파이썬 언어의 바이트 코드수준 진화를 이용한 유전 프로그래밍

박현수○ 김경중
세종대학교 컴퓨터공학과
rex8312@gmail.com, kimkj@sejong.ac.kr

Genetic Programming using Bytecode-Level Evolution for Python Programming Language

Hyunsoo Park○ Kyungjoong Kim
Dept. of Computer Engineering, Sejong University

요 약

Genetic Programming은 자동으로 프로그램을 작성하는 기법으로 많은 가능성을 지니고 있다. 하지만, 아직까지 고차원 프로그래밍 언어 수준에서 진화는 이루어지지 못하고 있다. 하지만, 최근 들어, 한단계 아래인 JAVA Baytecode를 이용한 진화가 몇몇 문제를 해결할 수 있는 가능성을 보여주고 있다. 본 연구에서는 최근 관심을 많이 받고 있는 프로그래밍 언어중의 하나인 Python의 Bytecode를 자동으로 진화하는 방법을 제안한다. 본 연구를 통해 JAVA와는 또 다른 형태의 언어인 Python언어에서의 Bytecode수준의 진화 가능성을 살펴본다. 두 가지 형태의 회귀문제를 통해 성능을 평가해 보았으며, 자동으로 Python Bytecode를 작성할 수 있음을 보였다.

1. 서 론

진화의 원리를 이용해서 최적화 문제를 풀어내는 방법에는 ES(Evolutionary Strategy), GA(Genetic Algorithm), GP(Genetic Programming) 등의 방법이 있다. 이 중에 Genetic Programming은 진화적인 방법론을 이용하여 주어진 문제를 해결할 수 있는 프로그램을 자동으로 작성한다. 본 논문에서는 전통적인 트리 구조를 사용하는 GP와 달리 실제 프로그래밍 언어의 중간단계 언어인 Bytecode를 이용한 GP를 제안하고 그 가능성을 보인다.

전통적인 트리 구조를 사용하는 GP와 달리 최근에는 단순한 형태의 구조를 사용해서 프로그램을 표현하려고 한다. 그런 과정에서 Linear GP, Stack기반 GP 같은 것이 연구 되었다[2][8]. 트리 구조가 아닌 GP를 사용하여 탐색 범위를 줄이는 효과를 볼 수 있었지만, Stack 기반 GP에서는 Stack underflow와 같은 문제도 발생했다[2].

이 논문은 2010년도 정부(교육과학기술부)의 재원으로 한국연구재단의 지원을 받아 수행된 기초연구사업(2010-0012878) 및 뇌과학원천기술개발사업임(2010-0018948)

최근에는 Stack 기반 GP를 실용적인 측면에서 접근하여 Java의 Bytecode를 기반으로 하는 연구가 있다[3][4]. 이 연구에서는 특별히 GP를 위해 설계한 명령어 집합을 사용하지 않고 실제 Java의 Bytecode를 사용함으로써 기존의 연구의 한계를 극복하려 했다. 특히, Java 가상머신(JVM)을 이용하면 실행이 불가능한 개체들의 예외를 자동으로 감지할 수 있다. 보통 GP를 이용하여 생성한 프로그램은 실행이나 컴파일이 불가능한 경우가 많지만, JVM을 이용하여 배제하는 것이 가능하다.

Python은 Java와 비슷하게 Bytecode로 변환하여 Interpreter에서 실행하는 언어이다. Python또한 Stack기반으로 연산을 수행하고 있으며, Java 가상머신과 마찬가지로 몇몇 예외를 찾아주고 있다. 최근, 과학기술관련 프로그래밍에 널리 사용되고 있어 사용자가 늘어나고 있는 추세이다.

이 논문에서는 Python의 Bytecode를 이용하여 GP를 수행하는 과정을 보여주고, 실험에서 회귀 문제를 풀어 보임으로써 이런 방법이 간단한 문제에 적용 가능함을 보인다.

2. Python Bytecode기반 Genetic Programming

2.1. Python Bytecode

Python의 Bytecode는 Stack에 기반해서 연산을 수행한다. 예를 들어 표 1 왼쪽과 같은 함수는 표 1 오른쪽과 같은 Bytecode로 변환되고 수행된다. 이 함수는 2*(x+x)를 수행하는 함수이다.

Bytecode의 명령어를 간단히 설명하면, LOAD_FAST x 명령은 Stack에 변수 x를 push 하라는 뜻이고, STORE_FAST x는 Stack에서 pop한 수를 변수 x에 저장하라는 뜻이다. BINARY_ADD, BINARY_MULTIPLY 등의 연산은 Stack에서 숫자 둘을 pop하여 해당 연산을 하고 그 결과를 push 한다. 마지막으로 RETURN_VALUE 명령은 Stack에서 숫자 하나를 pop하여 함수의 결과로 반환한다[5].

표 1의 오른쪽 bytecode의 연산 과정을 따라가면, 왼쪽의 함수와 똑같이 동작하고 있음을 알 수 있다. GP에서는 이러한 명령어를 저장한 리스트를 유전자형 개체로 사용한다. 본 논문에서는 Bytecode를 처리하기 위해 Python의 모듈인 Byteplay[6]를 사용했다.

표 1. Python Bytecode의 예

Python Script	Python Bytecode	
def f(x):	LOAD_FAST	x
x = x + x	LOAD_FAST	x
x = 2 * x	BINARY_ADD	
return x	STORE_FAST	x
	LOAD_CONST	2
	LOAD_FAST	x
	BINARY_MULTIPLY	
	STORE_FAST	x
	LOAD_FAST	x
	RETURN_VALUE	

2.2 Python 구현체

흔히 사용하는 Python의 구현체는 CPython이다. 이것은 C로 구현된 Python의 컴파일러와 인터프리터, 필수 모듈 등으로 구성되어 있다. 이 외에도 Java로 구현된 Jython, C#으로 구현된 IronPython 등이 있다[7].

본 논문에서도 초기에는 CPython을 이용하여 GP를 수행하려고 했으나, 필수적인 예외를 잡아주지 못하는 문제가 발생했다. 반면 Python으로 구현된 Python 구현체인 PyPy는 필요한 예외를 감지해 냈기 때문에 본 논문에서는 PyPy 1.6를 사용했다.

CPython과 PyPy 두 구현체는 Python의 Spec이 정의하는 안에서는 똑같이 동작해야 하지만, 진화를 통해 구현된 코드를 이용해서 테스트 했을 때, 일부 코드에서 차이를 보였다. 이런 경우 PyPy를 기준으로 했다.

3. Genetic Programming

본 논문에서는 돌연변이 연산은 사용하지 않고, 교배와 선택만을 사용했다. 알고리즘의 개요는 그림 1과 같다.

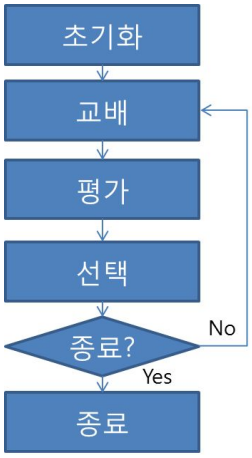

그림 1. 알고리즘 개요

3.1. 초기화

초기화 과정에서는 최초에 부모가 될 Python 프로그램들을 생성해야 한다. 일반적인 GA, GP에서는 무작위로 부모 세대를 생성하지만, Java나 Python의 Bytecode처럼 실제 사용하는 언어를 사용할 때 무작위로 생성해서 초기화를 한다면, 제대로 된 형식을 갖추지 못해 실행조차 되지 않는 프로그램들로 초기집단이 채워지는 문제가 있다.

따라서 본 논문에서는 표 2를 컴파일한 Bytecode를 초기집단으로 복사해서 사용한다. 초기 집단으로 사용하는 Bytecode에는 최적해에 필요한 모든 연산이 있을 필요가 있으며, 이것을 제한하면, 최적해가 아닌 근사해가 찾아진다.

표 2. 초기 집단으로 사용하는 함수

```
def func(x):
    x = cos(sin(log((((x+x)-x)*x)/x)))
    return x
```

3.2. 교배

본 논문의 GP는 두점 교배를 하는 GA의 교배 연산과 유사하다. 두 부모(A, B)를 임의로 정해서 그 중 한 부모(A)의 명령어 리스트의 일부분을 임의로 선정하여 다른 부모(B)의 명령어 리스트 임의의 지점에 복사한 형태의 자손을 생성한다.

만약 부모(A)에게서 선택한 리스트가 비어 있다면, 부모(B)의 선택한 부분을 지우는 것이며, 부모(B)에게서 선택한 부분이 비어 있다면, 단순히 부모(A)에서 선택한 부분을 부모(B)에 복사하는 것이다.

부모(A)에서 복사하기 위해 선택한 리스트의 길이나, 부모(B)에게 복사가 이루어질 영역의 길이는 정규분포 N(0,3)에서 얻은 난수에 절대값을 취하여 사용한다.

이 때 생성한 자손은 실행이 불가능한 자손이 될 수 있다. 만약 Bytecode를 함수에 적재하는 과정에서 예외가 발생하는 경우 바로 해당 자손을 버리고 다시 교배를 한다. 이런 방식으로 현재 부모의 숫자만큼 자손을 생산할 때까지 반복한다.

3.3. 평가 및 선택

적합도는 해당 개체가 가지고 있는 Bytecode를 이용해서 프로그램을 수행했을 때 찾으려는 목표함수와 차이가 얼마나 나는가에 따라 정해진다. [0, 1) 사이의 숫자를 임의로 20개 골라서 목표함수와 평가하려는 Bytecode로 만들어진 프로그램에 입력으로 주었을 때 결과 차이의 절대 값을 모두 더한 수치를 error라고 한다.

$$fitness = \frac{1}{(error + 1)}$$

따라서 최대 적합도는 1.0이 되며, error가 클수록 0에 가까워진다.

부모개체 수가 N이라고 한다면, 현재 세대의 자손 N개를 포함하여 총2N개의 개체가 있다. 선택 과정에서는 이 중에서 가장 적합도가 높은 N개의 개체만을 남기고 나머지 개체를 제거한다.

4. 실험

본 논문에서는 회귀 문제 두 가지를 사용해서 실험을 진행했다.

4.1. 회귀 문제 I

표 3. 간단한 회귀 실험의 조건

문제	$y = x^4 + x^3 + x^2 + x$
집단 크기	1000
세대 수	50
초기 집단	표 2의 Bytecode를 복사해서 사용
교배 비율	1.0
실험 횟수	100번

표 4. 실험 결과

30 세대에서 수율	62 %
50 세대에서 수율	94 %

총 100번의 실험 했을 때 30 세대에서는 62번 해를 찾아냈으며, 50세대에서는 94번 해를 찾아냈다.

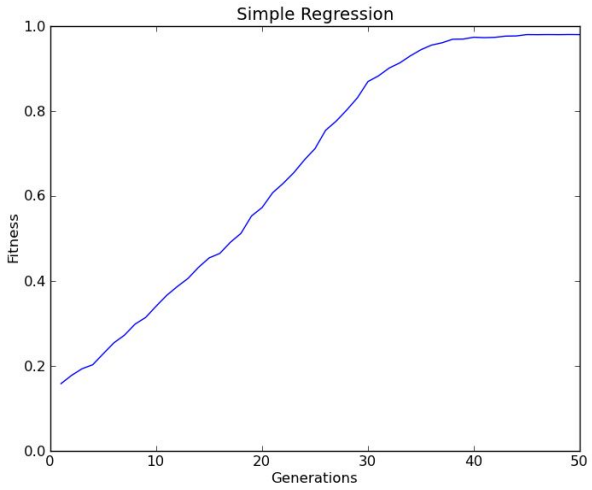

그림 2. 간단한 회귀 실험에서 최대 적합도의 평균

그림 2는 100번 실험을 수행하는 동안 각 세대의 최대 적합도의 평균이다. 표 5는 진화를 통해 구해낸 해답 중의 하나이다.

표 5. 간단한 회귀 문제 해답의 예

```
LOAD_GLOBAL          'cos'
LOAD_FAST            'x'
LOAD_FAST            'x'
BINARY_MULTIPLY       None
LOAD_FAST            'x'
BINARY_ADD            None
LOAD_FAST            'x'
BINARY_MULTIPLY       None
```

LOAD_FAST	'x'
BINARY_ADD	None
LOAD_FAST	'x'
BINARY_MULTIPLY	None
LOAD_FAST	'x'
BINARY_ADD	None
RETURN_VALUE	None

4.2. 회귀 문제 Ⅱ

표 6 복잡한 회귀 실험의 조건

문제	$y = \sum_{i=1}^{9} x^i$
집단 크기	1000
세대 수	300
초기 집단	표 2의 Bytecode를 복사해서 사용
교배 비율	1.0
실험 횟수	100

표 7. 복잡한 회귀 실험의 결과

150 세대에서 수율	23%
300 세대에서 수율	31%

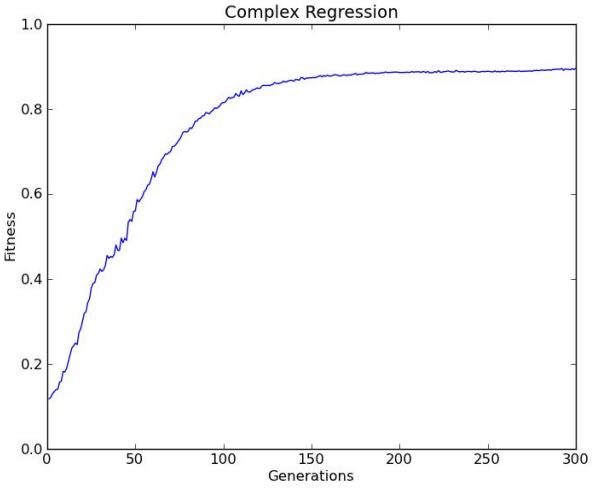

그림 3. 복잡한 회귀 실험에서 최대 적합도의 평균

그림 3은 100번 실험을 수행하는 동안 각 세대의 최대 적합도의 평균이다.

5. 결론 및 향후연구

Stack 기반으로 연산을 수행하는 Python의 Bytecode를 이용한 GP로도 기존에 Java의 Bytecode를 이용한 연구[2][3]와 비슷한 결과를 얻을 수 있었다.

하지만, Java Bytecode를 이용한 기존 연구에서는 진화를 통해 얻어낸 코드를 디컴파일러를 통해 소스코드 형태로 복원하여 결과를 분석하는데 사용했지만, 이 연구에서는 적합한 Python의 디컴파일러를 찾지 못했다. 아마도, Python이 Java보다 사용자가 적고, 스크립트 형태로 실행되는 특성상 디컴파일러가 많지 않은 것으로 판단된다. 따라서 진화를 통해 얻어낸 코드를 분석할 효과적인 방법이 필요하다고 생각한다.

또한, 진화를 통해 얻어낸 코드가 Python의 구현체에 따라서 다르게 작동하는 경우가 있었다. 본 논문에서는 진화한 코드를 테스트를 하는 과정에서 PyPy를 사용했다. 하지만 실험 초기에는 테스트를 하는 과정에서는 CPython 2.7을 사용하기도 했는데, 가끔이지만, 두 구현체의 결과가 다르게 출력되기도 했다. 각 구현체의 구현상의 차이점으로 인해 발생하는 것으로 보인다.

결과를 요약하자면, Python Bytecode를 이용한 GP는 Java Bytecode를 이용한 GP와 비슷한 특징을 가진다. Stack 기반의 GP이기 때문에 명령어 리스트 형태의 단순한 데이터 형을 사용하며, 실제 언어의 Bytecode를 사용하기 때문에 제약이 없이 실제 프로그램과 같이 사용될 수 있다. 반면, 유용한 디컴파일러를 구하기 힘든 점과, Python 구현체 간의 구현상의 차이로 인한 결과의 차이 등이 문제가 될 수 있다.

6. 참고 문헌

[1] J. R. Koza, *Genetic Programming: On the Programming of Computers by Means of Nature Selection*. MIT Press, December, 1992.

[2] T. Perkis, "Stack-based Genetic Programming," *In Proc. 1st Int. Conf. on Evol. Comp.*, vol. 1, pp. 148-153, June,1994.

[3] M. Orlov and M. Sipper, "Genetic Programming in the Wild: Evolving Unrestricted Bytecode," *Genetic Evolutionary Computation Conference*, July, 2009.

[4] M. Orlov and M. Sipper, "Flight of the FINCH through the Java Wilderness," *IEEE Transactions on Evolutionary Computation*, vol 15, no 2, April, 2011.

[5] http://docs.python.org/library/dis.html

[6] http://pypi.python.org/pypi/byteplay/0.2

[7] http://www.python.org

[8] R. Poli, W. B. Langdon, and N. F. McPhee, *A Field Guide to Genetic Programming*, Lulu Enterprises, March, 2008.