Evolutionary Computing Report

Jaro Camphuijsen - 6042473

October 17, 2017

1 Introduction

Evolutionary algorithms solve optimization problems. As most problems can be formulated as such, they are very powerful and can be used in many situations. Where evolutionary algorithm stand out is that they do not assume anything about the fitness landscape of the underlying problem. In this report we have three unknown ten dimensional functions which we want to optimize. This is the simplest possible mapping where the actual problem is used as the fitness function. It is given that the optima are real-valued, lying within the interval $[-5, 5]$. And it is known whether the function is multimodal, regular and/or seperable. This makes it so that this problem falls within the COP category, a Constrained Optimization Problem. [1]

2 Methods used & Comparison of results

The full configuration of the final Evulationary Algorithm can be found in table 2. The process to this configuration consisted of many steps of improvement of which the most important ones are explained.
To start off a **very simple EA** was constructed without parent selection (full population in mating pool) and mutation only with a single mutation rate. Survivor selection was present by selecting only the best part of the list of produced offspring. This algorithm already performed moderately on the simplest function which was not multimodal and separable. However to get better results some form of recombination should be implemented and parent selection showed to be of high importance as well.
First however the mutation operator was adjusted to have **multiple mutation rates and multiple self adapting step sizes**. This showed to be a major improvement since also on the multimodal and non separable functions a low but nonzero score was obtained.
Uniform arithmetic recombination, was added but the problem with this was that the only way to expand the search space was by mutation. Therefore the results even dropped after this implementation. **The Blend Crossover (BLX)** approach seemed more useful and improved results on the other functions as well.
Next some form of parent selection was introduced in the form of selecting the best few parents for reproduction. However the result was that the selection pressure was raised too much and only the simplest unimodal function reached

Representation	Real Valued Vectors
Recombination	Blend Crossover (BLX) recombination
Mutation	Gaussian perturbation
Parent Selection	Stochastic Universal Sampling
Survivor Selection	Full replacement with best offspring
Speciality	- Self adapting multiple stepsize - Time dependent stepsize lower bound - Time dependent parent selection pressure

Table 1: Final EA configuration

an optimum. Therefore the more sophisticated technique of **Stochastic Universal Sampling** was chosen with a ranking based Boltzmann Distribution. A cumulative probability function was computed at initialization and the selection pressure could be adjusted so the different functions could run with different selection pressures. The selection probability with which the cumulative probability distribution was calculated is given by:

$$p_{sel}(i) = \frac{1 - e^{-i \cdot \lambda}}{c} \tag{1}$$

Where λ is the selection pressure and c is a normalization constant.
From now on a lot of time was spent on adjusting and tuning the parameters of the algorithm. However one last major step that improved the results was to implement a **time dependent parent selection pressure**. This was done by computing the cumulative probability function at each evolution cycle with a selection pressure that depended on the fraction of evaluations run of the total number of evaluations. This way the system could first explore the full solution space and settle down in the minimum at a later stage of the algorithm evaluation.
Another minor change which improved results was making the **lower bound on the mutation step sizes time dependent**. This lower bound was implemented because the step sizes sometimes had a tendency to become very small at the start of the run. However implementing this bound resulted in a less accurate estimation of the optimum because the population could not settle down. Therefore by making it time dependent in the same way as the parent selection pressure, we got the same cooling effect. This improved exploration at first and exploitation in the later stage.
After this, the only improvements were reached by tuning the parameters of the algorithm. This was sometimes done by hand, keeping a log of the changes and results. And for some parameters this was done automatically with multiple runs with different setting and selecting the best performing ones.

3 Implementation

Since we use an object oriented programming language we can take advantage by defining a new kind of object for candidate solutions. A new Java class `Individual` from the `Comparable` interface was created with methods to set the Individual's parameters. Also the possibility for copying an individual was

created. Implementing the `Comparable` interface has the advantage that the sorting of individuals becomes really easy. Recombination and mutation were chosen to be implemented as methods in the main class since these methods would be changed multiple times in the process and in this way the `Individual` class could be left unchanged. Also recombination needs more than one individual so from a symmetry point of view this is also a more elegant choice. All other parts of the configuration (parent/offspring selection) were also implemented in the main function.

4 Recommendations

For further improvement, some more sophisticated (and maybe time dependent) way of offspring selection could be implemented. It seemed that making the selection pressure time dependent was a really important improvement and for future algorithm design this could be one of the first steps. Furthermore for improvement of results more parameter tuning or even Evolutionary parameter determination could be used, however this is very time consuming especially when training on the real functions since they were set to run only once per hour.
A recommendation for design could be to implement also a Population class next to an Individual class. This could improve readability of the code while maintaining the flexibility.
Final note to self: keep checking during the course whether your group members want to continue or dropped the course.

References

[1] A. Eiben and J. Smith, *Introduction to Evolutionary Computing*. Springer, 2003.