BioInspired Computing

Natural Computing Homework

Stephanie Athow Paul Blasi

March 16, 2015

Contents

List of Figures

List of Algorithms

Document Preparation and Updates

Current Version [0.0.1]

Prepared By:
Stephanie Athow
Paul Blasi

Revision History

Date	***Author***	***Version***	***Comments***
2/2/15	*Team Member #1*	*1.0.0*	*Initial version*
3/4/15	*Team Member #3*	*1.1.0*	*Edited version*

1

Evolutionary Algorithms - Text Chapter 3

1.1 Problem 1

Implement the various hill-climbing procedures and the simulated annealing algorithm to solve the problem exemplified in Equation 1.1. Use a real-valued representation scheme for the candidate solutions (variable x).

By comparing the performance of the algorithms, what can you conclude?

For the simple hill-climbing try different initial configurations as attempts at finding the global optimum. Was this algorithm successful?

Discuss the sensitivity of all the algorithms in relation to their input parameters.

$$g(x) = 2^{-2((x-0.1)/0.9)^2} * \sin(5\pi x)^6 \tag{1.1}$$

Simple Hill Climb:

In the simple hill climb algorithm, a point is randomly selected in the search space, evaluated, slightly perturbed and re-evaluated until either a max iteration value or non-signifacant change happens. This algorithm easily finds local maximums or minimums but can miss the global extrema since there is no way to 'escape' a local extrema.

Iterated Hill Climb:

The iterated hill climb algorithm repeats the simple hill climb for a specified number of loops, with random starting points each time, and keeps track of the best solution. Since this simply reruns simple hill climb over and over with different starting points, local extremas can be 'escaped' or 'ignored' and the global extrema found.

Stochastic Hill Climb:

The stochastic hill climb is quite similar to the simple hill climb, with one change that makes a significant difference. When the point is perturbed, it is perturbed randomly and its acceptance as a new point is probabilistically determined with Equation 1.2 The value of T plays an important part in this equation. T determines the decay of the exponential function. In plain words, T determines how important the relative difference between the evaluation of x and x' is. A large T will make the search very similar to a random search and does not provide consistent results. When T is small, local extremas can be escaped and global extremas found with fairly consistent results.

$$P = 1/(1 + exp[(eval(x) - eval(x'))/T]) \tag{1.2}$$

A few sample runs of the various hill climb algorthims can be found in Figure 1.1.

```
C:\Anaconda>ipython nc_hw2_hill_climb.py
Simple Hill Climb
X =  0.897613925322

Iterated Hill Climb
X =  0.099692725589

Stochastic Hill Climb
X =  0.0996053805818

C:\Anaconda>ipython nc_hw2_hill_climb.py
Simple Hill Climb
X =  0.721759360058

Iterated Hill Climb
X =  0.10007646174

Stochastic Hill Climb
X =  0.101296488508

C:\Anaconda>ipython nc_hw2_hill_climb.py
Simple Hill Climb
X =  0.901745241427

Iterated Hill Climb
X =  0.100168148946

Stochastic Hill Climb
X =  0.103018083057

C:\Anaconda>ipython nc_hw2_hill_climb.py
Simple Hill Climb
X =  0.296639976009

Iterated Hill Climb
X =  0.100045357411

Stochastic Hill Climb
X =  0.0973894547446
```

Figure 1.1: Sample Hill Climb Algorithms Answers

1.2 Problem 2

Implement and apply the hill-climbing, simulated annealing, and genetic algorithms to maximize function $g(x)$ used in the previous exercise assuming a bitstring representation.

Tip: The pertubation to be introduced in the candidate solutions for the hill-climbing and simulated annealing algorithms may be implemented similarly to the point mutation in genetic algorithms. Note that in this case, no concern is required about the domain of x, because the binary representation already accounts for it.

Discuss the performance of the algorithms and asses their sensitivity in relation to the input parameters.

1.2.1 Summary

Expanding on the evaluations in the previous problem we add a simple genetic algorithm to the mix. As mentioned, many of the solutions above can find themselves caught in a local extrema for many optimization problems. The genetic algorithm (granted a large enough initial population) avoids this problem by sampling a larger area of the search space.

1.2.2 Implementation

The code can be found in Listing A.1.

Encoding

For the gene encoding, we used an unsigned 16 bit integer. This gave us an easy way to store 16 bits. Since we know the domain of the problem is restricted to $[0, 1]$ decoding the genotype into a phenotype was done by dividing the integer value of the bit string by the maximum value of a 16 bit integer (65535). Because the values are stored as integers, we can use numpy's random_integers function to initialize the population.

Crossover & Mutation

Crossover and mutation were handled by binary operations. Binary masks were stored in lists for each operator and a random index into those lists was generated to pick the mask used.

For mutation the individual was XOR'd with the mask to flip the proper bit. For crossover one parent was AND'd with the chosen mask and the other was AND'd with that masks negation. These values were then OR'd together to make the child individual.

Selection

Since our fitness values were simply the y values of the function and maximizing the function was the goal, a simple roulette selection method was chosen.

To avoid having to mess with the list of fitness values, an unused index list is created, and this is what the indexes are chosen from. Iterating through the unused list we add the value at that list to a variable until it is greater than or equal to a random value between 0 and the summation.

Between choosing each index the value at the chosen index is subtracted from the sum to avoid having to sum the unused values again. Unfortunately, since this uses floating point calculations it occasionally didn't reach the sum before the algorithm ran out of indexes. We fixed this problem by defaulting to the last index if the sum hasn't been reached by that point.

1.2.3 Analysis

Through multiple runs with different parameters, it was found that a genetic algorithm could solve this problem consistently with an initial population as low as 10 individuals in 250 iterations. The fastest

consistent solution came from runs with around 100 individuals and 100 iterations. If more iterations were run it was able to get more precise, giving an answer of .0999923704891 for a max of .999999956813. Taking into account that the encoding isn't perfect this probably means we have the global max at .1 with a value of 1.

1.3 Problem 7

Determine, using genetic programming (GP), the computer program (S-expression) that produces exactly the outputs presented in Table 3.3 for each value of x. The following hypotheses are given:

- **Use only functions with two arguments (binary trees).**
- **Largest depth allowed for each tree: 4**
- **Function set: F = +, ***
- **Terminal set: T = 0, 1, 2, 3, 4, 5, x**

x	Program output
-10	153
-9	120
-8	91
-7	66
-6	45
-5	28
-4	15
-3	6
-2	1
-1	0
0	3
1	10
2	21
3	36
4	55
5	78
6	105
7	136
8	171
9	210
10	253

2

Artificial Neural Networks - Text Chapter 4

2.1 Problem 17

Apply the MLP network trained with the backpropagation learning algorithm to solve the character recognition task of Section 4.4.2.

Determine a suitable network architecture and then test the resultant network sensitivity to noise the the test data. Test different noise levels, from 5% to 50%

3

Immunocomputing - Text Chapter 6

A

Code

Listing A.1: Problem2EA.py

```
from __future__ import print_function
import numpy as np
import math
import random

class EA:
    bit_masks = ( 0b0000000000000001,
                  0b0000000000000010,
                  0b0000000000000100,
                  0b0000000000001000,
                  0b0000000000010000,
                  0b0000000000100000,
                  0b0000000001000000,
                  0b0000000010000000,
                  0b0000000100000000,
                  0b0000001000000000,
                  0b0000010000000000,
                  0b0000100000000000,
                  0b0001000000000000,
                  0b0010000000000000,
                  0b0100000000000000,
                  0b1000000000000000 )

    reproduction_masks = ( 0b1111111111111100,
                           0b1111111111111000,
                           0b1111111111110000,
                           0b1111111111100000,
                           0b1111111111000000,
                           0b1111111110000000,
                           0b1111111100000000,
                           0b1111111000000000,
                           0b1111110000000000,
                           0b1111100000000000,
                           0b1111000000000000,
                           0b1110000000000000,
                           0b1100000000000000 )

    binary_format = '{:#018b}'
```

```
    def __init__(self, population_size, mutation_rate):
        self.population_size = population_size
        self.child_population_size = self.population_size // 2
        self.mutation_rate = mutation_rate
        self.population = np.random.random_integers(0, 65535, self.population_size).as
        self.population_eval = np.zeros(dtype=np.float, shape=(self.population_size))
        self.__next_population = np.zeros(dtype=np.uint16, shape=(self.population_size
+ self.child_population_size))
        self.__next_population_eval = np.zeros(dtype=np.float, shape=(self.population_
        self.__child_population = np.zeros(dtype=np.uint16, shape=(self.child_populati
        self.__child_population_eval = np.zeros(dtype=np.float, shape=(self.child_popu
        self.__child_pop_indices = range(self.child_population_size)
        self.__pop_indices = range(self.population_size)

        self.evaluate(self.population, self.population_eval);

    def print_best(self):
        index = 0
        max = 0
        for i in self.__pop_indices:
            if self.population_eval[i] > max:
                max = self.population_eval[i]
                index = i
        print(self.binary_format.format(self.population[index]), "\nX:", np.float32(se

    def evaluate(self, pop, eval):
        for i in range(pop.size):
            x = np.float32(pop[i]) / 65535
            eval[i] = (2 ** (-2 * (((x - 0.1)/.9) * ((x - 0.1)/.9))) * math.sin(5 * ma

    def select_indices(self, count, eval):
        unused = range(eval.size)
        indices = []
        i = 0
        sum = np.sum(eval)

        while i < count:
            i = i+1
            rand = np.random.uniform(0, sum)
            index = 0
            curr = eval[unused[index]]

            while curr < rand and index < len(unused) - 1:
                index = index + 1
                curr = curr + eval[unused[index]]

            sum = sum - eval[unused[index]]
            indices.append(unused.pop(index))

        return indices

    def create_children(self, indices):
        for i in self.__child_pop_indices:
```

```
            child = self.combine(self.population[indices[2 * i]], self.population[indi
            if np.random.uniform() < self.mutation_rate:
                child = self.mutate(child)

            self.__child_population[i] = child

        self.evaluate(self.__child_population, self.__child_population_eval)

        for i in self.__child_pop_indices:
            self.__next_population[self.population_size + i] = self.__child_population
            self.__next_population_eval[self.population_size + i] = self.__child_popul

    def combine(self, mom, dad):
        mask = self.reproduction_masks[random.randint(0, len(self.reproduction_masks)
        return (mom & mask) | (dad & ~mask)

    def mutate(self, child):
        mask = self.bit_masks[random.randint(0, len(self.bit_masks) - 1)]
        return child ^ mask

    def run(self, iterations, printEvery):
        i = 0
        while i < iterations:
            i = i + 1

            for j in self.__pop_indices:
                self.__next_population[j] = self.population[j]
                self.__next_population_eval[j] = self.population_eval[j]

            self.create_children(self.select_indices(self.population_size, self.popula

            indices = self.select_indices(self.population_size, self.__next_population

            for j in self.__pop_indices:
                self.population[j] = self.__next_population[indices[j]]
                self.population_eval[j] = self.__next_population_eval[indices[j]]

            if i != iterations and i % printEvery == 0:
                self.print_best();

        print("\nSOLUTION\n--------")
        self.print_best();



if __name__ == '__main__':
    test = EA(250, .1)
    test.run(250, 1)
```