

ここでは, プログラムの複雑性評価
の尺度として一般的な, C^* およびプ
ログラム中の実行経路の総数 P^* と,
 E^* との比較評価を行う。比較は前述
の計算例の 3 つのアルゴリズムについ
て行う。

	実行ステップ数	C^*	E^*	C^N	P^N	$\overline{C^N}$	$\overline{P^N}$
計算例1 (A_1)	24	3.17	13.63	11	36	0.46	1.5
計算例2 (A_2)	26	4.95	28.80	4	8	0.15	0.31
計算例3 (A_3)	22	4.61	14.58	6	6	0.27	0.27

表1. 各計算例における複雑性評価法の比較

表1に、各計算例に対する E^* , C^N , および P^N の計算結果を示す。また、 C^N および P^N に対しては、 C^N および P^N の値をプログラムの実行ステップ数で基準化した $\overline{C^N}$ および $\overline{P^N}$ も示してある。表1から明らかなように、 C^N および $\overline{C^N}$ によるアルゴリズムの複雑性評価は以下のとおりであった:

$$C^N(A_1) > C^N(A_3) > C^N(A_2), \quad (31)$$

このことから、 C^N ではDO文による繰り返しのアルゴリズムが過小評価されること、また、SELECT文等による多分岐型アルゴリズムが過大評価されることわかる。

次に、 P^N および $\overline{P^N}$ によるアルゴリズムの複雑性評価は以下のとおりであった:

$$P^N(A_1) \gg P^N(A_2) > P^N(A_3), \quad (32)$$

このことから、 P^N ではDO文による繰り返し型のアルゴリズムやIF文による判定型のアルゴリズムに対しては、実感によく適合しており妥当な値をとるが、多分岐型のアルゴリズムを過大評価する欠点がある。従って、一般的に多分岐型のアルゴリズムを含まないプログラムに対して、 P^N は実感によく適合すると考えられる。

最後に、 E^* による複雑性評価は以下のとおりであった:

$$E^*(A_2) > E^*(A_3) > E^*(A_1), \quad (33)$$

この結果は、実感とよく適合する。 A_1 のアルゴリズムは、SELECT文による多分岐を除外すれば、簡単な構造であり、制御の流れに着目する限りその複雑性は低い。 A_3 のアルゴリズムは、IF文のネスト構造が全てである。IF文の実行を考える場合、その実行は常に前向きであり、DO文による繰り返し実行に比較してその複雑性は低い。従って、 E^* による評価は、このような実感によく適合しているといえる。

ところで、 $E^*(A_2)$ が $E^*(A_1)$ および $E^*(A_3)$ に比較して大きい値をとるのは、アルゴリズム A_2 の物理的長さが長いためであり、 C^* で比較した場合には、アルゴリズム A_3 のIF文が4重のネスト構造をもつことから、ほとんど差がない。これは $\overline{P^N}$ の値ともよく一致している。このことから、一般に C^* と P^N とは似た性質をもつと考えられる。

7. まとめ

プログラムの複雑性を、制御構造上の複雑さに加え、内部(制御)変数による意味論的な複雑さの変化を考慮することにより、アルゴリズムの複雑度係数 C^* を定義した。これにより、アルゴリズムの物理的長さ依存しない複雑性の評価基準が設定されるので、アルゴリズムの物理的長さを基にする、

基本的な複雑性の評価を乘じることにより、アルゴリズムを客観的に評価することが可能になる。また、計算例で示したように、これらの評価は実感によく適合している。

一般に、50 ステップ程度の実行文（実行制御文は含まないものとする）から成るアルゴリズムの複雑度は E^* で約 12 である。実行文で 50 ステップのプログラムは、通常ソース・リストで8頁にわたる。これを1つの日付とするならば、 E^* で 12 を超えるアルゴリズムはコーディング時に注意を要するモジュールといえる。特にアセンブラ言語では、このことは重要である。

今後の課題としては、本方法による複雑性評価と現実のエラーとの相関を調査し、本方法の妥当性を検証していくことが重要である。また、モジュール間で共通に参照・変更される（制御）変数の導入によって増加する複雑性をどう評価するかも、重要な問題である。

[参考文献]

- 1) Halstead M.: "Elements of Software Science"
Elsevier North-Holland, 1977.
- 2) McCabe T. J.: "A Complexity Measure"
IEEE Trans. Software Engineering,
(1976, pp308-320)
- 3) D. Potier, J.L. Albin, R. Ferreol, A. Bilodeau
"Experiments with Computer Software
Complexity and Reliability"
6th ICSE pp94-101
- 4) Baker A.L., Zweber S.H.
"A Comparison of Measure of Control
Flow Complexity"
IEEE Trans. Software Engineering,
(1980, pp506-512)