学位論文
進化計算による自動プログラミングのための表現の研究

矢吹 太朗
東京大学大学院 新領域創成科学研究科 基盤情報学専攻

概要

遺伝的プログラミング (Genetic Programming, GP) のための新しい表現形式を提案する。関数の回帰的なネットワーク (Recurrent Network consisting of Trees, RTN) である。GP は進化計算 (evolutionary computing, EC) の一種で、プログラムの自動生成に用いられる。EC は自動的な最適化やデザインのフレームワークである。

EC は次の要素で特徴付けることができる。

- 解候補の表現形式と解候補を改変するためのオペレータ
- 解候補の質を定義する評価関数
- 解候補の集合の中から有望なものを選び出す選択方法

通常これらの特徴は独立に定義される。このようなモデル自体を考え直すのも興味深いことではあるが、ここでは GP のための表現形式のみを研究対象とする。

GP を利用する際に、われわれは RTN を用いてプログラムを表現する。RTN は任意のアルゴリズムを表現できる、つまりチューリング完全な表現形式である。そのため、RTN の利用者は問題の解が RTN で表現可能かどうかを心配する必要はない。その一方で、最も普及している標準的な GP においては、プログラムは単一の構文木で表現され、その表現力は強く制限されたものになっている。たとえば、構文木を構成する非終端記号が純関数で、構文木を評価した結果をプログラムの出力（または振舞い）と見なすならば、この表現のレパートリは有限オートマトンのものよりも小さくなる。

制限された表現力でも与えられた問題の解を記述するには十分だということを、進化計算に先立って知っているならば、このことは問題にはならない。しかしながら、もしこの十分性を知らないで、しかも探索に失敗した場合、失敗の原因が進化計算にあるのか表現形式にあるのかを知ることはできない。たとえば、$\{ww|w \in \{0,1\}*\}$ という言語を判定するプログラムを生成したいとしよう。もしレパートリがプッシュダウン・オートマトンのそれと同じ表現を用いたならば、探索は決して成功しない。プッシュダウン・オートマトンではこの言語を判定できないからである。

考えられる解決法として、理想的には無限の番地付きメモリとそれにアクセスするための非終端記号を導入するというものがある。そのような非終端記号からなる構文木で解の候補を表現し、特定のメモリの値が停止条件を満たすまで構文木の評価を繰り返すならば、表現力はチューリング・マシンと同等つまりチューリング完全になることが示されている。

しかしながら、チューリング完全性以外にも GP のための表現形式が満たすべき条件はある。それは次のようなものである。

- 入出力方法を利用者が指定できる
- 有用な部品を導入しやすい
- 機能が分離している
- 階層化している

これらの必要性は本文で詳しく議論する。

本論文ではこれらの条件を満たすような表現形式、RTN を提案する。RTN は標準的な GP の自然な拡張である。標

準的な GP は解の表現として単一の構文木を用いる。一方、RTN は回帰的なネットワークである。ネットワークは複数のノードで構成されるが、個々のノードは値と構文木で表現される純関数を持つ。構文木を構成するのに特別な非終端記号は必要ない。終端記号は 4 つの変数と定数のいずれかである。標準的な GP においては、プログラムへの入力は変数に代入されるが、RTN においてはノードの値に代入される。

RTN の例を示す。2 つのノードからなる RTN である。各ノードが持つ関数は、

$$\#1 : (c - P[c])/2,$$
$$\#2 : P[a]d,$$

である。ここで P は 2 で割った余りを返すような手続きである。関数は最大で 4 つの引数 a, b, c, d を持つ。これらの引数にはノードの値が代入される。代入方法は、#1 については $\{*, *, 1, *\}$、#2 については $\{1, *, *, 2\}$ のように表現される。#1 の 3 番目の要素と #2 の 1 番目の要素はともに 1 であるから、#1 の第 3 変数つまり c と #2 の第 1 変数つまり a には #1 の値が代入される。#2 の 4 番目の要素は 2 であるから、#2 の 4 番目の変数つまり d には #2 の値が代入される。

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。ノード $\#n$ が持つ関数を f_n、時刻 t での値を $v[[n, t]]$ とする。関数 f_n が持つ引数の数を k_n とし、i 番目の引数には$\#l_{n,i}$ の値が代入されるものとする。時刻 $t + 1$ における$\#n$ の値は、

$$v[[n, t + 1]] = f_n[v[[l_{n,1}, t]], \cdots, v[[l_{n,k_n}, t]]]$$

となる。プログラムへの入力は #1 の値に代入され、#2 の値は $t = 0$ において 1 とする。例として、プログラムへの入力が 2 進数 1011 だったとする。RTN の遷移は次のようになる。

#1 の値	1011	101	10	1	0
#2 の値	1	1	1	0	0

#1 の値が 0 になったとき、プログラムへの入力が 0 を含んでいた場合に限って #2 の値が 0 になる。

RTN が任意のチューリング・マシンをシミュレートできること、つまり RTN が任意のアルゴリズムを表現できることは簡単に示すことができる。証明は本文中で与える。

応用例として、RTN を用いた GP による言語判定装置の自動生成を試みる。これは従来の GP が失敗していた課題である。RTN を用いた GP はこの課題に成功する。比較実験として、番地付きメモリを用いた GP でもこの課題を試みるが、これは成功しない。このことから、一般的な自動プログラミングにおける RTN の有効性が期待される。

GP のための表現形式はほかにもさまざまなものが提案されている。それらと RTN の違いについても議論する。また、違いを議論する際に用いる基準についても考察する。たとえば、No Free Lunch Theorem は重要ではないことがわかる。