

Second Assignment on Evolutionary Computation

Stefano Benedettini

Andrea Roli

Artificial Intelligence LM 2010

Contents

1	Second Assignment	1
1.1	EC for the Longest Common Subsequence Problem	1
1.2	Hard Instances for the Longest Common Subsequence Problem	2
1.3	Low Autocorrelation Binary Sequence Problem	3
1.4	Continuous Optimization with Genetic Algorithm	4
1.5	Symbolic Interpolation with Genetic Programming	5

Topics

- Genetic algorithms
- Genetic programming
- **Second assignment**

1 Second Assignment

For the first assignment you are required to design a family of Evolutionary Algorithm for *one* of the following problems. You must maintain the group of your previous assignment.

Each student or group, let's call them working unit, has to deliver an Evolutionary Algorithm written in `Python` language. As always, mind that the objective *is not* to mimic an algorithm proposed in literature *nor* improve on the state of the art!

Save all your results about computational experience in a machine-readable format.

1.1 EC for the Longest Common Subsequence Problem

Notice: This assignment is only for those groups that *didn't* tackle LCS in the previous assignment.

LCS is a problem that can naturally be tackled by a genetic algorithm. I bet you could find some literature on that. This assignment asks you to design your EC algorithm for LCS. You can choose whatever variant of genetic algorithm you prefer. Also memetics and master-slaves are well accepted.

- Design at least *two different* variants of your EC and compare them.
- Compare your algorithms on a subset of the instances handed out for the last assignment. Consider all instances with id (the final number in instance file name) in $\{1, 2\}$ and string length in $\{50, 100\}$ (32 instances total).
- Make a statistically significant number of runs (say 5) of your algorithms for each instance.
- See what happens if you initialize your initial population with randomly generated solution or with a more effective heuristic construction. Compare results from the two algorithms.
- Plot graphics of your data.

1.2 Hard Instances for the Longest Common Subsequence Problem

Notice: This assignment is only for those groups that *did* tackle LCS in the previous assignment.

For this assignment you will need your local search for the LCS you developed in the first assignment. The objective is to devise a genetic algorithm to generate *hard* instances for your specific algorithm. The algorithm must accept as parameters the number of strings in an instance and string length. We suppose that every string has the same length. Alphabet cardinality can optionally be an input or can be “evolved” during search. I suggest to make it bounded anyway so it does not inadvertently “explode” during algorithm runs. You have complete freedom on implementation (as long as it is **Python**), tools used (i.e., you may or may not employ PyEvolve) and more importantly algorithm design choices, though you might use some advices:

Solution representation: concerning solution representation (that is, the genotype) you have basically two choices:

Explicit representation: a genotype can be a one- or two-dimensional structure that holds the strings of your instance, so, in this toy example, the one-dimensional genotype `cabcdababacd` can map to instance $s_1 = \text{cabcdab}$, $s_2 = \text{abacd}$;

Indirect representation: a genotype represents the parameter that will be used by a parametrized instance generator to produce the instances to be evaluated. A sensible choice of parameters is therefore required. For instance, you can

decide that your instances have a fixed alphabet cardinality and a probability distribution over its letter, which give rise to one parameter for each letter.

The first choice is probably the easiest, because the second forces you to write also an instance generator. Nevertheless, the second choice provides much more insight into your algorithm since the best resulting genotype can compactly describe a whole family of potentially hard LCS instances.

Evaluation: no matter what genotype representation you select, you will have to evaluate it. The fitness should provide a measure of hardness of that genotype (LCS instance) with respect to your LCS solver. A possibility is to give a set amount of computational resources to your SLS algorithm and obtain solution cost. Obviously the lower the cost the higher the fitness.

Stochasticity: since your LCS solver will probably be a stochastic local search, you will have to cope with stochasticity. The problem is to estimate your solver performances on a single instance. A possible choice is to take the average solution cost on a set number of runs. This situation is somewhat different if you adopted an indirect representation *and* if your instance generator is stochastic itself: instead of generating a single solution and run your SLS on it a fixed number of times, you can generate that many instances and run your SLS *once* on every instance; afterwards you take the average of solution cost as before.

- Design at least *two different* variants of your EC and compare them.
- Generate hard instances for each combination of n and m where n is the number of strings and m is string length: $n = \{10, 50, 100\}$, $m = \{10, 20, 50, 75\}$.
- If alphabet cardinality is a parameter set it to $\{2, 4, 8, 24\}$.
- Make a statistically significant number of runs (say 5) of your algorithm for each (n, m) .
- Plot graphics of your data.

1.3 Low Autocorrelation Binary Sequence Problem

Given the miserable success of this problem previous appearance, I give it (and you) a second chance.

LABS problem aims at finding an n -length sequence $s_0, s_1, \dots, s_n, s_i \in \{-1, 1\}$ in order to minimize the energy E defined as:

$$C_k = \sum_{i=1}^{n-k} s_i s_{i+k}$$

$$E = \sum_{k=1}^{n-1} C_k^2$$

In the definition of the C_k 's you have to consider every pair of elements that are $k - 1$ positions apart. For instance, if $k = 1$ consider all adjacent elements $s_1s_2, s_2s_3, \dots, s_{n-1}s_n$, if $k = 2$ consider the sequence $s_1s_3, s_2s_4, \dots, s_{n-2}s_n$, up to $k = n - 1$ where you only consider s_1s_n .

- Design at least *two different* variants of your EC and compare them.
- Compare your algorithms on instance sizes $10, 11, \dots, 50$.
- Make a statistically significant number of runs (say 5) of your algorithm for each instance.
- Plot graphics of your data.

This problem is the fifth problem in CSPLib. Definition, results in literature and bibliography can be found here. On this site you can find tables that report optimum (or very good) energy values for different n : http://www.comp.nus.edu.sg/~stevenha/viz/results_labs.html.

1.4 Continuous Optimization with Genetic Algorithm

Design a genetic algorithm for continuous function optimization. You will test your algorithm on the benchmark used in CEC'08 Special Session on Global Optimization, an international competition on continuous optimization. You can find in the link references, results and a paper for each submitted solver. This competition follows a number of rules, that we will ignore. We will also reduce slightly the benchmark set. The attached file will provide all the details about the benchmark set.

For this exercise you will need to implement the functions comprising the benchmark set. For an efficient evaluation of multidimensional function, I suggest you have a look at Numpy libraries.

- Optimize both unimodal functions (F_1 and F_2) and 3 out of five multimodal functions (F_3 through F_7) with dimension (number of variables) $D \in \{5, 10, 20\}$. Disregard translation term $\mathbf{o}$. All functions should have a minimum at $\mathbf{0}$.
- Design at least *two different* variants of your EC and compare them. Following a good practice in this field, the termination condition should be the maximum number of function evaluation. If you have no idea on how to enforce such constraint, then set a maximum runtime.
- Make a statistically significant number of runs (say 5) of your algorithm for each instance.
- Plot graphics of your data.

1.5 Symbolic Interpolation with Genetic Programming

Design a Genetic Programming algorithm for symbolic interpolation. You will have to interpolate each function in CEC'08 benchmark described in the previous exercise. Consider only bi-dimensional version of benchmark functions, i.e., $D = 2$.

- Interpolate each and every function in the benchmark, from F_1 through F_7 . Disregard translation term $\mathbf{o}$.
- Design at least *two different* variants of your EC and compare them. One version should employ “simple” functions, that is, limit yourself to arithmetic ones, while the other should contain whatever function you see fit, from square roots to trigonometric functions.
- Make a statistically significant number of runs (say 5) of your algorithm for each instance.
- Plot graphics of your data.

The evaluation method shall be the following: draw uniform samples from the hypercube indicated in the property section for each test function in the attached document. You should see a domain definition for variable $\mathbf{x}$. For instance, in Rastrigin's Function $x \in [-5, 5]^D$. After you drew uniform samples $X = \{x_1, x_2, \dots, x_k\}$, evaluate them off-line and generate your goal-vector $Y = \langle f(x_i) \mid x_i \in X \rangle$. Now each evolved program will be evaluated this way: evaluate your program in each sample in X and obtain your estimate Z ; a program's fitness will be the mean squared error of Z with respect to Y that is, $\varepsilon = \frac{1}{k} \sum_{i=1}^k (y_i - z_i)^2$. Needless to say, the lower the better.