B5　計算機によるソフトウエアの検査の機械化について

安井　　裕，治田　倫男，永田　恒一　（大阪大学工学部）
藤田　安臣　　（住友電気工業株式会社）

1. まえがき

ソフトウェアの機能検査は一般に不充分となり，使用後もなおかつ多くの事故が発生しているのが現状である．プログラムの検査をすべて機械化することは不可能であるが，我々はソフトウェアの製作，受入れにおける検査の合理化の一環として，その機械化を計画し，実用になる範囲内でテストシステムの試作を行なった．

システム・プログラムなどのソフトウェアの製作に際して，一般にはプログラムをモジュール化して，各モジュールの変更，追加に対し他のモジュールにできるだけ影響を及ぼさないように構成し，モジュール毎に検査を行ないながら全体を組立てて行く方式をとっている．すべての機能を検査するためには無限に近いテストプログラムまたはデータが必要となり，実際には不可能であるが，少なくとも検査の対象としているプログラムのすべての論理経路（path）を通るテスト情報の集合が必要である．検査の対象としているプログラムの入力情報のフォーマルな表現としての文法表からテスト情報を自動的に発生させるプログラムがすでに作成され[1]，[2]，筆者らによって大阪大学大型計算機センターのTSS製作時におけるコマンド・システムの検査に使用し成果が得られた．それは特に，不特定多数のユーザ個人が独占する端末を介して行なわれるという利用形態下において起る．システム製作者達では予想もつかないようなコマンドの使用状況の発生に対するシステム・プログラムでの防禦の強化，またシステムの誤動作の発見を促進した．しかしこれらの過程で得られた検査における問題点は

1. どのように検査に適したテスト情報を発生さすか．
2. テストデータまたはプログラムを被テストプログラムに与えて，得られた結果に対する評価の機械化．

であり，1についてはさらに

（a）できるだけ少ない検査回数で全部を調べたい．

（b）誤りのある被テストプログラムの誤り個所を推定したい．

などが身近かな問題になった．そこで我々はさらに機械化を強化した検査システムとして上記の(a),(b)の解決を目的とする新たなシステムについてここに述べる．

これは被テストプログラムの機能を合理的，且つ迅速に検査するためのテスト情報の集合を，テストの対象となるプログラムの構造を考慮に入れて，ヒューリスティックに発生させるシステムである．すなわちテスト情報の発生部分と通過したモジュールとの間のフィードバックを行なっている．また，このテストシステムでは，発生したテスト情報の集合の使用結果から，プログラムのどのあたりが誤り箇所であるかを機械的に推定することができる．したがって，誤り箇所が局所的に限定された時点で，従来使用されているデバッグの道具を用いて窮極的な誤り箇所を検出し，修正を行なうことができる．

このシステムは検査作業およびデバッグ作業を計算機の助けを借りながら行なうことができるシステムであり，TSSの普及に伴ない，さらに効果的に使用することができる．

2. テストシステム（TSP）の概要

2.1 文法とテスト情報の作成

検査の対象となるプログラムのBackus記号で記述される入力情報の文法からテスト情報を発生させるプロセスを言語生成と見なし，つぎのごとく定義する．

テスト情報の文法は

$$U_i \rightarrow u_i \qquad (1)$$

の形のプロダクション（production）の有限な集合で表わされる．[3] (1)式の左辺U_iは非末端記号（nonterminal symbol），右辺u_iは末端記号（terminaj symbol），非末端記号，あるいはこれらの連系（string）のいずれかを表わしU_iがu_iと定義されることを意味している．但しu_iの中にはあらわれないU_iが必ず一つ存在する．これを識別記号（distinguished symbol）Zと称する．U_iが多義的である場合（$U_i \rightarrow u_{i1}$，$U_i \rightarrow u_{i2} \cdots U_i \rightarrow u_{in_i}$），$n_i$個のプロダクションで定義される．これを一つにまとめて

$$U_i \rightarrow \prod_{j=1}^{n_i} u_{ij} \qquad (2)$$

と表わし新たにこれをプロダクションと称することにする．u_{ij}は(1)のu_iに相当し，非末端記号U_iが一義的に定義されている場合は$n_i = 1$である．いまテスト情報がm個のプロダクション（(2)式で$i = 1, 2, \cdots, m$）の集合で定義されているとする．u_{ij}を明確にするためこの中にある末端記号あるいは非末端記号をμ_{ijk}とあらわし，その数をl_{ij}とするとu_{ij}は(3)式で表わされる．

$$u_{ij} = \mu_{ij1}\, \mu_{ij2} \cdots \mu_{ijl_{ij}} \cdots \qquad (3)$$

μ_{ijk}が非末端記号である場合はU_1，U_2，…，U_mのうちいずれかに等しくなり(2)式により同様に定義される．前述のZをU_1とすると(2)式から

$$Z \rightarrow \prod_{j=1}^{n_1} u_{1j} \qquad (4)$$

とあらわされる．このm個のプロダクションはテスト情報を定義している時，Zは＜テスト情報＞となる．

(2)式とBackus記号との関係を例をもって説明する．

＜正整数＞ ::＝＜数字＞ | ＜正整数＞＜数字＞

＜数字＞ ::＝ 0 | 1 | 2

これらは(2)式の記号とつぎのように関係している．

U_1＝＜正整数＞，u_{11}＝＜数字＞，u_{12}＝＜正整数＞＜数字＞

μ_{111}＝＜数字＞，μ_{121}＝＜正整数＞，μ_{122}＝＜数字＞

U_2＝＜数字＞，$u_{21} = 0$，$u_{22} = 1$，$u_{23} = 2$

$\mu_{211} = 0$，$\mu_{221} = 1$，$\mu_{231} = 2$

非末端記号：　＜正整数＞，＜数字＞

末端記号　：　0，1，2

試作テストシステムTSPの入力としての文法の表現には，入出力装置の制限から図3.1の例のごとく末端，非末端記号はすべて“（”と“）”で囲み，記号→は“＃”，末端記号，非末端記号の“AND”は“－”，“OR”は“／”で表わし，各々のプロダクションの終りには，“，”を付ける．（¥¥）はステートメントの行を改める書式記号である．

2.2　プロダクションの機械化と文法のマトリックス表示

2.1で使用した非末端記号U_i，および非末端記号あるいは末端記号のμ_{ijk}をそれぞれ一つのステート（state）と考えるとテスト情報を得るという問題はステートの遷移系列を得るという問題とみることができる．

U_1，μ_{111}，…，$\mu_{11l_{11}}$，μ_{121}，…，$\mu_{12l_{12}}$，…，$\mu_{1n_1l_{1n_1}}$；U_2，…，$\mu_{2n_2l_{2n_2}}$；U_m，…，$\mu_{mn_ml_{mn_m}}$をそれぞれ順番にステートS_1，S_2，…，S_nと対応させる．但し$n=m+\sum_{i=1}^{m}\sum_{j=1}^{n_i} l_{ij}$である．そこでMGP（Matrix Generator Program）によりプロダクションの集合をつぎの四つのマトリックスで置き換える．

2.2.1　ASM（Alternativity State Matrix）

$$\begin{array}{c} \\ S_1 \\ S_2 \\ \vdots \\ S_n \end{array}\begin{array}{c} \begin{array}{cccc} S_1 & S_2 & \cdots & S_n \end{array} \\ \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ & \cdots\cdots\cdots & & \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} \end{array}$$

$a_{ij}=0$または1

（$i=1,2,\cdots,n$；$j=1,2,\cdots,n$）

$b_j=\sum_{j=1}^{n} a_{ij}$

nはステートの総数．

ASMはU_iに相当するステートからμ_{ij1}に相当するステート（$i=1,2,\cdots,m$；$j=1,2,\cdots,n_i$）への推移をあらわす正方行列である．推移があるときはそれに相当するASMの要素を1とし，推移がない要素は0とする．前述の2.1での例においてはU_1がS_1に，μ_{111}がS_2に相当している場合$a_{12}=1$である．このASMから選択系列をつくるのに必要なSPM（Selection Probability Matrix）の初期状態が作成できる．

$$\begin{bmatrix} p_{11} & p_{12} & \cdots & p_{1n} \\ p_{21} & p_{22} & \cdots & p_{2n} \\ & \cdots\cdots\cdots & & \\ p_{n1} & p_{n2} & \cdots & p_{nn} \end{bmatrix}$$

$0\leqq p_{ij}\leqq 1$

$\begin{cases} b_j \neq 0 \text{のとき } p_{ij}=a_{ij}/b_j \\ b_j=0 \text{のとき } p_{ij}=0 \end{cases}$

$\sum_{j=1}^{n} p_{ij}=1$または0

（$i=1,2,\cdots,n$；$j=1,2,\cdots,n$）

SPMの要素はU_iが多義的である場合どのu_{ij}即ちどのμ_{ij1}（$j=1,2,\cdots,n_i$）を選べばよいか（推移すればよいか）その確率を示す行列である．

2.2.2　CSM（Concatenation State Matrix）

μ_{ijl-1}に相当するステートからμ_{ijl}に相当するステート（$i=1,2,\cdots,m$；$j=1,2,\cdots,n_i$；$l=2,3,\cdots,l_{ij}$）への推移をあらわす正方行列で，形式はASMと同じである．

2.2.3　DSM（Defined State Matrix）

μ_{ijk}（$i=1,2,\cdots,m$；$j=1,2,\cdots,n_i$；$k=1,2,\cdots,l_{ij}$）が非末端記号であるステートからこの非末端記号に等しいU_i（$i=1,2,\cdots,m$）に相当するステートへの推移をあらわすASMと同じ形式の正方行列である．

2.2.4　PSM（Parent State Matrix）

$\mu_{ijl_{ij}}$に相当するステートからU_iに相当するステート（$i=1,2,\cdots,m$；$j=1,2,\cdots,n_i$）への推移をあらわすASMと同じ形式の正方行列である．

2.3　テストプログラム・ジェネレータ（TPG）

2.2で述べた四つの行列ASM（SPM），CSM，DSM，PSMをもとにテストプログラムを発生させるのがテストプログラム・ジェネレータTPG（Test Program Generator）であり，このジェネレータの機能を簡単な流れ図で示すと図2.1のようになる．この図で選択ルーチン（selection noutine）とはステートS_iからSPMの要素

p_{ij}（ $j=1, 2, \cdots, n$ ）に比例した確率でS_jを推移ステートとする機能をもっている．出力ルーチン（ output routine ）とは発生される末端記号の連系をテストプログラムとして出力するルーチンである．選択ルーチン，出力ルーチン以外の部分が誘導ルーチン（ derivative routine ）である．

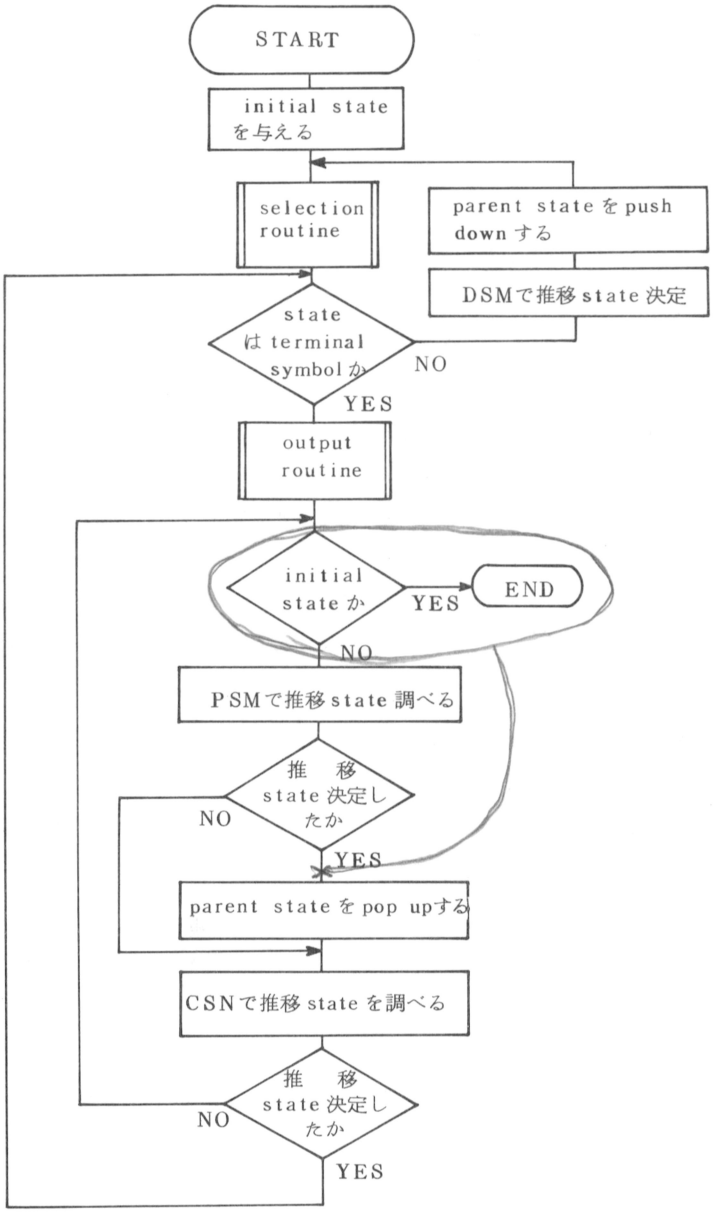

図2.1　Test program Generator の流れ図

2.4　検査の対象となるソフトウェアのマトリックス表示

プログラムをモジュール化するとプログラムはノード(node)とアーク(arc)がそれぞれのモジュールとモジュール間の情報の流れをあらわす有向グラフ(directed graph)であらわすことができる．図2.2はグラフの一例を示している．これはノードN_1，N_2，…，N_6とアークA_1，A_2，…，A_7であらわされている．このグラフは隣接マトリックス(adjacency matrix)と呼ばれる正方行列(図2.3)で置き換えることができる．アークを考慮してこの隣接マトリックスを用いると図2.4のように入力ノード(input node)N_1から出力ノード(output node)N_7への高高四つの可能な経路が存在する．この四つの経路はパス・マトリックス(path matrix)であらわすことができる(図2.5)。各々の経路は各列がノードまたはアークを示すBoolean要素をもつ行ベクトルであらわされる．　0はこの経路に該当したノードあるいはアークが存在しないことを示し，1は存在することを示す．アークの導入はノードをどのような順序で選ぶかを明確にするためである．但し経路P_3，P_4のようにノードN_2，N_4，N_5の部分でループする場合のそれぞれのノードの使用回数は考慮に入れていない．

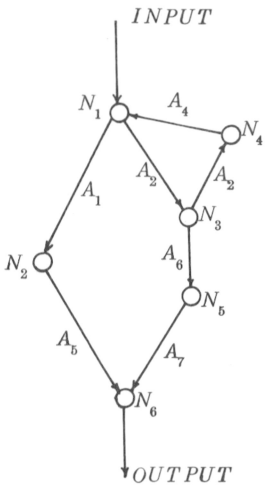

図2.2　directed graphの例

	N_1	N_2	N_3	N_4	N_5	N_6
N_1	0	1	1	0	0	0
N_2	0	0	0	0	0	1
N_3	0	0	0	1	1	0
N_4	0	0	0	0	0	0
N_5	0	0	0	0	0	1
N_6	0	0	0	0	0	0

図2.3　図2.2のadjacency matrix

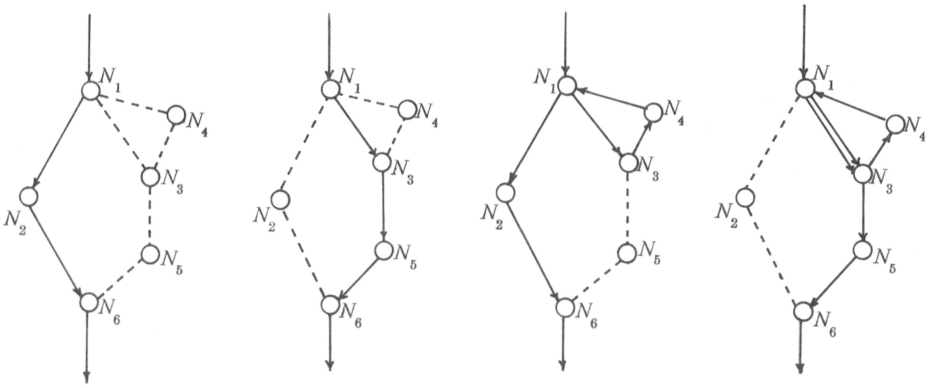

図2.4　図2.2の高々可能なpath

	N_1	N_2	N_3	N_4	N_5	N_6		A_1	A_2	A_3	A_4	A_5	A_6	A_7
P_1	1	1	0	0	0	1		1	0	0	0	1	0	0
P_2	1	0	1	0	1	1		0	1	0	0	0	1	1
P_3	1	1	1	1	0	1		1	1	1	1	1	0	0
P_4	1	0	1	1	1	1		0	1	1	1	0	1	1

図2.5　図2.2のpath matrix

検査の対象とするプログラムには通過したノードとアークを記録するTPC（Test Path Check Program）にブランチするステートメントを挿入する必要がある．TPCで作成された経路をもとにPMG（Path Matrix Generator）によりパス・マトリックスを作成する．

つぎにできるだけ多くの経路を検査できるテスト情報の集合を迅速に発生させるために，TRN（trainer）を導入する．

2.5　トレーナー（TRN）

トレーナーはプログラムの検査の目的に応じたテストプログラムの集合を発生させるため，TPGで発生されたテストプログラムが検査の対象としているプログラムのどの経路を通ったかをチェックすることによりSPMの要素である確率を変化させる機能をもっている．ここではSPMの $p_{ij} \neq 0$ である要素のみを変化させる学習方法の一例について述べる．

一つのテストプログラムを発生させるためにSPMで選択した推移ステートの数を L とすると選択系列（SPMで選択された推移ステートからなる系列）は

$$S_{t_1} \rightarrow S_{t_2} \rightarrow \cdots \rightarrow S_{t_L}$$

とあらわされる．この選択系列を得るために使用したSPMの確率要素 $p_{s_i t_i}$（$i = 1, 2, \cdots, L$）を変化させるオペレータとして線形オペレータ（linear operator）を採用した．[4]

（i）　同じ選択系列の発生の確率をさげる学習

$$\left.\begin{array}{l} p_{s_i t_i} \rightarrow \alpha_i \, p_{s_i t_i} \\ p_{s_i j} \neq 0 \text{ のとき} \quad p_{s_i j} \rightarrow \alpha_i \, p_{s_i j} + (1 - \alpha_i) / (m_i - 1) \\ p_{s_i j} = 0 \text{ のとき} \quad p_{s_i j} \rightarrow p_{s_i j} \qquad (j = 1, 2, \cdots, n \neq t_i) \\ \\ m_i \text{ は } p_{s_i j} \ (j = 1, 2, \cdots, n) \text{ 中 } p_{s_i j} \neq 0 \text{ の個数}, \ 0 < \alpha_i < 1 \ (i = 1, 2, \cdots, L) \end{array}\right\} \quad (5)$$

（ii）　同じ選択系列の発生の確率をあげる学習

$$\left.\begin{array}{l} p_{s_i t_i} \rightarrow \beta_i \, p_{s_i t_i} + (1 - \beta_i) \\ p_{s_i j} \rightarrow \beta_i \, p_{s_i j} \ (j = 1, 2, \cdots, n \neq t_i) \\ 0 < \beta_i < 1 \quad (i = 1, 2, \cdots, L) \end{array}\right\} \quad (6)$$

2.6　検査結果の評価

検査結果の評価とは発生させたテストプログラムを検査の対象としているプログラムにより正しく処理されたか否かを判定することである．この判定を人間が行う場合テストプログラムが処理される毎にTSPとの会話により行う方法（この場合はTRNに評価結果が反映される）と各テストプログラムの処理結果をまとめて最後に行う方法とが考えら

れる．

このテストシステムがソフトウェアのうち特にシステム・プログラムの検査に適しているのはシステム・プログラムの場合，検査結果の評価が比較的容易にできるためである．すなわち人間が入力に対する結果を予測しやすいためである．

この評価の部分の機械化はプログラムの機能拡充後，従来の機能が正しく維持されているかどうかを検査する場合には可能である．例えば新旧両方の言語処理プログラムで作成されたオブジェクト・コードなどの各種情報を機械的に比較することが可能である．またオブジェクト・プログラムを実際にラン（ run ）させて，その結果を比較することも可能である．

2.7　誤り箇所の判定

図2.5において経路P_1を通るテストプログラムの処理結果が誤りであり，他の経路を通るテストプログラムの処理結果が正しい場合には直観的にN_2に誤りがある可能性が一番高く，誤りのある可能性の少ないのはN_3，N_4，N_5であることに気がつく，そこでどのノードに誤りが存在するかを示す確率をつぎのように定義する．

誤り箇所を検出する時点までに調べられたパス・マトリックス上の経路の数をgとする．ノードの数をc個とし，パス・マトリックスの要素をf_{ij}とする．但し$i=1,2,\cdots,g$；$j=1,2,\cdots,c$であり$f_{ij}=0$または1である．テストプログラムが経路P_iを通って正しい結果を得た回数をC_i，誤った結果を得た回数をE_iとするとP_iの経路を検査した回数T_iは$T_i=C_i+E_i$となる．ノードN_jが誤り箇所である確率$P(j)$は

$$F(j)=\sum_{i=1}^{g} f_{ij}E_i \Big/ \sum_{i=1}^{g} f_{ij}T_i$$

が一つの尺度と考えられるため

$$P(j)=F(j)\Big/\sum_{j=1}^{c} F(j) \qquad (7)$$

と定義する．

誤り箇所の検出には誤り箇所としての確率の高いノードを更にいくつかのノードに分けて同様な検査を行い，誤り箇所を限定して行けばよい．誤り箇所が限定された時点で従来使用されているデバッグの方法を用いて窮極的な誤り箇所を検出し，訂正すれば効果的なデバッグを行うことができる．

3. 実験例

3.1　実験例1

ALGOL言語に類似する四則演算の言語を機械語に逐次翻訳する棚上げ方式の本実験のために特に設定したコンパイラCOMPの検査にこのテストシステムTSPを使用した結果について述べる．COMP言語の文法を図3.1に示す．COMPコンパイラの詳細[5]は省略するが図3.2に示すノードに分けられて作成されている．

```
(STATEMENT)#(VARIABLE)-(#)-(ARITH. EXPR. )-(¥¥),
(ARITH. EXPR. )#(TERM)/(ARITH. EXPR. )-(ARITH. OPERATOR)-(TERM),
(TERM)#(VARIABLE)/(INTEGER),
(ARITH. OPERATOR)#(+)/(-)/(※)/(/),
(VARIABLE)#(LETTER),
(LETTER)#(I)/(J)/(K)/(L)/(M)/(N),
(INTEGER)#(DIGIT)/(DIGIT)-(INTEGER),
(DIGIT)#(1)/(3)/(5)/(7)/(9)
END OF SYNTAX
```

図3.1　COMP言語の文法

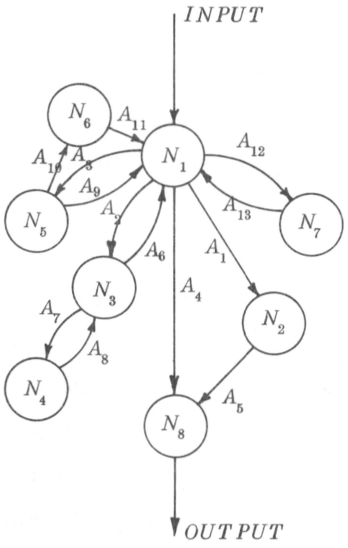

図3.2　COMPのdirected graph

3.1.1　実験手順

図3.1の文法からテストプログラムを発生させ，これを検査の対象としているCOMPに通してコンパイルする．TPCでCOMPのどの経路を通ったかが確認され，PMGによりパス・マトリックスが作成される．CCP（Complection Check Program）の指示に従ってテストプログラムの発生を続けるか否かを判定し，続ける場合はTRNでSPMを学習させ，TPGに戻る．以後同様に繰返す．以上すべて自動的に行われ，各テストプログラムのコンパイル結果が正しいか否かの判定はあとで人間が行う．この実験で行った学習は

(i)　学習を行わない場合（2.5の(5)式の$\alpha_i = 1$の場合）．

(ii)　2.5の(5)式の学習を行う場合．但しα_iは

$\left(T_k \Big/ \sum_{k=1}^{g} T_k \right) \leqq \left(1 / g \right)$のとき　$\alpha_i = 1$

$\left(T_k \Big/ \sum_{k=1}^{g} T_k \right) > \left(1 / g \right)$のとき

$$\alpha_i = \left(K / i \right) \cdot \left\{ \left(1 / g \right) \Big/ \left(T_k \Big/ \sum_{k=1}^{g} T_k \right) \right\}$$

但しKは$1 \leqq K \leqq i$の定数

選択系列内に同じステートが二つ以上ある場合はiの大きい推移ステートS_{t_i}に対してのみ学習を行う．

3.1.2　実験結果

図3.3は(i)，(ii)の学習方法で発生させたテストプログラムをCOMPでコンパイルした結果通るCOMPの各経路（アークの要素を記入していない）と回数を示している．(ii)の学習を行うことにより，より多くの経路を通るテストプログラム群を少ない回数で発生できることがわかる．

コンパイル結果を調べることにより経路P_9，P_{10}，P_{15}，P_{16}を通るテストプログラムすべてが誤りであるとき，(7)式からN_3を誤り箇所として判定することができる．

PATH MATRIX									(i)学習を行わない場合					(ii)学習を行った場合				
									テスト回数					テスト回数				
	N_1	N_2	N_3	N_4	N_5	N_6	N_7	N_8	10	30	50	100	200	10	30	50	100	200
P_1	1	1	0	0	0	0	0	1	1	2	11	21	45		3	4	11	27
P_2	1	1	0	0	0	0	1	1	5	12	15	28	55	2	6	14	24	48
P_3	1	1	1	0	1	1	0	1						1	1	1	1	1
P_4	1	1	1	0	1	1	1	1	1	1	1	2	3	1	1	1	2	2
P_5	1	1	1	0	0	0	0	1	1	1	1	2	7	1	2	2	4	12
P_6	1	1	1	0	0	0	1	1	2	6	8	15	30	3	3	3	7	14
P_7	1	1	0	0	1	0	0	1				2	2		1	2	3	5
P_8	1	1	0	0	1	0	1	1		2	3	9	20		2	3	6	9
P_9	1	1	1	1	1	0	0	1					3			1	5	10
P_{10}	1	1	1	1	1	0	1	1		2	4	7	14		2	5	11	21
P_{11}	1	1	0	0	1	1	0	1									1	4
P_{12}	1	1	0	0	1	1	1	1		2	4	6	7		2	2	4	12
P_{13}	1	1	1	0	1	0	0	1										1
P_{14}	1	1	1	0	1	0	1	1		1	1	4	8	1	3	4	5	7
P_{15}	1	1	1	1	1	1	1	1							1	1	1	3
P_{16}	1	1	1	1	1	1	0	1		1	2	4	6	1	3	7	15	24

図3.3　COMPのpath matrixと各pathの通過回数

3.2　実験例2

TSPをFORTRAN言語で作成し大阪大学大型計算センターのタイムシェアリングシステムの端末を使用して実験を行った．検査の対象として実係数二次方程式の根の種類を計数的に求めるプログラム（文献〔5〕より転載）を採用した．これは実験をFORTRANの範囲で実現するために，システム・プログラムの代替として用いたものである．このプログラムを基本的な仕事内容単位にブロック化し，それぞれにノード番号N_1，N_2，…，N_{10}を付け（図3.4），それらの通過箇所に次に示すステートメント

CALL PATH(n)

を挿入する．但しnはノード番号（アーク番号）を整定数で与える（N_1，N_2，…，N_{10}の添字に相当する）．

図3.4の有向グラフのもつ経路の数は高々六個である．端末からのメッセージのうち左端に＊印のついた行は人間が与えた情報を示し，それら以上は全部TSPシステムからの応答である．実験例[7]，[8]の一部を図3.5に示す．実験に際し図3.4のプログラムは一部意識的に間違わせてシステムに与えてあり，パス・マトリックスから誤りの原因となっているノードN_3を見い出すことができる．

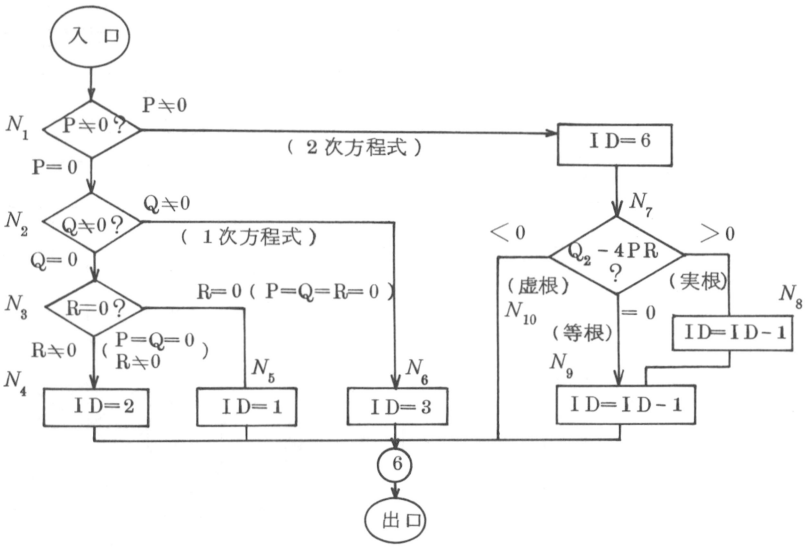

図 3.4　プログラムの流れ図

図 3.5　実験例 2 の結果 (一部省略)

```
 READY 11-19-55
%START TESTP

 WAIT  11-20-28

(( SYNTAX TABLE ))

(DATA)#(P)-(#)-(NUMBER)-(¥¥)-(Q)-(#)-(NUMBER)-(¥¥)-(R)-(#)-(NUMBER)-(¥¥),

(NUMBER)#(UNSN.NUM)/(+)-(UNSN.NUM)/(-)-(UNSN.NUM),

(UNSN.NUM)#(DIGIT)-(.)/(.)-(DIGIT)/(DIGIT)-(.)-(DIGIT),

(DIGIT)#(0)/(1),.


## GIVE P1, PARAMETER OF NUMBER OF DATUN GENERATED FIRST
        (P1:NUMBER OF PATHS, 4 FIGURES, REAL, GREATER THAN 1.0) ##
%2.

          P1#  2.0

## GIVE COEFFICIENT ALPHA (3 FIGURES, REAL, LESS THAN 1.0)
                    BETA  (3 FIGURES, REAL, GREATER THAN 1.0) ##

ALPHA#
%.8

          ALPHA#  .8

BETA#
%1.3

          BETA# 1.3

(( PROBABILITY RANDOM SUBSTITUTION SELECTION METHOD ))

¥¥ 1ST STEP ¥¥


(NO.   1)

P#-1.
Q#-1.
R#-.1 .

-- TWO REAL ROOTS --

 4)   1000001100          0        0,   SULG# 1312312322          0          0          0

## IS THE RESULT CORRECT(0) OR ERROR(1) ##
%0
 --- 0


(NO.   2)

P#+1.
Q#-0.0
R#1.0

-- IMAGINARY ROOTS --

 6)   1000001001          0        0,   SULG# 1212331113 2100000000          0          0

## IS THE RESULT CORRECT(0) OR ERROR(1) ##
%0
 --- 0


(NO.   3)

P#.0
Q#.0
R#+0.0

-- INCONSISTENT --

 2)   1110100000          0        0,   SULG# 1121121231 1000000000          0          0

## IS THE RESULT CORRECT(0) OR ERROR(1) ##
%1
 --- 1


(NO.   4)

P#-0.
Q#1.1
R#1.

-- SIMPLE REAL ROOT --

 3)   1100010000          0        0,   SULG# 1311132211 2000000000          0          0

## IS THE RESULT CORRECT(0) OR ERROR(1) ##
%0
 --- 0


(NO.   5)

P#+1.0
Q#-.1
R#-0.

-- TWO REAL ROOTS --

 4)   1000001100          0        0,   SULG# 1232132231 1000000000          0          0
```

```
(NO.  11)

P#-1.
Q#1.0
R#+0.0

-- TWO REAL ROOTS --

 4)   1000001100          0        0,   SULG# 1312132123 1100000000          0          0

## IS THE RESULT CORRECT(0) OR ERROR(1) ##
%0
 --- 0


(NO.  12)

P#+1.0
Q#+1.0
R#-0.

-- TWO REAL ROOTS --

 4)   1000001100          0        0,   SULG# 1232123213 1100000000          0          0

## IS THE RESULT CORRECT(0) OR ERROR(1) ##
%0
 --- 0

GENERATED GIVEN NUMBER OF DATUM

(( UNUSED PATHS ))

 1)   1111000000          0        0

(( PATH MATRIX ))

 2)   1110100000          0        0 / 1 / 1 // 1121121231 1000000000          0          0 (  1 )

 3)   1100010000          0        0 / 0 / 1 // 1311132211 2000000000          0          0

 4)   1000001100          0        0 / 0 / 6 // 1312312322          0          0          0

 5)   1000001010          0        0 / 0 / 2 // 1131231121 1000000000          0          0

 6)   1000001001          0        0 / 0 / 2 // 1212331113 2100000000          0          0

## CONTINUE OR END ##
%0

          CONTINUE

## HOW MANY DATUM SHALL I GENERATE FURTHER (2 FIGURES) ##
% 8

          8 DATUM


(NO.   1)
     :
(NO.   8)

P#+1.0
Q#-.0
R#-1.

-- INDETERMINATE --

 1)   1111000000          0        0,   SULG# 1221321312          0          0          0

## IS THE RESULT CORRECT(0) OR ERROR(1) ##
%1
 --- 1

USED ALL PATHS

(( PATH MATRIX ))

 1)   1111000000          0        0 / 1 / 1 // 1221321312          0          0          0 (  1 )

 2)   1110100000          0        0 / 1 / 1 // 1121121231 1000000000          0          0 (  2 )

 3)   1100010000          0        0 / 0 / 3 // 1311132211 2000000000          0          0

 4)   1000001100          0        0 / 0 / 9 // 1312312322          0          0          0

 5)   1000001010          0        0 / 0 / 3 // 1131231121 1000000000          0          0

 6)   1000001001          0        0 / 0 / 3 // 1212331113 2100000000          0          0

## CONTINUE OR END
%END

          END

## GO TO 2ND STEP(2), GO TO KB(3) OR STOP(4) ##
%2


¥¥ 2ND STEP ¥¥

## SELECT ERROR SULG IN SULG NUMBER (2 FIGURES) ##
% 1


FUNDAMENTAL SULG   1221321312          0          0          0 (  1 )

## HOW MANY DATUM SHALL I GENERATE (2 FIGURES) ##
% 0

          8 DATUM


(NO.   1)
```

4. あとがき

プログラムの検査をすべて機械化することは困難であり，最後の判定はやはり人間が行わねばならないであろう．この判定を下すまでの過程の中には機械の方がより適切に行える部分があるように思われる．検査という仕事において，このような部分を積極的に計算機に行わせようとした試みが本研究である．

検査の対象としているプログラムの構造を考慮に入れながらテストプログラムを発生させ，それぞれのテストプログラムの処理結果をまとめて後で評価する方法（実験例１），およびテストプログラムを発生させる毎に，テストプログラムの処理結果を人間と計算機が会話する形で評価し，目的に合った学習を行って行く方法（実験例２）を開発し，これらの方法により誤り箇所が発見できることについて述べた．

残された問題としては，更に広範囲への応用のために文法の表現方法，学習方法，被テストプログラムの有向グラフを自動的に決める方法などが考えられる．これらの研究および開発により，より充実したテストシステムが期待できるであろう．

参 考 文 献

1) Burkhardt，W.H. ：Generating Test Programs from Syntax，Computing，2，53-73(1967).

2) 安井，藤田，泉：テストプログラム・ジェネレータによるシステムプログラムの検査の機械化，昭和４４年度情報処理学会大会講演予稿集

3) Feldman，J. and Gries，D. ：Translator Writing Systems，Comm. ACM，11，77-113 (1968).

4) Bush，R.R. and Mosteller，F. ：Stochastic Models for Learning，John Wiley & Sons，Inc. ，New York (1955).

5) 藤田：システムプログラムの検査の機械化に関する研究，昭和４５年度大阪大学修士論文

6) 大泉，高橋：JISに準拠したFORTRAN基本コース，オーム社，131(1968).

7) 安井，他：タイムシェアリング・システムによるソフトウェアの検査の機械化，「計算システムとの遠隔タイプライターによる会話言語系の研究」研究班研究会資料，43-59 (1972).

8) 安井，治田，永田，藤田：システム・プログラムの検査の機械化，情報処理学会関西支部プログラミング言語研究会論文集(1971).

本 PDF ファイルは 1965 年発行の「第 6 回プログラミング―シンポジウム報告集」をスキャンし、項目ごとに整理して、情報処理学会電子図書館「情報学広場」に掲載するものです。

この出版物は情報処理学会への著作権譲渡がなされていませんが、情報処理学会公式 Web サイトの https://www.ipsj.or.jp/topics/Past_reports.html に下記「過去のプログラミング・シンポジウム報告集の利用許諾について」を掲載して、権利者の捜索をおこないました。そのうえで同意をいただいたもの、お申し出のなかったものを掲載しています。

過去のプログラミング・シンポジウム報告集の利用許諾について

情報処理学会発行の出版物著作権は平成 12 年から情報処理学会著作権規程に従い、学会に帰属することになっています。

プログラミング・シンポジウムの報告集は、情報処理学会と設立の事情が異なるため、この改訂がシンポジウム内部で徹底しておらず、情報処理学会の他の出版物が情報学広場 (=情報処理学会電子図書館) で公開されているにも拘らず、古い報告集には公開されていないものが少からずありました。

プログラミング・シンポジウムは昭和 59 年に情報処理学会の一部門になりましたが、それ以前の報告集も含め、この度学会の他の出版物と同様の扱いにしたいと考えます。過去のすべての報告集の論文について、著作権者（論文を執筆された故人の相続人）を探し出して利用許諾に関する同意を頂くことは困難ですので、一定期間の権利者捜索の努力をしたうえで、著作権者が見つからない場合も論文を情報学広場に掲載させていただきたいと思います。その後、著作権者が発見され、情報学広場への掲載の継続に同意が得られなかった場合には、当該論文については、掲載を停止致します。

この措置にご意見のある方は、プログラミング・シンポジウムの辻尚史運営委員長 (tsuji@math.s.chiba-u.ac.jp) までお申し出ください。

加えて、著作権者について情報をお持ちの方は事務局まで情報をお寄せくださいますようお願い申し上げます。

期間：2020 年 12 月 18 日 〜 2021 年 3 月 19 日
掲載日：2020 年 12 月 18 日

プログラミング・シンポジウム委員会

情報処理学会著作権規程
https://www.ipsj.or.jp/copyright/ronbun/copyright.html