Evolutamine: Evolutionary Computing

Computational Science - Vrije Universiteit Amsterdam

Martijn van Beest
fbt600@student.vu.nl
1620746

Rahiel Kasim
rkm700@student.vu.nl
2184532

20 October 2017

1 Introduction

We were given three 10-dimensional functions and were tasked with designing and implementing an algorithm to find their global maxima. Each function has a domain consisting of 10 real numbers: $x_1, x_2, \dots x_{10}$ where $x_i \in [-5, 5] \subset \mathbb{R}$. The functions return a value in $[0, 10] \subset \mathbb{R}$ that signifies the closeness to the maximum where a larger value is closer and a value of 10 corresponds with the maximum. This problem as described is a constrained optimisation problem (COP). For every function there is an evaluation limit: the maximum number of times the function is allowed to be evaluated by the algorithm.

Finding the maxima by brute-force is infeasible: because the inputs are real numbers, the number of possible inputs are uncountably infinite. Given such an enormous search space we desire an algorithm that terminates within polynomial time and finds a "good enough" solution i.e. close enough to the maximum. These requirements can be fulfilled by an evolutionary algorithm (EA).

In this report we describe the design and implementation of our EA. Our main reference was the textbook by Eiben and Smith [2].

The provided functions are Bent Cigar, Katsuura and Schaffers F7. An interesting property that both the Katsuura and the Schaffers F7 function posses is multimodality. After a description of our methodology in section 2 we will expand on the multimodality property in section 3.

2 Methodology

Because there is no single EA that works perfectly for all problems, our approach was to start with a simple EA and to experiment with the individual components. In general an EA has a set number of components that differ between algorithms: parent selection, recombination, mutation and survivor selection. We have tried different methods for each of these components and eventually settled on the combination of methods that produced the best results.

The first step in designing an EA is deciding on a representation of individuals. For our problem there is a very natural mapping between phenotypes and genotypes. Our phenotypes are tuples of 10 real numbers $(x_1, x_2, \dots x_{10})$ that comprise the input to a function. Real numbers have infinite precision while numbers represented on computers usually have finite precision. There are methods so computers can compute with numbers of arbitrary precision, but this will be large trade of performance for precision. We've instead chosen to represent the real numbers as double-precision floating-point numbers. This format is standardized by the IEEE and double-precision floating-point arithmetic is supported by most common processors, so it is fast [1].

Note that this encoding of phenotypes to genotypes is surjective but not injective: (infinitely) many phenotypes are mapped to the same genotype. We don't consider this an issue as the precision of doubles is at around 15 decimal digits for numbers in the range $[-5, 5]$ [8].

For the initialization of the EA we start with a population of some size and provide each individual with 10 uniformly sampled doubles (genes) in the range $[-5, 5]$. The evolution of the population is described in the following sections.

2.1 Parent Selection

Our first step in the evolutionary cycle is the selection of individuals that will reproduce to create offspring. Most of the methods we have implemented base the selection on the fitness of the individuals. We calculate the fitness of individuals with the provided fitness function.

2.1.1 Fitness Proportional Selection

In a first attempt to write an EA, *fitness proportional selection* (FPS) was used. This method calculates a probability for each individual to be selected for the mating pool by dividing the individuals fitness by the total fitness of the entire population. Hence, the fittest individual is most likely to be chosen for reproduction. The parents are then sampled independently. A known drawback of FPS is *premature convergence*, where a few fit individuals take over the entire population. This phenomenon immediately occurred in our population. Since our initial population is random, their fitness is very bad. There is however always an individual that is bad, but still significantly better than the other individuals. This individual gets sampled for the mating pool every time, due to its probability close to 1. Without advanced recombination or mutation, the population does not even change anymore.

2.1.2 Ranking Selection

To overcome the problem of premature convergence, we implemented *ranking selection* (RS). This method still samples individuals with some probability, but the probability does not depend on the absolute fitness of the individual anymore. The probability is rather based on the rank of the individual in the population. Ranks are distributed by sorting the population based on their fitness. The fittest individual acquires rank 1, the least fit individual acquires rank n, where n is the population size. The rank is then mapped to a probability by either a linear (1) or an exponential ranking function 2.

$$P_{\text{lin-rank}} = \frac{(2-s)}{\mu} + \frac{2i(s-1)}{\mu(\mu-1)} \quad (1)$$

$$P_{\text{exp-rank}} = \frac{1-e^{-i}}{c} \quad (2)$$

2.2 Recombination

We've experimented with all of the recombination methods described in section "4.4.3 Recombination Operators for Real-Valued Representation" of the book by Eiben and Smith [2]. In the following the arithmetic average of x and y means $\alpha x + (1-\alpha)y$ with some chosen α. In our experiments the value of α was uniformly chosen from $[0, 1]$ for each recombination.

The first method is called "Simple Arithmetic Recombination". Given two equal length vectors of size n, we choose a recombination point $k \in [1, n-1]$. Child 1 inherits the first k values of parent 1 and the following $(n-k)$ values are the arithmetic averages of the parents. Child 2 receives its first k values from parent 2 and the rest is the same as for child 1.

The second method "Single Arithmetic Recombination" is simpler. We pick a random position $k \in [1, n-1]$ and the children receive the arithmetic average of their parents' value at position k. The values at all other positions are directly inherited from their parents (child 1 gets the values of parent 1 and child 2 of parent 2).

The method we ended up using is called "Whole Arithmetic Recombination". Here all the values of the children are the arithmetic averages of the values of the parents. Note that the children are not identical unless $\alpha = 0.5$ (or if the parents are identical).

The final method discussed in this section is the "Blend Crossover". Here the idea is that with usual recombination, the value of the children is somewhere between the values of the parents. While with this crossover the children can obtain values outside of the range of their parents. This provides more exploration for an EA. After implementing this method we saw no (significant) increase in the score, so we opted to use the simpler "Whole Arithmetic Recombination" method instead.

2.3 Mutation

For mutation we've evaluated the methods described in the section "4.4 Real-Valued or Floating-Point representation" of the book by Eiben and Smith [2]. As the methods described in this section increased in complexity as the text continued, so did the EA achieve better results for our problem. We've implemented all the mutations except "Correlated Mutations" and selected "Uncorrelated Mutation with n Step Sizes" due to its superior results.

The section starts out with a simple method called "Uniform Mutation" where the idea is to reset a value with a new uniformly drawn number. This method is quite destructive as any correlation with the previous value is lost. A more gentle method with better results is "Nonuniform Mutation", here the value is drawn from a Gaussian distribution with a standard deviation or mutation step size σ with zero mean. The new values are normally close to the previous values, but larger deviations are still possible.

A better method still is "Uncorrelated Mutation with One Step Size", where the mutation rate itself becomes a parameter subject to evolution. In this mutation mechanism first the mutation rate is mutated and then this new rate is used for the mutation of the actual values of interest. This order is important: by using the new mutation rate in the mutation the following fitness test of the new individual is also a fitness test of the new mutation rate. In this method the same mutation rate σ is used for all dimensions.

The last improvement towards our final method employs a different σ_i for each dimension x_i in the method called "Uncorrelated Mutation with n Step Sizes". Here each individual has $2n$ parameters in evolution: its n values x_i and the related n mutation rates σ_i. The insight here is that different dimensions don't necessarily correlate, so they are provisioned with independent mutation rates.

The "Correlated Mutations" method takes the self-adaptation of the mutation even further by having the direction of mutation in the phase space as evolvable parameters. This is accomplished by introducing $n_\alpha = \frac{n \cdot (n-1)}{2}$ parameters α_i that specify the mutual angles between all n dimensions. The total number of parameters under evolution would then be: $2n + \frac{n \cdot (n-1)}{2} = \frac{n^2+3n}{2}$. This is a substantial increase in variables to evolve, and here one has to really consider if it is worth. Because if we have more evolution parameters, the algorithm has more options to consider and so more evaluations are needed to adapt to the fitness function. This is a disadvantage given our evaluation limit. More importantly, this method samples random vectors from a n-dimensional Gaussian, an algorithm absent from Java's standard library. We were unable to import the `MultivariateNormalDistribution` class from the Apache Commons library [5] due to unfamiliarity with the Java ecosystem and thus we could not implement this mutation mechanism.

2.4 Survivor Selection

In our first approach we used a generational model as a replacement strategy. This was the simplest way to complete a full evolutionary cycle. In later versions, we switched to a more sophisticated fitness-based method to select surviving individuals. Both $(\mu + \lambda)$ and (μ, λ) selection were considered, but since we had to work with a limited number of evaluations, we decided to create just one child for every parent (i.e. $\mu = \lambda$). The (μ, λ) strategy typically uses $\lambda > \mu$ (more offspring than parents), which resulted in our choice for the $(\mu + \lambda)$ strategy (although we would have preferred to use $\lambda > \mu$ here as well, to induce selection pressure).

We have not changed the replacement strategy after we implemented $(\mu + \lambda)$ selection. On hindsight, we should have given it another look when we implemented self-adaptive mutation and our method to overcome multimodality, given the advantages of (μ, λ) over $(\mu + \lambda)$ in these situations listed at the end of section 5.3 of the book by Eiben and Smith [2]. It is very likely our algorithm suffered from these drawbacks. We did however implement a tournament selection strategy, but only used it in the implementation of a *crowding* strategy, which will be discussed in section 3.1.

3 Multimodality

As we have mentioned in the introduction, the Katsuura and Schaffers function are multimodal. However, the methods described in section 2 turned out to be good enough to yield

Representation	Real-Valued Vector
Parent Selection	Linear Ranking
Recombination	Whole Arithmetic Recombination
Mutation	Gaussian Perturbation
Survivor Selection	Fitness-based replacement by $(\mu + \lambda)$
Initialisation	Random
Termination condition	Limited number of fitness evaluations
Specialty	- Self-adaptation of mutation step sizes - Time-dependent lower bound on mutation step sizes

Table 1: EA configuration

a fairly good score (> 8) for the Schaffers function. On the Katsuura function on the other hand, our EA performed only slightly better than a random search. The problem with multimodal functions is that the EA can get stuck in a local maximum. The rest of the search space will not be explored. Our EA clearly suffered from this. It was therefore important to maintain diversity in our population to be able to uncover new high-fitness regions in the search space. The book by Eiben and Smith [2] describes several methods to address this issue. We will describe the methods we have implemented in the following subsections.

3.1 Crowding

Our first attempt to preserve diversity was the implementation of *deterministic crowding*, an improvement by Mahfoud [3] over the original approach. The general idea of deterministic crowding is that offspring competes with similar parents for survival, such that individuals stay distributed over the search space. It works as follows:

1. Each parent in the population is randomly paired with another parent.
2. The pair produces two children via recombination.
3. The offspring is mutated and evaluated
4. Some distance metric between the four possible parent-offspring pairs is calculated. Since we are working with Real-Valued genotypes, we used the Euclidean distance [7].
5. Labeling the parents p_1 and p_2 and the offspring o_1 and o_2, the “combined-pair-distance” is minimized such that $d(p_1, o_1) + d(p_2, o_2) < d(p_1, o_2) + d(p_2, o_1)$. Then, o_1 competes in a tournament for survival with p_1, o_2 competes with p_2. If the other pairing turns out to be minimal in distance, they compete in that configuration.

The result of this is that offspring tends to compete with the most similar parent. Thus, replacement is done locally and individuals that explore an area are replaced by individuals that explore the same area. Assuming the initial population is well-spread over the search space, this preserves diversity. Unfortunately, deterministic crowding did not yield an improved performance for our EA.

3.2 Fitness Sharing

Figure 1 illustrates the difference between fitness-sharing and crowding (This figure and its caption are directly taken from the book by Eiben and Smith [2] (Fig. 5.4)). Fitness sharing tries to distribute the individuals proportional to the peaks they are exploring where crowding tries to distribute evenly among peaks. Fitness sharing calculates the distance between every pair (i, j) in the population (including self-pairings) and adjusts the fitness for every individual i by:

$$F'(i) = \frac{F(i)}{\sum_j sh(d(i,j))}. \tag{3}$$

Here, d is the distance for which we re-used our Euclidean

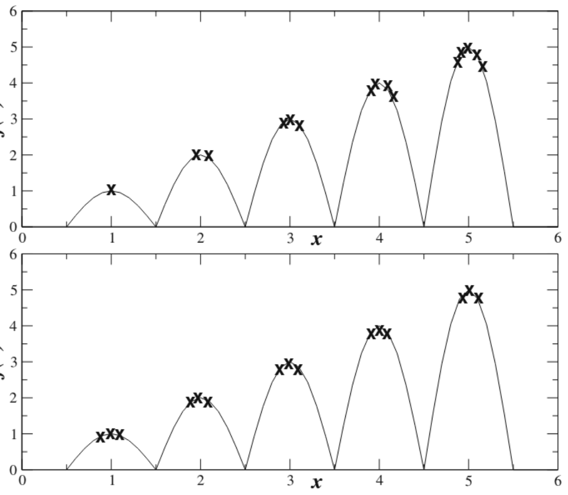

Figure 1: Idealised population distributions under fitness sharing (top) and crowding (bottom). There are five peaks in the landscape with fitnesses (5,4,3,2,1) and the population size is 15. Fitness sharing allocates individuals to peaks in proportion to their fitness, whereas crowding distributes the population evenly amongst the peaks.

distance function. Details on the sharing function sh can be found in chapter 5.5.3 of the book by Eiben and Smith [2]. There was a big drawback for our EA using fitness sharing. As can be seen in (3), we need to calculate the distance for each possible pairing in our population. For a population of size n, this requires n^2 calculations. We can exploit the symmetry of the distance property, but that still leaves $\frac{1}{2}n^2$ calculations, which can be a performance issue when n becomes large. Because the first results were hardly any better than a random search, we decided that this time performance penalty (which seemed to matter for the online contest) was not worth the effort of optimizing the fitness sharing method. Hence, we implemented a third method to overcome multimodality.

3.3 Island Model

As a last approach, we implemented the *island model* described in chapter 5.5.6 of the book by Eiben and Smith [2]. The main idea is very simple. Instead of keeping a single population, one keeps several subpopulations in parallel. Ideally, every subpopulation explores a different region of the search space. Every so many generations, a small number of individuals is exchanged amongst the islands, to increase exploration when recombination happens between newcomers and inhabitants. When the individuals are migrated, they replace the

Islands	5
Epoch length	50 generations
Migration size	5
Migration selection	Copy Best
Migration type	Effective Move

Table 2: Island Model configuration

worst individuals on the island of destination. Between exchanges, the populations use the methods described in section 2 to exploit their particular region of the search space. Table 2 shows the configuration of our Island Model. Our EA improved significantly on the Katsuura function. Instead of slightly outperforming a random search, this method scored around 3. We have experimented with distinct settings for the different islands, but this did not improve the performance further.

4 Further Improvements

Once the desired methods were implemented, the focus shifted to the parameters we used. The right values for your parameters are very important for the performance of your EA. Some parameters, like the epoch length and the migration size in the island model, were chosen based on the literature, in this case the book by Eiben and Smith [2]. Other parameter values were found by tuning.

4.1 Parameter Tuning

The most important parameters in our EA turned out to be the population size, and the mutation parameters τ, τ' and ϵ. Since we had a limited number of evaluations, the bigger the population the fewer generations we could create. We kept the population size therefore between 20-50 for each function, depending on the limit. The appropriate way to find the right values for the remaining parameters would be through a grid search. However, due to time pressure we performed most of the tuning manually and recorded the results, keeping the best settings. We moved the mutation parameters from their respective mutation functions to the global settings to be able the distinguish between functions. The mutation parameters τ and τ' turned out to be tricky. According to the literature we should set the τ values in our uncorrelated mutation with n step sizes as follows: $\tau' \propto \frac{1}{\sqrt{2n}}$ and $\tau \propto \frac{1}{\sqrt{2\sqrt{n}}}$.

Interestingly, when we experimented with proportionally different values, we acquired better results. With the contest in mind, we decided to use the tuned parameter values: $\tau' \propto \frac{1}{n}$ and $\tau \propto \frac{n}{10}$.

4.2 Time dependent variables

Another major change was the dependency on time for parameters. In the first generations one wants to explore the search space, but after some time one might rather exploit the uncover high-fitness regions. We used a self-adaptive mutation step size σ which is bounded by some ϵ such that

$$\sigma < \epsilon \implies \sigma = \epsilon.$$

We implemented a time dependency for the lower bound on the mutation step size. The result of this is that in early stages mutation can be large so there is a lot of exploration, but in later stages close to the evaluation limit the reduced lower bound allows for smaller mutations, such that the high-fitness region can be exploited. This improvement bumped our performance on Katsuura to a score of 8.5. We planned to make the selection pressure time dependent as well, but we were not able to implement this in time. We strongly believe this would have been a big improvement.

5 Implementation

In this section we will give a brief overview of the java specific choices that we made regarding the implementation. All our code is available at our online repository [4].

We wanted to keep the main class as clean as possible. Therefore we created an `Individual` class to store a candidate. We also created a seperate `Population` class to store multiple `Individuals`. The `Population` class also contains all the functions that implement the components of the EA, except for mutation. The mutation operator works on individuals, so it made sense to put all the mutation functions in the `Individual` class. Since we planned from the beginning to implement a lot of different methods per EA component, we created an `Options` class which can be passed to a `Population` object. The `Options` class contains several `Enum` types to differentiate between component methods, as well as some default parameter settings. This allowed us to easily switch between methods. Lastly, we implemented a separate `IslandModel` class, which in turn contains an array of `Population` objects. To be as flexible as possible in our main class, both the `Population` class and the `IslandModel` class implement a custom `EAPopulation` interface, which specifies the basic EA components.

6 Results & Conclusion

In figure 2 you can see the results of our EA. The algorithm achieves a score of $9.999977801879053 \pm 2.050283620022662\mathrm{e}{-05}$ on BentCigar, $8.893163092810429 \pm 0.38613823273700193$ on Katsuura and $9.783751107206767 \pm 0.5746303080312153$ on the Schaffers function. These scores are the average over 25 runs, and the errors the corresponding standard deviation. It shows our EA works well on the BentCigar and Schaffers functions, but less so on Katsuura. From the boxplot its also clear that there is a much bigger variation in the scores achieved on Katsuura. A possible cause is the multimodality of Katsuura: our probabilistic algorithm has trouble finding the global maximum. This is also seen in the results for the Schaffers function: the crystals shown in figure 2 are the outliers.

7 Recommendations

While experimenting with the recombination methods from section 2.2 we always chose the recombination parameter α uniformly from $[0, 1]$. It would be interesting to also experiment with constant values for α, such as the commonly uses $\alpha = 0.5$.

As mentioned in section 2.3 we have an incomplete implementation of the “Correlated Mutations” method where the only missing part is an import of a class from an external package. We used a makefile to manage compiling the program. It could be worthwhile to learn how to use Maven, a build tool

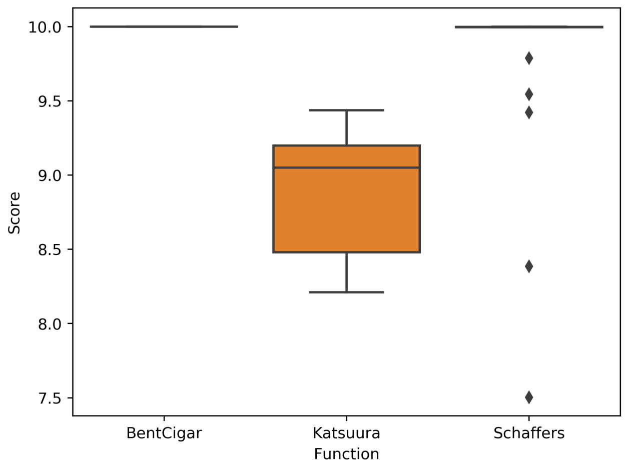

Figure 2: A boxplot of the scores achieved by the EA on the three functions.

specially crafted for use with Java programs [6]. We expect that this build tool alleviates the problems one stumble upon when trying to use an external Java package.

Another recommendation is to have an extensive grid search for the optimal parameter values. Since tuned mostly manually, we probably haven't found the best settings, which might be a reason why we have not found the function maxima.

More sophisticated adaptation of variables as we did with the bound on the mutation step size woulc probably have lead to better scores. An adaptive selection pressure would have been a very nice improvement.

An idea left unexplored is prohibiting incest. The possible defects in offspring from incestuous relations have been researched [9]. It would be amusing to see if prohibiting incest in our program would also lead to fitter individuals.

References

[1] IEEE Standard for Binary Floating-Point Arithmetic. *ANSI/IEEE Std 754-1985* (1985).

[2] Eiben, A. E., Smith, J. E., et al. *Introduction to evolutionary computing*, vol. 53. Springer.

[3] Mahfoud, S. W. Crowding and preselection revisited. *Urbana 51* (1992), 61801.

[4] Martijn van Beest, Rahiel Kasim. Evolutamine. [https://github.com/sunsistemo/evolutamine; accessed 20-October-2017].

[5] Wikipedia. Apache Commons — Wikipedia, The Free Encyclopedia, 2017. [https://en.wikipedia.org/w/index.php?title=Apache_Commons&oldid=777251653; accessed -October-2017].

[6] Wikipedia. Apache Maven — Wikipedia, The Free Encyclopedia, 2017. [https://en.wikipedia.org/w/index.php?title=Apache_Maven&oldid=803745745; accessed 20-October-2017].

[7] Wikipedia. Euclidean distance — Wikipedia, The Free Encyclopedia, 2017. [https://en.wikipedia.org/wiki/Euclidean_distance; accessed -October-2017].

[8] Wikipedia. IEEE 754 — Wikipedia, The Free Encyclopedia, 2017. [https://en.wikipedia.org/w/index.php?title=IEEE_754&oldid=805674690; accessed 19-October-2017].

[9] Wikipedia. Inbreeding — Wikipedia, The Free Encyclopedia, 2017. [https://en.wikipedia.org/w/index.php?title=Inbreeding&oldid=805228048; accessed 20-October-2017].