

Assignment 2 – Evolutionary Computation

Stefano Benedettini

Andrea Roli

Artificial Intelligence LM 2011

Contents

1	General Requirements	1
2	The Longest Common Subsequence Problem	2
3	The Haplotype Inference Problem	2
4	The Founder Sequence Reconstruction Problem	3
5	Continuous Optimisation	4

Topics

- Genetic algorithms
- Combinatorial Optimisation
- Continuous Optimisation

1 General Requirements

For the first assignment you are required to design a family of Evolutionary Algorithm for *one* of the following problems.

Each student or group, let's call them working unit, has to deliver a GA algorithm written in the C++ language. The algorithm should be parametrised in the sense that it should be possible for the user to tune its parameters in order to accommodate for different requirements or problem instances. Mind that the objective *is not* to mimic an algorithm proposed in literature *nor* improve on the state of the art!

For each problem, test instances and reference material are provided, such as papers, which are very useful as they provide insights and many hints like: greedy searches, smart construction heuristics, upper and lower bounds.

Additional requirements:

1. For all algorithm the user should also be able to provide a maximum run time
2. Save all your results in a machine-readable format as you are requested to provide a graphical analysis of your results

2 The Longest Common Subsequence Problem

LCS is a classical string problem. Given an alphabet Σ and a set of string $\mathcal{S} = \{s_1, s_2, \dots, s_n\}$ (lengths can be different), find the longest string t such that t is *subsequence* of each s_i . A string s' is said to be a subsequence of a string s if s' can be obtained by erasing characters from s . For example, let $\Sigma = \{a, b, c, d\}$ and $s = aabccdbad$ then ε (the empty string), $acdb$, $aaad$, cdd and s itself are subsequences while $aaaa$, $cbab$ or $abdc$ are not.

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.

Attached to this document you will find 80 random instances. Instance filename is read this way: first the alphabet cardinality, then the number of strings and finally their length. All string have (a fairly large) equal length of 1000 characters, but we will consider a trimmed down version.

- Design at least *two different* variants of your EC and compare them
- Compare your algorithms on a subset of the instances. Consider all instances with id (the final number in the instance file name) 1 and string length in $\{100, 200\}$ (16 instances total). If your algorithm is fast enough, you can try string length 500.
- Make a statistically significant number of runs of your algorithms for each instance
- See what happens if you initialise your initial population with randomly generated solution or with a more effective heuristic construction. Compare results of the two algorithms
- Plot graphics of your results

3 The Haplotype Inference Problem

The Haplotype Inference Problem (HIP) is an important problem in bioinformatics. A $m \times n$ HIP instance consists of a set $\mathcal{G}$ of m vectors $g_i \in \{0, 1, 2\}^n, \forall i = 1, 2, \dots, m$ called *genotypes*. The HIP aims to find a set $\mathcal{H} \subseteq \{0, 1\}^n$ of Boolean vectors of length n , called

haplotypes, such that for each genotype $g \in \mathcal{G}$ there exists a pair of haplotypes $\langle h, l \rangle$ which satisfies:

$$\left. \begin{array}{lcl} g[i] = 0 & \Rightarrow & h[i] = l[i] = 0 \\ g[i] = 1 & \Rightarrow & h[i] = l[i] = 1 \\ g[i] = 2 & \Rightarrow & h[i] \neq l[i] \end{array} \right\} \forall i = 1, 2, \dots, n$$

In this case, we say that $\langle h, l \rangle$ resolves g .

The objective is to minimise the cardinality of $\mathcal{H}$, that is, to find the minimal set of haplotypes that resolves a given set of genotypes.

Attached to this document you will find a large number of benchmark instances varying in dimension (number of genotypes and genotype length) and difficulty, **hapmap** and **uniform** being the “easiest”.

- Design at least *two different* variants of your EC and compare them
- Compare your algorithms on 20 instances. Try at least one instance per set. Notice that in **uniform** there are instances of various dimensions.
- Make a statistically significant number of runs of your algorithms for each instance
- Plot graphics of your results

4 The Founder Sequence Reconstruction Problem

The Founder Sequence Reconstruction Problem (FSRP) is another important problem in bioinformatics. A FSRP instance consists of a $m \times n$ matrix of Boolean elements R , called *recombinant* matrix where each row is called *recombinant*, and a number k_f . The aim is to find a $k_k \times n$ Boolean matrix $\mathcal{F}$, called *founder* matrix where each row is called *founder*, such that the *minimal decomposition* of R through $\mathcal{F}$ uses the minimal number of *breakpoints*.

Attached to this document you will find a three sets of benchmark instances of varying dimension (number of haplotypes and haplotype length) and difficulty, from **evo**, the easiest, to **rnd** the hardest. Inside the archive you will also find code to compute the solution value given recombinant and founder matrices.

- Design at least *two different* variants of your EC and compare them
- Compare your algorithms on 20 instances. Try at least one instance per set. Statistical analysis will not include **rnd** instances with 10 recombinants since they are too easy, you can use them freely for testing purposes though.
- Make a statistically significant number of runs of your algorithms for each instance
- Plot graphics of your results

5 Continuous Optimisation

Design a genetic algorithm for continuous function optimisation. You will test your algorithm on the benchmark used in CEC'08 Special Session on Global Optimisation, an international competition on continuous optimisation. 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.

You can find a C implementation of the functions comprising the benchmark in the attached archive.

- Optimise 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) $D \in \{5, 10, 25, 50, 100\}$, the latter two only if your algorithm can handle it. This degree of parametrisation, on both function type and dimension, should be easy to accomplish if you accurately design your code.
- Design at least *two different* variants of your EC and compare them. Following a good practise 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 of your algorithm for each instance.
- Plot graphics of your results