제23회 한국정보처리학회 춘계학술발표대회 논문집 제12권 제1호 (2005. 5)

Python 을 사용한 유전 알고리즘 구현

이원재, 김학영
한국전자통신연구원
e-mail : {russell, h0kim}@etri.re.kr

Genetic Algorithm Implementation in Python

Wonjae Lee, Hak-Young Kim
Electronics and Telecommunications Research Institute (ETRI)

요 약

본 논문에서는 Python 을 사용한 유전 알고리즘 구현을 다룬다. 유전 알고리즘은 생물의 진화과정에서 일어나는 자연선택과 같은 유전법칙을 모방한 확률적 탐색기법이다. 유전 알고리즘에서는 염색체를 하나의 리스트 혹은 문자열로써 다룬다. 리스트나 문자열 처리 위주인 유전 알고리즘의 경우, 기존의 C/C++/Java 보다 표현력이 풍부한 Python 으로 프로그래밍할 경우 별도의 라이브러리 없이 쉽게 구현이 가능하다. 본 논문에서는 Python 을 사용한 유전 알고리즘 구현 방법에 대해 소개하고, 추가적으로 높은 성능을 얻기 위한 방법들에 대해 논의한다.

1. 서론

유전 알고리즘(genetic algorithm)은 생물의 진화과정인 자연선택(natural selection)과 유전법칙을 모방한 확률적 탐색기법이다. [1] 유전 알고리즘은 문제의 잠재해를 표현한 개체들로 이루어진 모집단을 가지고 시작한다. 모집단은 매 세대마다 일정수의 개체를 유지한다. 매 세대에서 각 개체의 적응도(fitness)를 평가하여 이에 따라 다음 세대에 생존할 개체들을 확률적으로 선별(selection)한다. 선별된 개체들 중 일부의 개체들이 임의로 짝을 지어 교배하여 자손을 생성한다. 이때 교차(crossover)에 의해 부모의 유전자가 자손에게 상속되고, 돌연변이(mutation)가 일어날 수 있다. 자손은 부모로부터 좋은 유전형질을 상속받는다고 가정할 때, 다음 세대의 잠재해들은 평균적으로 전 세대보다 더 좋아진다고 볼 수 있다. 이러한 진화과정은 종료조건을 만족할 때까지 반복한다.

유전 알고리즘에서 문제의 잠재해를 표현하는 개체는 염색체(chromosome) 또는 스트링(string)이라 불린다. 그리고 각 개체를 이루는 단위는 유전자라 부른다. 대부분의 경우 프로그램에서 개체는 리스트(배열)나 문자열로 표현된다. 따라서 유전 알고리즘에서 일어나는 여러 가지 연산들은 리스트나 문자열에 대해 일어난다. 이러한 종류의 연산에는 기존 C/C++/Java 와 같은 언어보다는 Python[2], Perl[3]과 같은 동적 언어가 생산성이 높다. 이 때문에 Bioinformatics 에서는 Python[4] 이나 Perl[5]등이 널리 활용되고 있다.

본 논문에서는 가독성과 표현력이 뛰어난 Python version 2.4 를 사용한 유전 알고리즘을 구현 방법에 대해 소개한다. 그리고 Python 에서 높은 성능을 얻기 위한 방법들에 대해 논의한다.

2. Python 개요

Python 은 다양한 프로그래밍 패러다임을 지원하는 동적 타입(dynamic typing) 언어이다. 절차형 프로그래밍(procedural programming), 함수형 프로그래밍(functional programming), 객체지향 프로그래밍이 모두 가능하다. 따라서 작업의 특성에 맞는 패러다임을 사용하여 코드를 쉽게 작성할 수 있다. 유전 알고리즘의 경우 리스트에 대한 처리가 많기 때문에 함수형 프로그래밍 스타일을 많이 사용하게 된다.

Python 은 Perl 과 자주 비교되는 데, Python 은 난해한 Perl 비해서 문법이 단순하여 가독성 및 관리성이 뛰어나다. 때문에 많은 프로그래머들이 Perl 대신 Python 을 선택하고 있다. Python 은 매우 쉽기 때문에 입문용 프로그래밍 언어로도 적합하다. Python 프로그램은 컴파일, 링킹, Makefile 등이 필요 없기 때문에 개발과정이 단순하고 빠르다. Python 은 가변 리스트, associative array(Python 에서는 dictionary 라고 불린다)와

같은 높은 수준의 자료 구조들을 기본적으로 지원하기 때문에 생산성이 대단히 높다. 또한 Python 의 기본 라이브러리는 매우 방대하여 3rd Party 라이브러리가 필요한 경우는 드물다.

Python 에서는 indentation 이 블록 구조를 결정한다. 따라서 C/C++/Java 에서와 달리 중괄호({,})가 쓰이지 않고, 결과적으로 코드가 매우 깔끔하며 프로그래머로 하여금 가독성이 좋은 코드를 생산하도록 유도한다. 이러한 Python 의 특성과 단순하고 직관적인 문법덕분에 Python 코드는 종종 '실행 가능한 유사 코드(executable pseudo-code)'라고 불린다.

Python 프로그램의 디버깅은 매우 쉽다. 오류가 발생했을 때는 Java 와 유사한 trace-back 메시지에 기반해서 디버깅할 수 있다. 그리고 기본 라이브러리에 unit test 모듈이 제공된다.

필요한 경우, 개발과정에서 PyChecker[6]를 이용해서 프로그램 실행 전에 문법상 오류를 찾아낼 수 있다. PyChecker 는 C/C++/Java 같은 언어들의 경우 컴파일 과정에서 발견되는 문법상의 오류들을 찾아준다. Lint 와 유사한 역할을 한다고 볼 수 있다.

Python 은 동적 언어(dynamic language) 특성상 매우 유연(flexible)하기 때문에 C/C++/Java 등에 비해 생산성이 매우 높다. 따라서 여러 가지 다양한 방법들을 시험해 보는데 적합하다.

그리고 Python 으로 작성된 프로토타입 코드는 프로토타입핑에 그치지 않고, 뒤에 소개되는 최적화 기법을 사용하면 높은 성능을 얻을 수 있다.

3. 유전 알고리즘 구현

본 논문에서는 도시 개수가 9 개인 외판원 문제 위주로 유전 알고리즘 구현에 대해 설명한다. 각 개체는 방문하는 도시를 순서대로 나열한 리스트이다. [1]

초기 모집단 생성

초기 모집단을 생성하기 위해서는 우선 가능해(feasible solution)를 하나 만든 후, 이를 무작위적으로 변형시키면 된다. 여기서는 5 개의 개체를 생성해 본다.

```
import random, copy

sample_chrm = range(1,10)   # a feasible solution
init_population = [ ]       # an empty list
random.seed(0)
population_size = 5

for i in xrange( population_size ):
    new_chrm = copy.copy( sample_chrm )
    random.shuffle( new_chrm)
    init_population.append( new_chrm)
```

range(1,10)는 1 부터 9 까지의 숫자를 element 로 가지는 list 를 생성한다. random.seed(0)는 난수 발생기를 초기화 시키는 것이다. 항상 같은 순열을 생성할 필요가 없다면 난수 발생기를 특정 수로 초기화 시킬 필요가 없다. random 모듈이 import 될 때 OS 가 제공하는 randomness source 가 사용되기 때문이다.

for 문에서는 population_size 만큼의 개체를 생성한다. copy.copy 는 shallow copy 를 수행한다. random 모듈의 shuffle 함수는 리스트의 원소들을 무작위적으로 섞는다.

적응도 평가

적응도는 자연개체의 생존능력을 나타낸다. 최적화 문제에서 적응도는 목적 함수에 의해 측정된다. [1]

외판원 문제에서 도시 간 이동 시 발생하는 비용을 나타내는 cost matrix 는 다음과 같은 방법으로 생성할 수 있다.

```
cost_matrix = []
cost_matrix.append([0,0,0,0,0,0,0,0,0,0])
cost_matrix.append([0,0,1,5,6,9,2,3,7,8])
cost_matrix.append([0,1,0,8,6,2,4,7,9,5])
cost_matrix.append([0,5,8,0,3,2,7,6,8,9])
cost_matrix.append([0,6,6,3,0,9,7,4,1,5])
cost_matrix.append([0,9,2,2,9,0,1,4,7,3])
cost_matrix.append([0,2,4,7,7,1,0,7,4,1])
cost_matrix.append([0,3,7,6,4,4,7,0,8,3])
cost_matrix.append([0,7,9,8,1,7,4,8,0,1])
cost_matrix.append([0,8,5,9,5,3,1,3,1,0])
```

주어진 개체 chrm(chromosome)에 대해 다음과 같은 방법으로 총 비용을 계산할 수 있다.

```
chrm = [4, 1, 5, 6, 9, 2, 3, 7, 8]
cost = 0
last_city = chrm[0]
for current_city in chrm:
    cost += cost_matrix[last_city][current_city]
    last_city = current_city
```

비용이 크다는 것은 적응도가 떨어진다는 이야기이다. 따라서 뒤에서 설명할 확률바퀴 방법 같은 경우 비용 값을 그대로 사용할 수 없으며, 여러 가지 방법을 사용하여 적절한 적응도를 계산해야 한다. [1]

선별

선별은 환경에 대한 적응도에 의해 현 세대의 모집단으로부터 다음 세대에 생존할 개체를 선택하는 과정이다. [1]

여기서는 확률바퀴 방법을 사용하여 선별을 한다. 개체들의 적응도가 다음 fitness_list 와 같을 때, 다음과 같은 방법으로 누적 확률 리스트(cum_prob_list)를 구할 수 있다. 마지막의 cum_prob_list[-1] = 1.0 문장은 roundoff error 를 보정하기 위한 것이다. Python 리스트에서 음수로 인덱스 값을 주면 리스트 마지막에서부터 숫자를 세어 해당하는 원소를 가져온다. 따라서 cum_prob_list[-1]은 마지막에 있는 원소를 가리킨다.

```
import operator

fitness_list = [6.0, 9.0, 4.0, 3.0, 5.0, 8.0, 3.0, 6.0, 3.0, 3.0]

fitness_sum = reduce( operator.add, fitness_list)

prob_list = map( (lambda x: x/fitness_sum), fitness_list)

cum_value = 0
cum_prob_list = [ ]
for prob in prob_list:
    cum_prob_list.append( cum_value + prob )
    cum_value += prob

cum_prob_list[-1] = 1.0
```

위와 같이 누적 확률 리스트 cum_prob_list 가 주어졌을 때, 다음과 같은 선형 탐색(sequential search) 방법으로 원하는 수만큼의 개체를 선택할 수 있다. 개체수가 많을 경우, 이진 탐색(binary search)을 사용하여 더 빠르게 동작하도록 할 수도 있다.

```
import random

selected = []
size = 100

for i in xrange(size):
    rn = random.random()
    for j, cum_prob in enumerate(cum_prob_list):
        if rn<= cum_prob:
            selected.append( j)
            break
```

교차

교차는 두 부모가 갖는 유전자를 조합하여 자손을 생산하는 과정이다. 교차는 좋은 해를 이용하는 역할을 한다. [1] Python list 의 slice operation 을 사용하면 일점 및 이점 교차를 간단히 구현할 수 있다. Binary list 에 대한 일점교차는 다음과 같이 구현할 수 있다.

```
parent1 = [ 1, 0, 1, 1, 0, 1, 1, 1 ]
parent2 = [ 0, 1, 0, 0, 1, 0, 1, 1 ]

pt = 3      # crossover point

offspring1 = parent1[:pt] + parent2[pt:]
offspring2 = parent2[:pt] + parent1[pt:]
```

여기서는 3 에서 교차가 일어났다. 이때 offspring1, offspring2 는 다음과 같다. 이탤릭 체로 된 부분은 parent1 에서 상속받은 부분을 나타낸다.

```
[1, 0, 1, 0, 1, 0, 1, 1]
[0, 1, 0, 1, 0, 1, 1, 1]
```

이점 교차 같은 경우 다음과 같이 할 수 있다.

```
pt1 = 2      # crossover point 1
pt2 = 5      # crossover point 2

offspring1 = parent1[:pt1] + parent2[pt1:pt2] + parent1[pt2:]
offspring2 = parent2[:pt1] + parent1[pt1:pt2] + parent2[pt2:]
```

이때 offspring1, offspring2 는 다음과 같다. 이탤릭 체로 된 부분은 parent1 에서 상속받은 부분을 나타낸다.

```
[1, 0, 0, 0, 1, 1, 1, 1]
[0, 1, 1, 1, 0, 0, 1, 1]
```

외판원 문제와 같은 경우 비가능해(infeasible solution)를 피하기 위해 순서교차[1]를 해야 한다. 다음은 순서교차 예이다.

```
parent1 = [4, 1, 5, 6, 9, 2, 3, 7, 8]
parent2 = [3, 1, 8, 6, 2, 4, 7, 9, 5]

pt1 = 2      # crossover point 1
pt2 = 5      # crossover point 2

latter_length = len(parent1) - pt2

prt1_mid = parent1[pt1:pt2]
prt2_mid = parent2[pt1:pt2]

prt1_reordered = parent1[pt2:] + parent1[:pt2]
prt2_reordered = parent2[pt2:] + parent2[:pt2]

prt1_reord_filtered = filter( lambda x: x not in prt2_mid, prt1_reordered )
prt2_reord_filtered = filter( lambda x: x not in prt1_mid, prt2_reordered )

offspring1 = prt2_reord_filtered[-pt1:] + prt1_mid + prt2_reord_filtered[:latter_length]
offspring2 = prt1_reord_filtered[-pt1:] + prt2_mid + prt1_reord_filtered[:latter_length]
```

이때 offspring1, offspring2 는 다음과 같다.

```
[8, 2, 5, 6, 9, 4, 7, 3, 1]
[5, 9, 8, 6, 2, 3, 7, 4, 1]
```

돌연변이

유전 알고리즘에서 돌연변이는 해공간을 다양하게 탐색하는 역할을 한다. [1] 돌연변이 방법으로는 삽입과 교환 등이 있다. 삽입 같은 경우 다음과 같은 방법으로 구현할 수 있다.

```
import random

chrm = [4, 1, 5, 6, 9, 2, 3, 7, 8]

element_position = random.randint(0, len(chrm)-1 )
```

```
insert_position = random.randint(0, len(chrm)-2 )

element_value = chrm[element_position]
del chrm[element_position]
chrm.insert( insert_position, element_value )
```

위의 예에서 element_position 과 insert_position 이 같으면 안되지만, 간결성을 위해 이러한 경우를 무시하였다.

교환 같은 경우 다음과 같은 방법으로 구현할 수 있다.

```
import random

chrm = [4, 1, 5, 6, 9, 2, 3, 7, 8]

position1 = random.randint(0, len(chrm)-1 )
position2 = random.randint(0, len(chrm)-1 )

chrm[position1], chrm[position2] = chrm[position2], chrm[position1]
```

위의 코드의 경우도 실제로는 position1 과 position2 가 동일하지 않도록 체크하는 루틴이 추가되어야 할 것이다.

4. 성능을 높이기 위한 방법들

Python 은 인터프리터 방식이기 때문에, 계산 중심 프로그램의 경우 수행속도가 C/C++에 비해 많이 느리다. 그래서 numerical computing 을 위해 Numerical Python [7], Numarray [8] 등과 같은 모듈이 나와 있다. 이러한 모듈들은 많은 부분이 C 로 작성되어 있기 때문에 C 에 근접한 성능을 얻을 수 있다.

또 다른 방법으로는 빠른 속도가 필요한 부분을 C 로 작성한 후 wrapper 를 통해 Python 에서 사용하는 것이다. SWIG[9], SIP [10] 등은 wrapper 를 자동으로 생성해 준다.

또 다른 방법으로는 Pyrex [11]를 사용하는 것이다. Pyrex 는 Python 과 유사한 문법을 가지는 코드를 작성하면 C code 를 생성한다. Pyrex 는 Python 과 C 사이의 격차를 좁혀주는 중간형태의 언어라고 생각할 수 있는데, 높은 생산성과 빠른 속도 둘을 모두 충족시킬 수 있는 훌륭한 대안이라고 생각된다.

IA32 기반에서 성능을 높이기 위한 가장 쉬운 방법은 Psyco[12][13]를 사용하는 것이다. Psyco 는 일종의 just-in-time 컴파일러로서, 주어진 Python 프로그램에 대해 기계어를 생성해 낸다. 최대 100 배까지 높은 성능을 얻을 수 있고, 보통 4 배 정도 빨라진다고 한다.

5. 결론

유전 알고리즘(genetic algorithm)은 생물의 진화과정인 자연선별(natural selection)과 유전법칙을 모방한 확률적 탐색기법이다. Python 은 절차형 프로그래밍(procedural programming), 함수형 프로그래밍(functional programming), 객체지향 프로그래밍 등을 모두 지원하는 동적 타입 언어이다. Python 은 C/C++/Java 보다 표현력이 풍부하여, 유전 알고리즘의 경우 별도의 라이브러리 없이도 빠른 시간 안에 구현이 가능하다. 특히, functional programming tool 들을 유효 적절히 사용하면 매우 높은 생산성을 얻을 수 있다. 순수 Python 으로만 작성된 유전자 알고리즘 프로그램은 C/C++로 구현된 것에 비해 느리다. 하지만, Numerical Python, Numarray, SWIG, SIP, Pyrex, Psyco 등을 사용하면 C 에 근접하는 성능을 얻을 수 있다.

참고문헌

[1] 김여근, 윤복식, 이상복, “메타 휴리스틱”
[2] Python Programming Language, http://www.python.org
[3] Perl Programming Language, http://www.perl.org/
[4] Biopython Project, http://biopython.org/
[5] Bioperl project, http://bioperl.org/
[6] PyChecker, http://pychecker.sourceforge.net/
[7] Numerical Python, http://numeric.scipy.org/
[8]Numarray, http://www.stsci.edu/resources/software_hardware/numarray
[9] SWIG, http://www.swig.org/
[10] SIP, http://www.riverbankcomputing.co.uk/sip/
[11]Pyrex, http://www.cosc.canterbury.ac.nz/~greg/python/Pyrex/
[12] Psyco, http://psyco.sourceforge.net/
[13] David Mertz, “Make Python run as fast as C with Psyco”, http://www-106.ibm.com/developerworks/linux/library/l-psyco.html?t=gr,lnxw03=PsycoC