関数の回帰的なネットワークを用いた言語判定装置の生成

矢吹太朗† 伊庭斉志†

進化計算によって解が未知の問題のためのプログラムを自動生成する場合，その個体表現はチューリング完全でなければならない．しかしながら，標準的な遺伝的プログラミング（GP）において用いられる木構造表現はチューリング完全ではなく，チューリング完全にするためにはさまざまな拡張が必要である．我々は関数の回帰的なネットワークを表現として用いることを提案する．これによって新たに特殊な非終端記号を導入することなしに，GP をチューリング完全にすることができる．さらに，この表現を言語判定装置の生成に応用し，成功した結果も示す．

Language Decider Creation Using Recurrent Networks of Functions

Taro Yabuki† and Hitoshi Iba†

In generating a program automatically, the representation of the program must be Turing-complete if we do not know whether the problem is solvable or not in advance. However, a tree structure used by the standard Genetic Programming (GP) is not Turing-complete. We propose a representation scheme, which is a recurrent network of functions. It makes GP Turing-complete without introducing any new non-terminals. We empirically show how it succeeds in evolving language deciders.

1. はじめに

進化計算とは，良い解の候補を組み合わせたり，変異させたりすることを繰り返しながら，より良い解に到達することを目指す探索・設計手法である．この手法によって，プログラムや手続きを生成するのが遺伝的プログラミング（Genetic Programming，GP）である[6]．広く普及している標準的な GP においては，プログラムや手続きのための表現形式は単一の構文木である．構文木は与えられた非終端ノード（例：+，−，×，/）と終端ノード（例：x，y，0〜10）を組み合わせて作られる（例：$(+\ (/\ (-\ y\ 2)\ 5)\ x)$）．

本論文では単一の構文木に代わる個体表現を提案する．関数の回帰的なネットワーク（recurrent network of trees，RTN）である．

GP を用いる際には，個体表現（個体のデータ構造とその構成要素，利用方法，進化オペレータ）や各種パラメータなど，ユーザが選択しなければならない要素が数多くある．これらを決定する戦略は，目的のタスクによって大きく異なる．

GP のタスクは次のように分類することができる．

(1) 人間が簡単に書けるプログラム
(2) 人間が書いたものよりも単純なプログラムや効率の良いプログラム
(3) 未解決の問題を解くプログラム

タスクが (1) や (2) に属しているならば，GP のための各種設定は既知の解を参考に決めることができる．一方，タスクが (3) に属している場合にはそれらの選択は難しい．小さい命令セットや表現力を採用すれば探索が容易になる可能性があるが，探索に失敗した場合に，その原因が GP にあるのかその設定にあるのかを知ることはできない．

そこで，単純なものから始め，徐々に複雑な命令セットその他を導入していくというアプローチが考えられる．たとえば，四則演算のような関数のみで構文木を構成することから始め，繰返しや再帰などを順次導入していくという手法が提案されている[6]．このような拡張法は先述のクラス (1)，(2) の場合には採用しやすいが，(3) の場合には採用しにくい．繰返しや再帰などを導入することが，個体の表現力にどう影響するのかが分かりにくいためである．(3) に属するタスクについては，表現力が増加することが確認できるようなアプローチが望ましい．そして，表現力は最終的にはチューリング完全，つまりすべてのアルゴリズム（チューリング・マシン）を表現可能にならなければ

† 東京大学大学院新領域創成科学研究科基盤情報学専攻
Department of Frontier Informatics, Graduate School of Frontier Sciences, The University of Tokyo

ならない．

GP のための個体表現にはほかにもいくつかの要求がある．膨大な数の終端記号などを必要としない単純さや，新しい非終端記号の導入の容易さである．また，標準的な GP において用いられる表現である構文木の自然な拡張であることが望ましい．RTN はこれらの要求を満たす個体表現である．

本論文の構成は以下のとおり．2 章では標準的な GP 個体の表現力を確認する．3 章では RTN を実例とともに定義する．4 章では RTN のチューリング完全性を示す．5 章では RTN のための進化オペレータを示す．6 章では RTN を言語判定装置生成に応用する方法とその結果を示す．7 章では既存の研究との比較を中心にそれまでの節について考察する．

2. GP 個体の表現力

2.1 計算可能性の理論

計算機の能力はそのモデルによってさまざまなクラスに分けられることが計算可能性の理論によって知られている[10]．レパートリの小さいものから順に不可能なことの例とともにあげると以下のようになる☆．

表 入力に対し表を検索して答える．例として，正規表現を受理するということはできない．

有限オートマトン 有限の内部状態を持った機械である．例として，括弧検査のようなことはできない．

プッシュダウン・オートマトン 無限のスタックを持った有限オートマトン．例として，$\{ww|w \in \{0,1\}^*\}$ を受理するということはできない．

チューリング・マシン 前後に動いて読み書きできる無限に長いテープを持った有限オートマトン．現在使われている一般的な計算機と等価．例として，任意のチューリング・マシンが停止するかどうかを判定することはできない．

最も広いレパートリを持つチューリング・マシンにも不可能なことはあるが，機械的な手順でできることはすべてチューリング・マシンでできることだと考えられている（チャーチ・チューリングの提唱）．

2.2 GP 個体の表現力

GP 個体の表現力はその構成要素や使い方による．ここでは例として単一の構文木を個体表現に用いる場合を考える．

☆ 有限の場合に限るならば，このように区別することはできない．たとえば，あらかじめ入力サイズの上限を指定しておくなら，有限オートマトンで $\{ww|w \in \{0,1\}^*\}$ を受理することができる．しかし，そのための有限オートマトンの記述はチューリング・マシンのものに比べて非常に大きくなる．つまり，有限の場合にはこれらの区別は可能性ではなくエレガントさを表すことになる．

表 1 GP 個体の表現力
Table 1 Expressiveness of the standard GP.

クラス	繰返し評価	記憶域
表	なし	なし
有限オートマトン	あり	有限
プッシュダウン・オートマトン	あり	無限スタック
チューリング・マシン	あり	無限メモリ

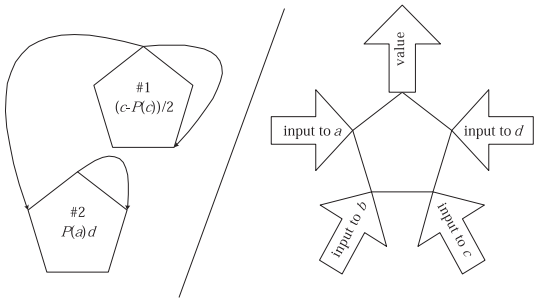

図 1 RTN の例（左）．右はノードのリンクと変数との対応の定義
Fig. 1 Example of RTN (left) and the relation between variables and link (right).

生成された個体を単独で完全なプログラムとして用いるならば，そのプログラムが扱える範囲は強く制限されたものになる．たとえば，四則演算と変数・定数のみから構成される構文木で表現される手続きは，有限オートマトンのレパートリでさえ含むことはできない．

構文木を繰り返し評価することを許せば可能性は広がる．有限の記憶領域があって，それにアクセスするノードを持つ構文木の評価を繰り返しながらタスクを実行するならば，そのプログラムは有限オートマトンと等価であることは明らかだろう．同様にスタックとそれを操作するノードがあれば，プッシュダウン・オートマトンのレパートリを含むことができる．無限の番地付メモリにアクセスするノードを持つ構文木の評価を，メモリがある値になるまで繰り返すような外部プログラムがあれば，システムはチューリング完全になる[11]．

つまり，**表 1** のように GP 個体の表現力を拡張していくことができる．しかしながら，PUSH や POP を構成要素としてプッシュダウン・オートマトンのレパートリを持たせた局面から，WRITE や READ を用いてチューリング・マシンのレパートリを持たせる局面に自然に移行させるのは難しい．それに対して，RTN は表からチューリング・マシンまで自然に計算のクラスを拡大していくことができる．

表 2 図 1 の RTN の時間発展
Table 2 Transition of RTN of Fig. 1.

タイムステップ	0	1	2	3	4
#1 の値	1011	101	10	1	0
#2 の値	1	1	1	0	0

3. 関数の回帰的なネットワーク

RTN を実例とともに概説する．**図 1** は RTN の一例である（P は 2 で割った余りを表す）．それぞれのノードは構文木で表現される純関数と値を持つ☆．関数は最大で 4 つの引数を持ち，引数は順番に a，b，c，d で表される．ノードの値は関数の引数に別のノードの値を代入し，関数を評価することによって決まる．

図 1 の RTN は次のように表現することもできる．

$$\begin{aligned} v_1^* &= (v_1 - P(v_1))/2, \\ v_2^* &= P(v_1)v_2. \end{aligned} \quad (1)$$

v_1 や v_2 などの終端記号は必要ないことに注意してほしい．これらの記号は見やすさのために用いているだけであって，必要な終端記号変数は a，b，c，d のみである．

プログラムは離散的なタイムステップに従って実行される．ある時刻 t における n 番目のノードの値を $v(n,t)$，そのノードへの入力となるノードの番号を $l_{n,i}$ とすると，$(t+1)$ における値は，

$$v(n,t+1) = f_n(v(l_{n,1},t),\cdots,v(l_{n,k},t)) \quad (2)$$

となる．ここで f_n はそのノードが表す関数，k はそのノードへの入力の数である．

今，ノードの値がともに 1 だとする．プログラムへの入力を 1 番目のノード（#1）の値とする．例として 2 進数 1011 を入力したときの動作は**表 2** のようになる．#1 の値が 0 になったとき，入力した数の 2 進数表記に 0 が含まれている場合に限り#2 の値が 0 になることが分かる．

3.1 RTN の記述

RTN は次の 6 要素を定めることで定義される．

(1) 式のリスト
(2) ネットワーク・トポロジ・ベクタ
(3) 初期状態
(4) 入力の仕方
(5) 停止条件
(6) 出力の得方

例として図 1 の RTN を考える．式のリストは，

$$\{(c - P(c))/2, P(a)d\} \quad (3)$$

である．

ネットワークは次のように記述される．#1 の 3 番目の引数（つまり c）に代入されるのは#1 の値であるから，#1 へのリンクは $\{*,*,1,*\}$ と記述する．$*$ の場所は任意の整数が入ることができ，RTN の振舞いはその値によらない（引数 a，b，d はないため，どのノードの値を入力することにしてもよい）．同様に，#2 へのリンクは $\{1,*,*,2\}$ と書ける．こうしてできるリンクの連結して $\{*,*,1,*,1,*,*,2\}$ のように書き直し，これをネットワーク・トポロジ・ベクタと呼ぶことにする．実際には*の部分にも数値が入り，$\{1,2,1,2,1,1,1,2\}$ などとなる．

初期状態においてはすべてのノードの値が 1 とする．入力は#1 の値として与える．停止条件は#1 の値が 0 になることとし，そのときの#2 の値を出力とする．以上で図 1 の RTN を完全に定義することができた．

3.2 標準的な GP との違い

RTN は標準的な GP の自然な拡張になっている．標準的な GP は個体の表現として単一の構文木を用いるが，RTN においては複数の構文木を用いる．構文木を構成する終端・非終端記号に特別なものは必要ないが，後で示すようにチューリング完全性のためには四則演算と P（2 で割った余り）が必要である．使用できる変数には制限があり，最大で 4 つ（a，b，c，d）である．標準的な GP においては，プログラム（手続き）への入力は変数に代入されるのに対し，RTN においては，ノードの値に代入される．

ここで扱うのは 1 入力のプログラムに限る．複数テープのチューリング・マシンをテープが 1 本のチューリング・マシンでシミュレートできるのと同様に，複数の入力を適当なコード化によって 1 つにまとめれば，複数入力のプログラムもここで扱う RTN で原理的には処理できる．しかし，その場合には利用者がコード化の方法を考えなければならない．また，4 より多いノードを用意して，複数の入力を複数のノードに割り当てることも可能である☆☆．

4. RTN のチューリング完全性

4 つのノードがあり，その構成要素として四則演算と P，定数（原理的には 1 だけでよい），4 つの変数があれば，RTN は任意のチューリング・マシン，つまり任意のアルゴリズムを表現できるようになる．証明は，まずチューリング・マシンを算術化し，算術化さ

☆ 本論文では見やすさのために構文木を S 式ではなく普通の数式で書く．

☆☆ ノードが $(6n+8)$ 個ある RTN ならば，n 本テープのチューリング・マシンをシミュレートできることが分かっている．

れたチューリング・マシンをシミュレートする RTN を実際に構成することで行う．

4.1 チューリング・マシンの算術化

チューリング・マシンのテープを整数としてとらえ，マシンの動作を整数の演算で記述するのがチューリング・マシンの算術化である[7]．以下では簡単のためにテープには 0 か 1 のみが書かれるようなチューリング・マシンを算術化する（こうしても一般性は失われない）．

チューリング・マシンが次のような状態にあり，ヘッドが右に動こうとしている（$D(q_i, s_j) = 1$）とする（左に動こうとする場合は $D(q_i, s_j) = 0$）．

$$\underbrace{\cdots\ b_3\ b_2\ b_1\ b_0}_{m}\ \underbrace{s_j}_{q_i}\ \underbrace{c_0\ c_1\ c_2\ c_3\ \cdots}_{n}$$

ここで s_j は現在読んでいるテープの文字，q_i はマシンの状態である．

$$s = s_j, \quad m = \sum_{i=0}^{\infty} b_i 2^i, \quad n = \sum_{i=0}^{\infty} c_i 2^i \qquad (4)$$

という記号を導入すれば，テープの状態は 3 つの数 (s, m, n) で表すことができる（1 の数は有限であるから上の和は実際には有限である）．さらにチューリング・マシンの状態 q_i を整数 q で表すことにすれば，結局チューリング・マシンの完全な状態を 4 つの数（q，s，m，n）によって表現することができる．

テープを $s_{ij} = R(q, s)$ に書き換え，ヘッドが右に動いた後のチューリング・マシンは，

$$\underbrace{\cdots\ b_3\ b_2\ b_1\ b_0\ s_{ij}}_{m^*}\ \underbrace{c_0}_{q_{ij}}\ \underbrace{c_1\ c_2\ c_3\ \cdots}_{n^*}$$

のようになる．このステップは，(q，s，m，n）の変換として記述することができる．この例では，新しい数（q^*，s^*，m^*，n^*）は，

$$\begin{aligned} q^* &= Q(q, s), \\ s^* &= P(n), \\ m^* &= R(q, s) + 2m, \\ n^* &= H(n) \end{aligned} \qquad (5)$$

となることが容易に分かる（$Q(q, s)$ は次の状態，$H(n)$ と $P(n)$ はそれぞれ n を 2 で割った商と余りである）．

一般の場合にはチューリング・マシンの変換を次のように表現できることはすぐに確かめられる．

$$\begin{aligned} q^* &= Q(q, s), \\ s^* &= P(m)(1 - D(q, s)) \\ &\quad + P(n)D(q, s), \\ m^* &= (R(q, s) + 2m)D(q, s) \\ &\quad + H(m)(1 - D(q, s)), \\ n^* &= (R(q, s) + 2n)(1 - D(q, s)) \\ &\quad + H(n)D(q, s). \end{aligned} \qquad (6)$$

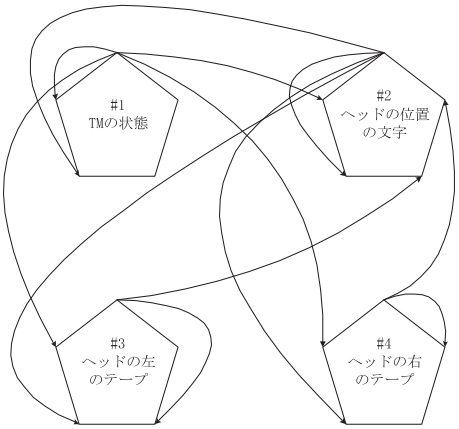

図 2 チューリング・マシンをシミュレートする RTN
Fig. 2 RTN to simulate TM.

表 3 RTN のノードとチューリング・マシンとの対応関係
Table 3 The relationship between TM and RTN.

	値の意味	式
#1	マシンの状態	$Q(a, b)$
#2	ヘッド位置の文字	$P(c)(1 - D(a, b))$ $+P(d)D(a, b)$
#3	ヘッドの左側のテープ	$(R(a, b) + 2c)D(a, b)$ $+H(c)(1 - D(a, b))$
#4	ヘッドの右側のテープ	$(R(a, b) + 2d)(1 - D(a, b))$ $+H(d)D(a, b)$

4.2 RTN によるチューリング・マシンのシミュレーション

算術化したチューリング・マシンは**図 2** のような RTN によってシミュレートすることができる．ノードが表す関数とチューリング・マシンとの対応関係は**表 3** のようになる．文字 a，b，c，d はそのノードが表す関数の引数の順番を表すだけであることに注意してほしい．つまり，#1 と#2 の a には関連はない．

最後にチューリング・マシンでは表として与えられる $Q(q, s)$，$R(q, s)$，$D(q, s)$ を四則演算から構成できることを確認する．内部状態の数を k とすると，次のような有理数，

$$A_{a,b} = \frac{Q(a, b)}{\prod_{i=0, i\neq a}^{k}(a - i)} \qquad (7)$$

を用いれば，$Q(q, s)$ を返すような関数 $f(q, s)$ を 1 つの式で書くことができ，

$$f(q, s) = (1 - s)\sum_{i=0}^{k}\left(A_{i,0}\prod_{j=0, j\neq i}^{k}(q - j)\right)$$

表 4 RTN の表現力
Table 4 Expressiveness of RTN.

	ノード数	変数の数	非終端記号
表	1	1	四則演算
FSA	2	2	四則演算・P
PDA	3	3	四則演算・P
TM	4	4	四則演算・P

$$+s\sum_{i=0}^{k}\left(A_{i,1}\prod_{j=0,j\neq i}^{k}(q-j)\right) \quad (8)$$

となる（s は 0 か 1，q は 0 から k までの整数をとることに注意）．例として $k=2$ の場合を書き下すと，

$$\begin{aligned}&(1-s)\,((q-2)\,(q-1)\,Q(0,0)/2\\&-(q-2)\,q\,Q(1,0)+(q-1)\,q\,Q(2,0)/2)\\&+s\,((q-2)\,(q-1)\,Q(0,1)/2\\&-(q-2)\,q\,Q(1,1)+(q-1)\,q\,Q(2,1)/2)\end{aligned} \quad (9)$$

となる☆．

以上から，4 つのノードからなる RTN で，それぞれのノードが表す関数の構成要素に四則演算と P，定数，4 つの変数があれば任意のチューリング・マシンをシミュレートできることが示された（これが最小セットだと主張しているわけではない）．

構成要素のセットを小さくすることにこだわらないのであれば，if のような非終端記号を導入して，以上の議論を省くこともできる．その場合，チューリング・マシンの遷移は，

$$\begin{aligned}q^* &= Q(q,s),\\ s^* &= \text{if } D(q,s)=0 \text{ then } P(m)\\ &\qquad \text{else } P(n),\\ m^* &= \text{if } D(q,s)=0 \text{ then } R(q,s)+2m\\ &\qquad \text{else } H(m),\\ n^* &= \text{if } D(q,s)=0 \text{ then } H(n)\\ &\qquad \text{else } R(q,s)+2n\end{aligned} \quad (10)$$

のように表される．$Q(q,s)$，$R(q,s)$，$D(q,s)$ の表現が簡単になるのは明らかだろう．

表や有限オートマトン（FSA），プッシュダウン・オートマトン（PDA）も同様の方法で RTN でシミュレートすることができる（**表 4**）．

4.3 ノードの値としてのリスト

これまではノードの値は数値のみであったが，数値のリストもとれるように RTN を拡張することができる．上述の証明中のチューリング・マシンの左右のテープ，m，n がリストで表現されているとして，チューリング・マシンの遷移を表す式を，

$$\begin{aligned}q^* &= Q(q,s),\\ s^* &= \text{if } D(q,s)=0\\ &\qquad \text{then } \mathrm{car}(m)\\ &\qquad \text{else } \mathrm{car}(n),\\ m^* &= \text{if } D(q,s)=0\\ &\qquad \text{then } \mathrm{cons}(R(q,s),m))\\ &\qquad \text{else } \mathrm{cdr}(m),\\ n^* &= \text{if } D(q,s)=0\\ &\qquad \text{then } \mathrm{cdr}(n)\\ &\qquad \text{else } \mathrm{cons}(R(q,s),n)\end{aligned} \quad (11)$$

と書き直せばよい．ただし car はリストの初めの要素を，cdr は残りの要素を，$\mathrm{cons}(a,b)$ はリスト b の先頭に a を加えたリストを返すような関数である（ここで a はリストではないとしてよい．ただし空リストにはなりうる☆☆）．リストが空のときは car，cdr ともに nil を返し，リストの要素が 1 つのときも cdr は nil を返す．

この変更によって，テープに新しい記号 nil が導入されるため，Q，D，R などを構成するためには if が必要不可欠になる．その結果，必要な非終端記号は四則演算と P，if，car，cdr，cons となる．これらを組み合わせて作られる構文木をつねに評価可能にするために，car の引数がリストでなく数値だった場合や，四則演算の引数がリストだった場合の規則を別に定めなければならない．ここでは標準的な規則に合わない引数が非終端記号に与えられた場合の評価値はその第 1 引数とすることにする．たとえば $\mathrm{plus}(4,\{1,0,1\})=4$，$\mathrm{car}(12)=12$ などである．

4.4 停止条件

RTN の停止条件はノードやネットワーク・トポロジとは別に与えなければならない．これは，チューリング・マシンの停止条件はその状態遷移関数とは別に与えなければならないのと同様である☆☆☆．

チューリング・マシンの停止条件は，停止状態を指定することで定められる．つまり，前述のチューリング・マシンならば，停止すべき q の値（複数も可）を設定し，q が実際にその値になったときに処理を停止させる．RTN の場合も同様に，停止条件の対象になるノードとその値を定め，そのノードを監視することで処理を停止させることにする．

☆ $Q(0,0)$ などの数は定数と四則演算で作ることができる．

☆☆ 以下では空リストを nil や {} で表す．

☆☆☆ ここではヘッドの移動に左右だけでなく停止も含むようなチューリング・マシンの定義は想定していない．

与えられたチューリング・マシンがある入力に対して停止するかどうかを決定することはできない．そのため，チューリング完全な個体を用いた進化計算には本質的な困難がある．解の完璧な評価は有限時間では行えないのである．しかしながら，進化計算によって実際に生成したいプログラムには，適当な時間内で実行できるという条件がつねにあるため，このことは実際には問題にならない．

5. 進化オペレータ

先述のとおり，RTN のデータ構造（構文木のリストとネットワーク・トポロジ・ベクタ）は次のように表現できる．

$$\{\ \{(c - P(c))/2, P(a)d\}, \\ \{1,2,1,2,1,1,1,2\}\ \}. \tag{12}$$

このデータ構造のためのさまざまな進化オペレータを導入することができる．例をあげると，

(1) 構文木の突然変異
(2) 構文木の交換
(3) リンクの突然変異
(4) リンクの交換
(5) ネットワーク・トポロジ・ベクタの交叉
(6) ノードの追加または削除

などがある．(1) と (3)，(2) と (4) は同時に行うことも考えられる．

ネットワーク・トポロジ・ベクタの交叉は遺伝的アルゴリズムで行われる交叉と同様である．ただし，親のノード数（ベクタのサイズ）は同じでなければならない．ノードの追加または削除オペレータによって個体のノード数は変化するが，集団内のすべての個体のノードが一定になるようにこのオペレータを用いていれば，この条件は満たされる．

ノード数が 4 以上ならば，それ以上ノードを追加しても RTN の表現力はチューリング完全のまま変化しない．しかしながら，ノードが増えることによって得られる冗長性のために探索性能が上がる可能性がある．

ネットワークのクラスタ（孤立した部分）を交換するようなオペレータも考えられる．しかしながら，ランダムに作った RTN のネットワーク（各ノードは 4 つの入力を持ち，その接続先は初めはランダムに決められる）がクラスタ化することはまれである．つまり，RTN のネットワークに孤立した部分はできにくい．そのため，クラスタの交換のようなオペレータはここでは導入しない☆．

表 5 Tomita languages
Table 5 Tomita languages.

言語	定義	
	長さ 10 以下の文字列数	トレーニング例の数
	正例 負例	正例 負例
L1	1^*	
	11 2036	9 8
L2	$(10)^*$	
	6 2041	5 10
L3	奇数個の 1 の直後は奇数個の 0 ではない	
	716 1331	11 11
L4	000 を部分文字列として含まない	
	1103 944	10 7

6. 例：言語判定装置の生成

例として，言語判定装置（入力した文字列がその言語に属するかどうかを決定する）の生成に応用した結果を示す．対象とした言語は Tomita languages である（**表 5**）．トレーニング例の数は文献 4) にならった．

6.1 探索の目標

探索の目標は次のように振る舞う RTN を生成することである．

(1) すべてのノードの値を nil にする．
(2) 入力用に決めたノードに，0，1 からなる文字列をタイムステップに従って 1 文字ずつ与える．
(3) 文字列がなくなったら nil を与える．
(4) それからある決まったステップ数以内で，ある決まったノードが判定結果を示す．つまり，入力した文字列が対象となる言語に属しているならば値が 1 に，そうでなければ 0 になる．

RTN の 6 要素のうち，式のリストとネットワーク・トポロジが探索の対象である．

ここでは 1 文字ずつ与える代わりに，文字列をリストとして一度に与えるようにする．そのために#1 と #2 はそれぞれ $\mathrm{cdr}(a)$，$\mathrm{car}(a)$ とし，a には#1 の値が入力されるようにしておく．これによって，#1 は入力された文字列を 1 タイムステップに 1 ずつ短くした文字列を保持し，#2 は入力された文字列の文字を順番に受け取ることになる．

残りの 4 要素のうち，初期状態と入力の仕方，出力の得方は上記のように問題の制約に入れ，これらも探索させるという方法は採らない．停止条件，つまりど

☆ 式の中に変数が存在しなかったり，$(a - a)$ における a のように式を整理することによってその変数が消えたりすることがある．このようにして無意味になるリンクの存在や，停止までに影響を及ぼしあわない距離にあるノードがあることまで考慮すれば，ネットワークはクラスタ化される．ただし，そのようにして分かるクラスタを調べる計算量は小さくない．

表 6 正解の遷移
Table 6 Transition of solution.

	ノード				
t	#1	#2	···	#n	···
0	{1,0,1,1}	{}		{}	
1	{0,1,1}	1			
2	{1,1}	0			
3	{1}	1			
0	{}	1			
1	{}	{}			
⋮					
i	{}	{}		1	

のノードがいつ答えを与えることになるのかはトレーニングの過程で決める．他の方法として，チューリング・マシンのように，指定したノードが指定した値になったときの別のノードの値を出力とするということも可能である．

6.2 停止条件の決定・適合度関数

最も良い確率で言語を判定しているノードとタイムステップを求めることによって，停止条件を決定する．停止条件を決定する過程で適合度も計算する．具体的には次のように行う．

解の候補となる RTN に対し，正負すべてのトレーニング例を入力する．それぞれの入力に対し，

(1) #1 の値が nil になるまで RTN を遷移させる．その時点を新たに $t = 0$ とする．
(2) タイムステップ t がノード数になるまで RTN を遷移させる（これによって k 番目のトレーニング例を入力した場合のタイムステップ t における n 番目のノードの値 $v(k, n, t)$ の表が得られる）．

タイムステップ t の最大値をノード数にすることは，文字列がすべて入力されてからその影響が全体に行き渡るまでに判定結果を出さなければならないということを意味する．なぜなら，ノード数は変化の影響がネットワーク全体に行き渡るのに必要な時間の上限だからである．

次に，ノードが言語の判定結果を持っているかどうかを調べる．そのために，次のような関数 g を定義する．番号 k で表される正例［負例］を入力したとする．タイムステップ t において，n 番目のノードの値が 1［負例の場合は 0］ならば $g(k, n, t) = 1$，そうでないならば $g(k, n, t) = 0$ とする．

最も良い確率で判定に成功したノードとタイムステップを求める．つまり，$\Sigma_k\, g(k, n, t)$ が最大になるような n と t を求める．

正解の場合，RTN の遷移は**表 6** のようになるはずである．ノード #1 の値が nil（表中では {}）になるまで RTN を遷移させ，$t = 0$ とする（水平線の下）．ここまでのステップ数（ここでは 3）は入力によって異なる．その後 i ステップ目で#n の値が判定結果を表す（この例では 1 であるから正例と判断したことを意味している）．数 i や n は入力に依存してはならない．

表 7 進化計算の設定
Table 7 Setting of evolutionary computation.

集団数	1,000（L1，L4）または 5,000（L2，L3）
RTN のノード数	20
進化オペレータ	構文木のランダムに生成した構文木による置き換え，ネットワーク・トポロジ・ベクターの（1 点のみでの）突然変異・一点交叉
選択法	トーナメント（サイズ 5）
終端記号	a，b，c，d，nil，0〜10 の整数
非終端記号	plus，minus，times，divide，p，if，car，cdr，cons

適合度は，この（トレーニングの）成功率，つまり $(\Sigma_k\, g$ の最大値 $)$/(トレーニング例の数) とする．トレーニング例は世代ごとに長さ 10 以下のものをランダムに作る．

6.3 進化計算の設定

進化計算の設定は**表 7** のとおりである．

3 つの進化オペレータで子孫の 1/3 ずつを作る．divide は 0 除算のチェックを行い，2 番目の引数が 0 ならば 1 番目の引数を返す．p は引数が整数ならばそれを 2 で割った余りを返し，それ以外ならば引数をそのまま返す．if は次のように定義される．

$$\mathrm{if}(a, b, c, d) := \mathrm{if}\ (a = b)\ \mathrm{then}\ c\ \mathrm{else}\ d. \quad (13)$$

RTN の遷移を計算する過程において，ノードの値が 10^{10} より大きい場合には値を強制的に 0 にする．この制限は，チューリング・マシンの無限のテープを実際には扱えないことに相当する．

6.4 結　　果

長さ 10 以下の文字列をトレーニング例として用いて進化計算を行った．生成された判定装置の適合度は 1，つまり長さ 10 以下の文字列をすべて正しく判定した．

結果をノードの書き換え規則の形で示す（判定結果を保持するノードと因果関係を持つノードのみ）．v_i などの終端記号が必要なわけではないことに注意してほしい．この記法は見やすさのためだけに用いている．実際に必要な終端記号変数は a，b，c，d の 4 つのみである．

6.4.1 L1

1,000 個体を用いた進化計算で，5 世代目で生成さ

表 8 L1 のための判定装置
Table 8 Language decider for L1.

$v_1^* = \mathrm{cdr}(v_1)$
$v_2^* = \mathrm{car}(v_1)$
$v_9^* = \mathrm{times}(\mathrm{if}(v_{19}, \mathrm{divide}(v_{19}, \mathrm{minus}(\mathrm{divide}(v_{19}, \mathrm{cdr}(\mathrm{if}(v_{19}, v_{15}, \mathrm{times}(v_{20}, v_{15}), \mathrm{cdr}(v_{19})))), v_{19})), 1, v_{20}), v_{15})$
$v_{15}^* = v_{17}$
$v_{17}^* = \mathrm{times}(v_{19}, v_{17})$
$v_{19}^* = v_2$
$v_{20}^* = v_{19}$

表 9 L2 のための判定装置
Table 9 Language decider for L2.

$v_1^* = \mathrm{cdr}(v_1)$
$v_2^* = \mathrm{car}(v_1)$
$v_{10}^* = \mathrm{cdr}(\mathrm{p}(v_{16}))$
$v_{16}^* = \mathrm{minus}(\mathrm{divide}(6, v_1), \mathrm{times}(\mathrm{car}(\mathrm{p}(\mathrm{times}(v_2, 3))), v_{10}))$
$v_{18}^* = \mathrm{times}(v_{20}, v_{10})$
$v_{20}^* = \mathrm{cdr}(\mathrm{p}(\mathrm{if}(\mathrm{cons}(\mathrm{car}(v_2), v_2), \mathrm{cons}(\mathrm{p}(\mathrm{if}(v_{18}, v_2, \mathrm{cons}(\mathrm{cons}(4, \mathrm{car}(v_2)), \mathrm{divide}(v_2, v_2)), \mathrm{minus}(\mathrm{car}(\mathrm{minus}(v_2, v_2)), \mathrm{car}(\mathrm{plus}(\mathrm{divide}(0, 3), v_2))))), v_{18}), \mathrm{plus}(4, v_2), \mathrm{minus}(9, v_2))))$

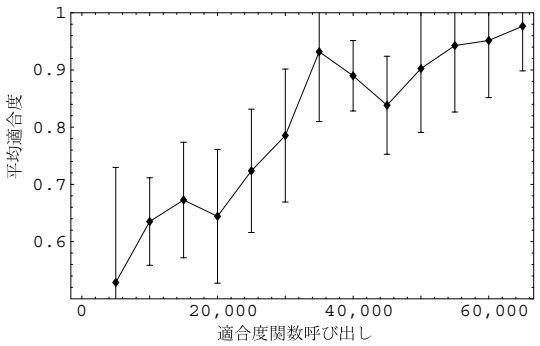

図 3 適合度の変化の様子（L2 の場合）
Fig. 3 Evolution of the fitness of the RTN for L2.

表 10 L3 のための判定装置
Table 10 Language decider for L3.

$v_1^* = \mathrm{cdr}(v_1)$
$v_2^* = \mathrm{car}(v_1)$
$v_4^* = \mathrm{times}(v_{17}, v_{10})$
$v_6^* = \mathrm{if}(v_{17}, 1, v_{10}, v_4)$
$v_8^* = \mathrm{times}(\mathrm{cdr}(v_2), v_{17})$
$v_{10}^* = v_6$
$v_{17}^* = \mathrm{p}(\mathrm{plus}(5, \mathrm{minus}(v_2, v_8)))$

表 11 L4 のための判定装置
Table 11 Language decider for L4.

$v_1^* = \mathrm{cdr}(v_1)$
$v_2^* = \mathrm{car}(v_1)$
$v_3^* = v_{20}$
$v_5^* = v_6$
$v_6^* = \mathrm{cdr}(\mathrm{plus}(v_{19}, v_{19}))$
$v_9^* = \mathrm{if}(1, v_{16}, v_{16}, \mathrm{if}(v_{10}, \mathrm{p}(v_{16}), 3, 8))$
$v_{10}^* = \mathrm{if}(\mathrm{times}(1, v_{15}), v_5, \mathrm{cons}(v_3, v_3), v_{10})$
$v_{12}^* = v_9$
$v_{13}^* = \mathrm{cdr}(v_{19})$
$v_{15}^* = v_{20}$
$v_{16}^* = v_2$
$v_{19}^* = \mathrm{p}(v_{12})$
$v_{20}^* = \mathrm{if}(v_{12}, v_{20}, v_{20}, v_{13})$

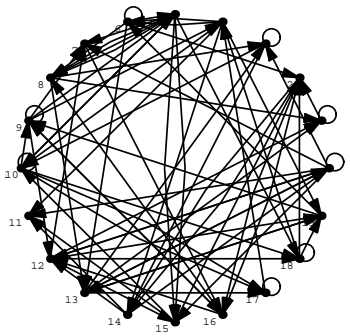

図 4 生成された L4 のための判定装置
Fig. 4 Generated RTN for L4.

れた L1 のための判定装置が**表 8** である．v_1 が nil になってから 4 ステップ目における v_9 の値が判定結果になる．

6.4.2 L2

5,000 個体を用いた進化計算で，14 世代目で生成された L2 のための判定装置が**表 9** である．v_1 が nil になってから 2 ステップ目における v_{18} の値が判定結果になる．適合度の変化の様子は**図 3** のとおりである（エラー・バーは標準偏差）．

6.4.3 L3

5,000 個体を用いた進化計算で，249 世代目で生成された L3 のための判定装置が**表 10** である．v_1 が nil になってから 4 ステップ目における v_4 の値が判定結果になる．

6.4.4 L4

1,000 個体を用いた進化計算で，40 世代目で生成された L4 のための判定装置が**表 11** である．v_1 が nil になってから 9 ステップ目における v_{19} の値が判定結果になる．

先述のように RTN のネットワークはそのままではクラスタ化していないが（**図 4**），式を整理して無意味なリンクを取り除くとクラスタが現れる（**図 5**）．

6.5 検　　証

進化計算は長さ 10 以下の文字列をトレーニング例にして行った．検証のために，生成された RTN で長さ

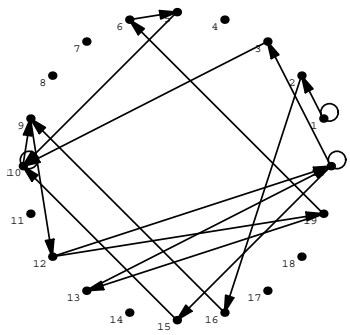

図 5 判定に関わるノードだけを残した結果
Fig. 5 Essential part of RTN of Fig. 4.

16 以下の文字列を判定することを試みた．その結果，すべての文字列を正しく判定することが確認できた．

7. 考　　察

7.1 チューリング完全性およびグラフを用いた表現

GP のための個体表現はすでに数多く提案されているが，チューリング完全性とグラフ表現の 2 点で関連するものとしては，次のようなものがある．

標準的な GP において，個体は単一の構文木であってその構成要素は非終端記号（関数）と終端記号（変数と定数）である．個体への入力を変数に代入し，構文木を評価した結果が個体の出力になる．データ構造はそのまま（構文木）に，番地付きメモリにアクセスするための手続き（READ，WRITE）を構成要素に加え，特定のメモリが特定の値になるまで構文木の評価を繰り返すような利用方法を採用すれば，その個体表現はチューリング完全になることが示されている[11]．

チューリング完全な表現としてはほかにもセル・オートマトンや recurrent artificial neural network などがある．これらの方法には新しい機能を導入するのが難しいという欠点がある．これに対して構文木を用いる GP は非終端記号を増やすことで容易に機能を追加できる．

個体表現にグラフ構造を用いるものとしては GP-automata[2] や Parallel Distributed Genetic Programming（PDGP）[9]，PADO[12]，Multiple Interacting Programs（MIPs）[1] などがある．またグラフ構造を用いているがそのノードは構文木でないものとしては Linear-Graph GP[5] がある．

RTN と GP-automata の大きな違いはその表現力である．GP-automata のノードの 1 つ 1 つが有限オートマトンの状態に対応する．そのため，GP-automata の表現力は有限オートマトンと同等である．

RTN と PDGP の大きな違いは，PDGP はそのノードが非終端記号であるということである．RTN はそのノードが構文木で，そのために本論文で見たように計算可能性の議論が非常に単純になり，そのチューリング完全性を簡単に示すことができる．PDGP の表現力は明らかではない．

RTN と PADO との大きな違いは，PADO は外部に番地付きメモリを持っていることである．ネットワーク内を流れるデータは，ある場合は処理される対象として，別の場合はメモリのアドレスとして扱われることになる．そのため独自の非終端記号を導入する際には，それがメモリのアドレスの計算にも使われることを考慮しなければならない．これに対して，RTN は外部メモリを持たない．ノードの変数には別のノードの値が割り当てられるが，どのノードの値が割り当てられるかは固定されている．また，“n 番目のノードの値を読む” というような非終端記号を含むことはできない．RTN の非終端記号は純関数に限られる．このため，ユーザは非終端記号の導入に際して上で述べたようなことを考慮する必要はない．

7.1.1 Multiple Interacting Programs

MIPs は RTN と非常によく似たシステムであるが，次のような 2 つの相違点とそれにともなう欠点がある．

まず，RTN では他のノードの値と変数をリンクという形でつないでいるのに対し，MIPs においてはノードの数だけ終端記号を用意し，その記号によって他のノードの値を利用するようになっている．そのためネットワークが大きくなると，膨大な数の終端記号が必要になる．RTN で必要な定数以外の終端記号は 4 つだけである．

次に，MIPs においては，ノード数やノードごとのリンク数を決める具体的な指針や，チューリング完全にするための非終端記号その他の必須条件が与えられていない．そのため，適切な非終端記号を与えられず，探索が困難になる可能性がある．たとえばビット反転装置の生成において，非終端記号に sin や cos を用いているために生成された結果は複雑なうえに 5 ビットに限定されたものになっている[1]．ここで扱った RTN ならば任意の長さのビット列を扱える反転装置を簡単に生成できることを実験で確認している．しかも，その際に用いた非終端記号はチューリング完全性のために必須なものだけである．

7.2 言語判定装置生成

Tomita language L4 の判定装置の生成は Giles らよっても報告されている[3]．しかしながら，彼らの結果は制限された表現力を用いて得られたものである．また，この問題は GP と recurrent neural network を

統合したシステム，ℜ-STROGANOFF でも試みられている[4]．そこで得られた判定装置は長さ 10 以下の文字列の 91%を正しく判定するものである．それに対して，ここで得られた判定装置は長さ 16 以下の文字列をすべて正しく判定する．この結果は，この問題については我々の方法が優れていることを示している．

Tomita language のための判定装置は有限オートマトンで実現できる．このように，必要な計算のクラスがすでに分かっている現実の問題に適用する場合には，表現力をそのクラスに限定して探索した方が成功しやすいだろう．それにもかかわらず，チューリング完全な表現を用いて探索することの意義は 2 つある．

第 1 に，これは進化計算によってプログラムを自動生成する一般的な手法の研究である．先にも述べたように，プログラム生成において用いる表現はチューリング・マシンのレパートリを含んでいなければならない．なぜなら，レパートリの制限された表現しか扱えない手法は，解がその範囲に入っていることが分からない限りは利用できないからである．

第 2 に，チューリング完全な表現を用いた探索に成功することは，より少ないヒントのもとでの探索の成功を意味する．制限された表現を用いて探索することは，（意図せずに）大きなヒントを与えることになる．探索は人間がヒントをあまり与えずに行えることが一般的には望ましい．定量的に示すのは難しいが，チューリング完全な表現を用いる探索は，制限された表現を用いる場合に比べてヒントは少ないといっていいだろう．命令セットが小さければなおさらである．

8. 今後の課題・おわりに

進化計算によってプログラムを自動生成する場合に用いる表現（関数の回帰的なネットワーク，recurrent network of trees，RTN）を提案し，それがチューリング完全であることを示した．例として Tomita language のための判定装置の生成に応用し，GP を用いて従来得られていた結果よりも良い判定装置が得られた．しかしながら，このタスクは制限された表現力（有限オートマトン）の個体表現でも達成できる．そのような制限のある個体表現では本質的に不可能な，つまりチューリング完全な表現力を必要とするタスクに応用することが今後の課題である．

RTN のノード数をはじめとして探索効率に影響する可変パラメータが存在することはすでに分かっている．これらについての理論および実験的研究も今後の課題である．

ここで用いたネットワークはノード間をランダムにつないで作られるものだが，秩序あるネットワークによく見られるスケールフリー性などの導入も検討の余地があるだろう．

チューリング完全な表現を用いた進化計算の例はまだ少ない[8),11)]．進化計算によるプログラムの自動生成の実現のためにはさらなる研究が必要である．

参 考 文 献

1) Angeline, P.J.: Multiple interacting programs: A representation for evolving complex behaviors, *Cybernetics and Systems*, Vol.29, No.8, pp.779–806 (1998).
2) Ashlock, D.: GP-automata for dividing the dollar, *Genetic Programming 1997: Proc. 2nd Annual Conference*, pp.18–26 (1997).
3) Giles, C.L., Miller, C.B., Chen, D., Chen, H.H., Sun, G.Z. and Lee, Y.C.: Learning and extracting finite state automata with second-order-recurrent neural networks, *Neural Computation*, Vol.4, pp.393–405 (1992).
4) Iba, H., de Garis, H. and Sato, T.: Temporal data processing using genetic programming, *Proc. 6th International Conference ICGA-95*, pp.279–286 (1995).
5) Kantschik, W. and Banzhaf, W.: Linear-graph GP — a new GP structure, *Genetic Programming, Proc. 5th European Conference, EuroGP 2002*, Vol.2278 of LNCS, pp.83–92, Springer-Verlag (2002).
6) Koza, J.R., Andre, D., Bennett III, F.H. and Keane, M.: *Genetic Programming III: Darwinian Invention and Problem Solving*, Morgan Kaufman (1999).
7) Minsky, M.L.: *Computation: Finite and Infinite Machines*, Prentice-Hall, Inc. (1967). 金山裕（訳）：計算機の数学的理論，近代科学社 (1970).
8) Nordin, P. and Banzhaf, W.: Evolving turing-complete programs for a register machine with self-modifying code, *Genetic Algorithms: Proc. 6th International Conference* (*ICGA95*), pp.318–325 (1995).
9) Poli, R.: Evolution of graph-like programs with parallel distributed genetic programming, *Genetic Algorithms: Proc. 7th International Conference*, pp.346–353 (1997).
10) Sipser, M.: *Introduction to the Theory of Computation*, Thomson Learning (1997). 渡辺 治 ほか（訳）：計算理論の基礎，共立出版 (2000).
11) Teller, A.: Turing completeness in the language of genetic programming with indexed memory, *Proc. 1994 IEEE World Congress on Computational Intelligence*, Vol.1, pp.136–141 (1994).

12) Teller, A. and Veloso, M.: PADO: A new learning architecture for object recognition, Ikeuchi, K. and Veloso, M. (Eds.), *Symbolic Visual Learning*, pp.81–116, Oxford University Press (1996).

(平成 15 年 4 月 11 日受付)
(平成 15 年 6 月 6 日再受付)
(平成 15 年 7 月 11 日採録)

矢吹　太朗（学生会員）

1976 年 1 月生．1998 年東京大学理学部天文学科卒業．1999 年東京大学大学院理学系研究科天文学専攻中退．2001 年東京大学大学院新領域創成科学研究科基盤情報学専攻修士課程修了，修士（科学）．現在，同専攻博士課程に在学．研究テーマは情報・進化・計算．

伊庭　斉志（正会員）

1962 年 10 月 28 日生．1985 年東京大学理学部情報科学科卒業．1990 年東京大学大学院工学系研究科情報工学専攻博士課程修了．工学博士．同年電子技術総合研究所入所．1998 年から東京大学大学院工学系研究科電子情報工学専攻助教授．1999 年から東京大学大学院新領域創成科学研究科基盤情報学専攻助教授．進化システムおよび人工知能基礎の研究に従事．特に遺伝的プログラミング，学習，推論，知能ロボットに興味を持つ．