Experiences in Writing Evolutionary Algorithms

Optimizing a Secret Function Using an Evolutionary Algorithm in Java

Erik van Zijst and Sander van Loo
November 13th, 2003

Vrije Universiteit,
Faculty of Sciences, Department of Mathematics and Computer Science,
Amsterdam, The Netherlands

{erik,sander}@marketxs.com

1. Abstract

This text describes our experiences in maximizing the output of a secret function with *n* input variables using an evolutionary algorithm written in Java. We will experiment with different techniques including recombination, mutation and self-adaptation. After we have discovered which strategies yield the best results, we will use those to build an implementation that tries to find good values for the secret function.

2. Introduction

The function we want to optimize is a black box problem. It takes up to *n* double values as parameters where *n* in the dimension of the problem. Given a set parameters, the function returns a value in the range [-2, 11] which we will use as the fitness value of the input parameters in our evolutionary algorithm. The experiment comes with the constraint that only 10^6 fitness evaluations are allowed during maximization.

3. Mutation

The simplest algorithm is one that relies purely on mutation to find better values. Our first algorithm therefore generates a set of random initial values and applies mutation after every cycle. When the fitness value of the new mutated values is higher than the previous value, the new, mutated values are used in the next cycle. The black box problem takes a set of Java primitive doubles as its arguments and therefore we have to be careful when applying mutation, as floating point representations do not allow us to use conventional mutation strategies such as random bit-flipping. In our experiment we use non-uniform mutation by adding to a gene that is to be mutated a value randomly drawn from a Gaussian distribution with mean zero and a user-defined standard deviation. Since the black box problem only accepts values in the range of [0-1], we correct the mutated result if necessary. As an alternative to the Gaussian distribution, we could have used a Gauchy distribution[i] where there is a higher probability of generating a larger mutation value, while using the same standard deviation.

4. Recombination

In our second experiment we chose to add recombination to our test program. In order to use recombination or crossover (we shall use these terms interchangeably) we introduced a population of individuals where an individual is defined as a single one-dimensional array of double values that can be fed to the black box. We stored the population in a two-dimensional array of doubles where the width of the array is the dimension *n* of the black box and the height represents the number of individuals in the population. During an iteration of the algorithm, we first determine the absolute fitness value for every individual in the population. Based on these figures we derive the relative fitness figures that can be used to select the mating pool. Selecting parents for the mating pool is done by a simple algorithm that implements a roulette wheel. Each individual is mapped to a part of the wheel with a size equal to its relative fitness value. The bigger the slice, the larger the chance that it is selected for the mating pool. When an individual is selected twice, it is copied into the mating pool twice as well. We keep the size of the mating pool equal to the size of the population. Using the absolute fitness values for the roulette wheel becomes very inefficient when all individuals have a similar fitness (during the maximization, fitness values converge), as the wheel's slices become almost equal in size. To avoid this deficiency of absolute fitness selection[ii], we apply a simple Fitness

Proportional Selection (FPS) mechanism known as Windowing Down[iii] where we take the fitness value of the least fit member and subtract that value from all individuals. The remaining values are the relative fitness values and can be mapped to the roulette wheel. The advantage over absolute selection is that the selection mechanism is not slowed down by converged fitness values. However, still a number of disadvantages can be identified. For example, because the relative value of the least fit member is reduced to zero, it will never be selected for the mating pool. Also, outstanding individuals can quickly take over the entire population. This is known as *premature convergence*. An alternative FPS technique that we did not include in our tests, could be Goldberg's sigma scaling[iv], which includes the average as well as the standard deviation of the population's fitness values when computing relative values.

After the mating pool has been selected, crossover is applied. We have experimented with different applications of arithmetic recombination and eventually selected *Whole Arithmetic Recombination* for the rest of our tests as it is the most commonly used operator. It works by taking the weighted sum of the two parents for each double value:

$$\text{child1} = \alpha \cdot \overline{\text{parent1}} + (1-\alpha) \cdot \overline{\text{parent2}}$$
$$\text{child2} = \alpha \cdot \text{parent2} + (1-\alpha) \cdot \text{parent1}$$

To compute the weight of both parents, we use a constant parameter α. When α is 0.5, both parents are equally important and the result is the average of both values. This is also known as *Uniform Arithmetic Recombination*. When crossover has been applied on the mating pool, we swap the current population with the mating pool and use that for the next iteration.

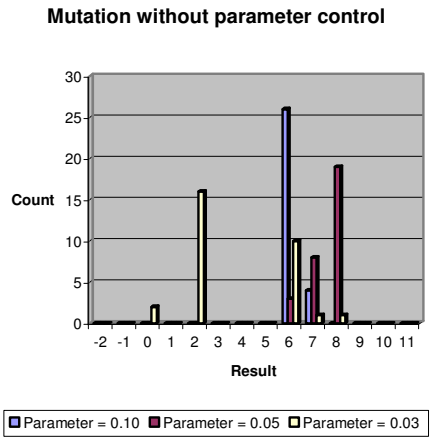
5. Dynamic Parameter Control

Since the running of the algorithm is a dynamic and adaptive process, we have to reflect this on the individual constant parameters that are used in the various parts of the algorithm. An example of these parameters is the standard deviation of the Gaussian distribution used to compute the mutation parameter[1]. Different values of the mutation parameter might be optimal at different stages of the evolutionary process. This dynamic nature of configuration parameters in general has been proven in the past[v,vi]. During our tests we found that large mutation steps are good in the beginning of the process to help the algorithm quickly find peaks in the black box function. However, after a peak has been found, smaller mutation steps are necessary to carefully converge to the most optimal solution. When a relatively large mutation parameter is used, the algorithm will quickly reach an interesting peak in the problem space, but will be unable to climb the peak high enough as the high mutation parameter does not allow for careful fine-tuning of the suboptimal chromosomes. In our test program we now start with a relatively high mutation parameter and shrink that value exponentially when the fitness value of the population grows nearer to its maximum. This works in both ways in that when the population's fitness accidentally deteriorates, the mutation parameter increases again.

1Note that we dynamically fine-tune the step-size of the mutation, rather than the rate at which mutation is performed.

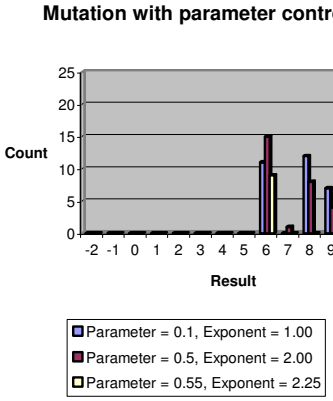
6. Findings

First we implemented non-uniform mutation with a fixed parameter. We followed normal practice [iii] and used a mutation probability of one per gene and use the mutation parameter to control the standard deviation.

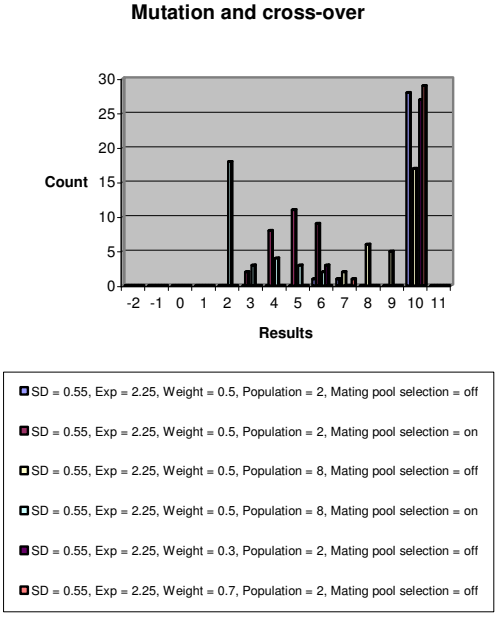


From observing the progress of the search during experiments we learned that a large mutation parameter was beneficial for the early stage of the search, but a smaller mutation parameter was required to optimize the results. Starting with a parameter that is too small will lead nowhere and starting with a parameter too big will find the rough contours of the problem space early on but fails to explore the interesting areas in depth. We

thus added parameter control to the mutation algorithm. The size of the mutation parameter is initially configured from the configuration file and then modified with the progress of the search. The standard deviation is decreased as the search progresses to create a zoom-in effect. The higher the fitness value of the current solution, the smaller the chance that the mutation algorithm will generate a large change. We experimented with linear and non-linear scaling of the mutation parameter.



After optimizing the parameter control we noticed that although we found good solutions frequently, we were also frequently stuck in some sub-optimal local maximum. We implemented arithmetic cross-over to overcome this problem.



7. Conclusions

From our experiments we concluded that we dealt with a multimodal problem. Progress dependant parameter control on non-uniform mutation proved very efficient in exploring local maximums. However, when only using mutation we frequently got stuck in sub-optimal regions of the search space, with a mutation parameter that was no longer large enough to allow us to deviate away from this sub-optimal maximum. We experimented with a zoom-out mechanism if no better value was found after N consecutive cycles but this did not yield reliable results. In the end cross-over proved a reliable way to get out of these local maximums. A higher arithmetic cross-over weight improved results. Enabling mating pool selection significantly worsened the results. Although we think that this can be explained by noting that the mating pool selection actually reduces the variety in the population and therefore decreases the chance that a new local maximum is found, it is an area for further research.

i X. Yoa, Y. Liu. *Fast Evolutionary Programming.* Proceedings of the 5th Annual Conference on Evolutionary Programming. MIT Press, Cambridge, MA, 1996

ii J.H. Holland. *Adaptation in Natural and Artificial Systems.* MIT Press, Cambridge, MA, 1992. 1st edition: 1975, The University of Michigan Press, Ann Arbor

iii A.E. Eiben, J.E. Smith. *Introduction to Evolutionary Computing.* Springer, Berlin, Heidelberg, New York, 2003, page 59

iv D.E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning.* Addison-Wesley, 1989

v T. Bäck. *The Interaction of Mutation Rate, Selection and Self-Adaptation within a Genetic Algorithm.* Proceedings of the 2nd Conference on Parallel Problem Solving from Nature. Amsterdam, 1992, pages 85-94

vi L. Davis. *Adapting Operator Probabilities in Genetic Algorithms.* Proceedings of the 3rd International Conference on Genetic Algorithms. Morgan Kaufmann, San Francisco, 1989, pages 61-69