JavaEvA
A Java based framework
for Evolutionary Algorithms
- Manual and Documentation -

Felix Streichert and Holger Ulmer
Tech. Report WSI-2005-06

ISSN 0946-3852
Centre for Bioinformatics Tübingen (ZBIT)
of the Eberhard-Karls-University Tübingen
Sand 1, 72076 Tübingen, Germany

Abstract

The package JavaEvA (a Java implementation of Evolutionary Algorithms) is a general modular framework with an inherent client server structure to solve practical optimization problems. This package was especially designed to test and develop new approaches for Evolutionary Algorithms and to utilize them in real-world applications.
JavaEvA already provides implementations of the most common Evolutionary Algorithms, like Genetic Algorithms, CHC Adaptive Search, Population Based Incremental Learning, Evolution Strategies, Model-Assisted Evolution Strategies, Genetic Programming and Grammatical Evolution. In addition the modular framework of JavaEvA allows everyone to add their own optimization modules to meet their specific requirements.
The JavaEvA package uses a generic GUI framework that allows GUI access to any member of a class if get and set methods are provided and an editor is defined for the given data type. This approach allows very fast development cycles, since hardly any additional effort is necessary for implementing GUI elements, while still at the same time user specific GUI elements can be developed and integrated to increase usability.
Since we cannot anticipate specific optimization problem and requirements, it is necessary for users to define their optimization problem. Therefore, we provide an additional framework and explain how one can include JavaEvA in an existing Java project or how one can implement ones own optimization problem and optimize it by using JavaEvA. This gives users total control of the optimization algorithms used.
In the following chapters we will give a short introduction to JavaEvA in ch. 1 and to optimization algorithms in ch. 2. In ch. 3 we give a detailed description of the modules in JavaEvA from the general users and the developers perspective. Finally in ch. 4 we give a tutorial how to implement one's own optimization problem and how to use JavaEvA in a users's framework.

Contents

Chapter 1

Introduction

JavaEvA (a Java implementation of Evolutionary Algorithms) is a modular framework for Evolutionary Algorithms (EA) and other iterative and preferably population based Optimization Algorithms, like Simulated Annealing or simple Hill-Climbing strategies. JavaEvA has been developed as a resumption of the former EvA (Evolutionary Algorithms) software package [47, 46] [1].
The optimization toolbox JavaEvA aims at two possible groups of users. First, the standard user who does not know much about the theory of Evolutionary Algorithms, but wants to use Evolutionary Algorithms to solve his application problem. Second, the experienced programmer who wants to investigate the performance of different optimization algorithms or wants to compare the effect of alternative mutation or crossover operators. The latter usually knows more about Evolutionary Algorithms or Optimization Algorithms and is able to extend JavaEvA by adding more specialized optimization strategies or solution representations.
Currently, JavaEvA encompasses the following optimization strategies:

- Monte-Carlo Search (MCS)
- Hill-Climbing (HC)
- Simulated Annealing (SA)
- Genetic Algorithms (GA)
- CHC Adaptive Search (CHC)
- Population Based Incremental Learning (PBIL)
- Evolution Strategies (ES)
- Model-Assisted Evolution Strategies (MAES)

[1]For details on EvA package please visit our web page at http://www-ra.informatik.uni-tuebingen.de/forschung/eva/welcome_e.html

- Genetic Programming (GP)
- Grammatical Evolution (GE)
- Multi-Objective Evolutionary Algorithms (MOEA)

Although this already covers the most common EA optimization strategies, JavaEvA can easily be extended with other more problem specific optimization strategies like Gradient Descent or the Simplex Algorithm, due to the general modular framework of JavaEvA. These algorithms can be used in combination with the existing EA approaches, which leads to so called Memetic Algorithms, or in competition against them.

One key element of JavaEvA that allows multiple future extensions are the model assisted approaches, like MAES, which use a model of the fitness function as surrogate and detect local features of the search space to guide the optimization process and to save target function evaluations for faster speed of convergence. Currently, the implemented mathematical models range from classical Polynomial Regression Models, Radial-Basis-Function networks, Gaussian Processes, to Support Vector and Relevance Vector Machines. These models are not limited to MAES approaches, but can also be used for Global Optimization strategies like Efficient Global Optimization (EGO).

There are several advantages of a Java implementation of Evolutionary Algorithms. First, using Java the optimization toolbox can be run on any operating system that supports a Java Virtual Machine using Java 1.4, among others, Windows (2000, XP), Linux (SuSE, RedHat, Fedora), AIX and Solaris. Second, Java allows easy integration into web based services. The Java WebStart application of JavaEvA on the JavaEvA web pages is just one example for that. Third, the Remote Method Invocation (RMI) of Java allows easy distribution of an application using an intranet or the internet. For example, JavaEvA uses a client/server architecture that allows one to start the server running the optimization algorithms on one machine, while accessing the server using a client running on a totally different machine. Further, RMI can be used to parallelize Evolutionary Algorithms, either on the level of populations using an island-model with occasional communication between subpopulations or on the level of target function evaluations. In general, EAs are quite easy to parallelize, due to their population based optimization approach, and the Java inherent support for distributed computation makes it even easier.

JavaEvA was developed as part of a Project on "Automated Crystallization" [2]. In that project JavaEvAm was developed for on-line optimization of production parameters for an automated crystallization process for crystalline catalysts. The production process was to be optimized in respect to the yield of the production

[2] This project was sponsored by the German Federal Ministry of Research and Eductation (BMBF), contract No. 03C0309E. For details on this project please visit our web page at http://www-ra.informatik.uni-tuebingen.de/forschung/kombikat/welcome_e.html

output and the catalytic properties of the crystals produced. Because the target function evaluation required real-world experiments, which were not only time consuming but also quite expensive, the previously mentioned MAES approach was developed in JavaEvA to suit the requirements of this real-world optimization problem.
Currently, JavaEvA is also used as educational and teaching tool for the lectures on 'Genetic Algorithms and Evolution Strategies' to illustrate the general behavior of EA approaches and the effect of special mutation or crossover operators. A standard student project is to define an optimization problem encountered in their field of studies, to implement it as optimization problem for JavaEvA and to solve it using the available tools in JavaEvA.
This documentation on JavaEvA gives a short introduction to optimization algorithms in chap. 2, while a detailed description of the modules in JavaEvA is given from the general users perspective, in ch. 3. Finally in ch. 4 a tutorial is given on how to implement one's own optimization problem and how to use JavaEvA in one's own framework to optimize it. The examples for the tutorial, additional quick tour guides and the JavaEvA download links can be found on our JavaEvA web pages at http://www-ra.informatik.uni-tuebingen.de/software/JavaEvA/. Ch. 5 gives a short FAQ, which is updated continuously.

Chapter 2

Optimization Algorithms

Optimization problems are some of the most common application problems. An optimization problem is given as a search for the best (optimal) solution x from a number of possible solution alternatives, the search space L. The quality measure that identifies the 'best' solution is given by a target function $y = f(x)$, which may be multidimensional in y or even a priori unknown. The target value y is either to be maximized or minimized to be optimal, depending on the given application problem. The solutions x_{opt} that produce such an optimal target value are called global optima, if no $x \in L$ produces a better target value.
An optimization algorithm is a process that searches the search space L for the global optimum. A simple example for such an optimization algorithm would be a total enumeration approach that tries all possible solutions x. Unfortunately, this approach is rather inefficient with increasing problem dimension, even if more efficient covering methods are used. Other approaches utilize special properties of the target function to find the optimal solution through an efficient heuristic, e.g. Linear or Quadratic Programming. If the target function is real-valued and differentiable, local search algorithms like gradient descent can be applied. In case the gradient of the target function is not available or the target function is multimodal, e.g. contains multiple local and global optima, still some less efficient global optimization strategies can be applied. While local search algorithms use local properties of the search space to guide the search, global search algorithms tend to take the whole or at least a larger part of the search space into consideration for exploration.
In case the optimization problem is still too complex to be searched using standard approaches due to high dimensionality, noise, non-linearity or other unusual properties of the search space, the choice of available optimization algorithms becomes significantly smaller. In such cases Evolutionary Algorithms (EAs) are often applied to search for the optimum. Unfortunately, EAs are not guaranteed to find the global optimal solutions, but they are often able to find sufficiently good solutions within a limited amount of time.

Evolutionary Algorithms are stochastic, iterative, population based approaches, which are inspired by the mechanisms of natural evolution. Charles Darwin first postulated that natural evolution is based on the principle of the 'survival of the fittest' and improvements are caused by random alterations in the reproduction cycle. In biology these random alteration are either caused by random mutations or by recombination.

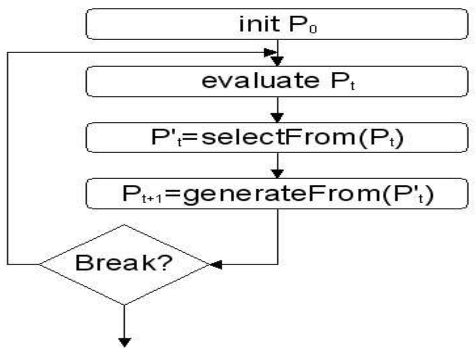


Figure 2.1: The general scheme of an Evolutionary Algorithm.

EAs start with a population P of randomly initialized solutions and iteratively apply directed selection and reproduction using recombination and mutation until a termination criterion is met, see fig. 2.1. This way the EA gradually produces more and more adapted/optimized solutions and converges to close to optimal solutions. Contrary to other optimization algorithms the EA requires only the output of the target function for a given solution to guide the selection process and a suitable solution representation to apply the evolutionary operators of recombination and mutation to.

Therefore, EAs can be applied to many kinds of optimization problems, where classical optimization approaches often fail. The target function may be noisy, non-linear, non-differentiable, discontinuous, multi-modal, high dimensional and may be subject to multiple kinds of constraints.

Another class of optimization problems, where EAs have been successfully applied are multi-objective optimization problems. Here multiple, often conflicting, objectives y are to be optimized at the same time. Most standard optimization algorithms have considerable difficulties to solve multi-objective optimization problems. They often require repetitive optimization runs with varying control parameters to obtain the complete Pareto-front for a given multi-objective optimization problem. In case of EAs, on the other hand, we only need to extend the selection process to a multi-objective selection and the multiple solutions maintained in a population based approach allow EAs to approximate the Pareto-front in a single optimization run. But additional precautions are necessary, to maintain a diverse population to cover the complete Pareto-front.

Since JavaEvA was developed as a toolbox for Evolutionary Algorithms, most of the implemented optimization strategies are stochastic, iterative and population based approaches. To describe these algorithms, it is necessary to introduce some terms, see tab. 2.1. These terms will later be used to explain some of the features of the optimization algorithms implemented in JavaEvA.

In the following chapters we will give short descriptions of the optimization algorithms implemented in JavaEvA, ranging from simple Monte-Carlo Search and Hill-

Term	Abbreviation	Meaning in JavaEvA
Solution	x	Possible solution for a given problem or sample of the search space. If x is a vector n gives the length of the vector. The variables x are often called decision variables.
Population	$P(t)$	Set of alternative solutions at time t. $\|P(t)\| = m$ gives the size of the population.
Individual	a_i	Data structure stored in a population, which gives a possible solution x. i is the index within a population.
Fitness	$y = f(x)$	Quality measure or target value for a solution x.
Generation	t	Index of current iteration of the optimization process.
Parents	P_p	Subset of $P(t)$ selected to generate new individuals.
Offspring	P_c	Individuals generated from parents through recombination and mutation.
Crossover		Method that mixes decision variables of multiple individuals. Operates either on the level of genotypes or on the level of phenotypes.
Mutation		Method that alters decision variables in an individual. Operates either on the level of genotypes or on the level of phenotypes.
Genotype		Data structure of an individual that gives the solution representation. Typically the evolutionary operators of mutation and crossover act on the genotype.
Phenotype		Solution representation that can be evaluated by the target function. Often needs to be decoded from the genotype of an individual.

Table 2.1: Nomenclature

Climbing approaches to Genetic Algorithms and Evolution Strategies. Although these are general descriptions of the optimization algorithms, in some cases we will comment on the properties of the JavaEvA implementation to increase the insight in the optimization toolbox. This is necessary, since some implementations may require special representations or problem properties and cannot be applied to arbitrary problem instances.

2.1 General Approaches

Some general non-evolutionary approaches are also implemented in JavaEvA. Although these approaches are often local optimization techniques, we implemented them as iterative, population based algorithms to fit into the general scheme of evolutionary approaches in JavaEvA. In most cases the population is simply interpreted as the multi-start variant of a local search algorithm.
All of the here mentioned general approaches can be applied to all problem instances, since they solely rely on the initialization and mutation methods on given solution representations. Although these general approaches do not represent the most sophisticated optimization strategies, they can be used to analyze the search space. For example, in case the Monte-Carlo Search performs as well as the Hill-Climber or a Genetic Algorithm, the search space is either extremely shallow or too rugged and non-causal to be optimized efficiently. In case of simple search spaces with a single global optimum or only few local optima the Hill-Climber or Simulated Annealing strategies often outperform the population based Evolutionary Algorithms. Therefore, it is always a good choice to apply the general approaches first to get some idea about the characteristics of the search space.

2.1.1 Monte-Carlo Search

Monte-Carlo (MC) Search, also known as Random Search, is a blind search strategy without any feed-back. Basically the Monte-Carlo Search performs a random sampling of the complete search space, while memorizing the best solution found so far, see alg. 1 for details. The population in this scheme serves no other purpose than to initialize multiple solutions in one iteration instead of one. It is important to note that it does not effect the performance of the search strategy whether a population size of one individual is used or a population size of 100,000, but bigger population sizes may speed up the plot graphics.

```
t = 0;
result = initNewSolution();
evaluate(result);
while isNotTerminated() do
    a = initNewSolution();
    evaluate(a);
    if a.isBetterThan(result) then
        result = a;
    end
    t = t +1;
end
```

Algorithm 1: General scheme of the Monte-Carlo search strategy

2.1.2 Hill-Climber

In JavaEvA a simple Hill-Climbing (HC) strategy is also implemented. In this strategy the algorithm starts with a random initial solution. In each generation the current solution is mutated. If the mutant is better, the mutant replaces the current solution else the current solution is kept for the next generation, see alg. 2 for details. This way the Hill-Climber performs a greedy local search [29]. This strategy can be rather efficient in simple unimodal search spaces, but is prone to premature convergence in local optima in case of multimodal search spaces. To reduce the chance of premature convergence the Hill-Climbing strategies can be extended to a Multi-Start Hill-Climber, where multiple local Hill-Climber start from randomly chosen initial solutions. In JavaEvA the population size gives the number of multi-starts for the Hill-Climber (MS-HC).

```
t = 0;
result = initNewSolution();
evaluate(result);
while isNotTerminated() do
    a = clone(result);
    mutate(a);
    evaluate(a);
    if a.isBetterThan(result) then
        result = a;
    end
    t = t + 1;
end
```

Algorithm 2: General scheme of the Hill-Climbing search strategy

2.1.3 Simulated Annealing

Simulated Annealing (SA) is quite similar to Hill-Climbing but less strict regarding the replacing scheme [21]. Instead of only keeping the best solution Simulated Annealing allows temporary degradation of the solution quality. This strategy uses an analogy between the mechanism of metals cooling and freezing into a minimum energy crystalline structure (the annealing process) and the search for a minimum in a given solution space. This mechanism is simulated using a control parameter T, which is decreased during the optimization process, see alg. 3. The degradation function for T is called the Annealing Schedule and causes the optimization process to become more and more restrictive accepting worse solutions toward the end of the optimization process. The speed of convergence and also the vulnerability to premature convergence of the Simulated Annealing strategy depends on this Annealing Schedule. In the example in alg. 3 we use a simple linear Annealing Schedule with

$0 \leq \alpha \leq$, but other more complicated Annealing Schedules are also possible.
The ability to allow temporary degradation in solution quality enables the Simulation Annealing strategy to escape local optima. Similar to the Hill-Climber, a population in Simulated Annealing is interpreted as multi-start strategy (MS-SA), which can further decrease the chance of premature converge in a local optimum.

```
t = 0;
T = 1.0;
result = initNewSolution();
evaluate(result);
while isNotTerminated() do
    a = result.clone();
    mutate(a);
    evaluate(a);
    if RNG.flipCoin($e^{\frac{-\Delta Fitness}{T_t}}$) then
        result = a;
    end
    T = $\alpha\cdot$ T;
    t = t + 1;
end
```

Algorithm 3: General scheme of the Simulated Annealing search strategy

2.2 Genetic Algorithms

Genetic Algorithms (GA) are inspired by the principle of natural evolution to achieve incremental adaptation of an individual to a given environment to increase the fitness of the individual. If the target function of an optimization problem is interpreted as fitness measure for a given individual and the individual represents a possible solution for the optimization problem, the evolutionary process can be applied to arbitrary optimization problems. Although there have been suggested multiple alternative GA implementations in the recent years, the original implementation by Holland [18] can be considered as the blue print of GA and it also the most intuitive translation of natural evolution into an algorithm, see alg. 4. GA utilize the evolutionary operators of selection and random variation, through recombination/crossover and mutation, repeatedly on a population of possible solutions (individuals). GA are therefore a stochastic, population-based search heuristic, which requires nothing but the target function of the optimization problem to guide its search.

```
t = 0;
initialize(P(t));
evaluate(P(t));
while isNotTerminated() do
    $P_p$(t) = selectParentsFrom(P(t));
    $P_c$(t) = reproduction($P_p(t)$);
    mutate($P_c$(t));
    evaluate($P_c$(t));
    P(t+1) = buildNextGenerationFrom($P_c$(t), P(t));
    t = t +1;
end
```

Algorithm 4: General scheme of a generational Genetic Algorithm

The initial population $P(t = 0)$ of the GA is usually initialized randomly to give a broad sampling of the search space. This initial population is then evaluated, i.e. tested on the optimization problem, before the generational cycle of the GA is entered. The GA first selects possible parents P_p from the current population $P(t)$ based on their achieved fitness. Those parents P_p are then used to generate a population of offspring P_c either by generating simple clones of the parents or through recombination of multiple parents. Recombination is designed to exchange traits of the solutions represented in the parent individuals to generate new combinations of those traits in the offspring, but recombination by chance may also produce perfect clones of the parents. The offspring P_c are then subject to random mutation, which may again alter the traits of the solutions stored in the individuals. Then the offspring are evaluated to determine their fitness on the optimization problem. Finally, the next generation is generated from the current population $P(t)$ and the offspring

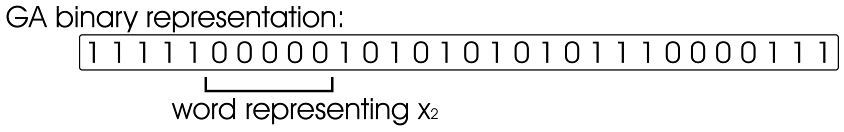


Figure 2.2: An exemplary GA genotype

$P_c(t)$, utilizing a distinct generation strategy ranging from complete replacement ($P(t+1) = P_c(t)$) with or without elitism to the gradual changes in case of the steady-state GA. Finally, there is some termination criterion to be met before the GA breaks the iterative optimization process.
Although there is not much of theoretical work about GA, there is some ranging from the early works on the schema theorem or the building-block hypothesis to more recent approaches utilizing Markov-Chain approaches [17].
In the following sections we will give details on the basic elements of the GA like solution representation, selection mechanisms, recombination/crossover and mutation operators, generation strategies and termination criteria.

2.2.1 Solution Representations

The original GA by Holland [18] used a binary solution representation $\tilde{x}$ of fixed length l for the GA individuals to perform the evolutionary operators on. More generally speaking the GA uses a string of characters as solution representation using an alphabet of h characters. In case this representation can not be mapped directly on the decision variables x of the optimization problem, a dual solution representation is required and also a mapping or coding function between them $\Gamma(\tilde{x}) \to x$. In analogy to nature the binary GA representation $\tilde{x}$ is called genotype and the translated decision variables are interpreted as the phenotype x.
To illustrate this geno-/phenotype duality we give an example how to encode six real-valued decision variables x_i with $(-1 \leq x_i \leq 1)$ in a genotype of length $l = 30$ using a binary alphabet of $h = 2$ characters and a standard binary encoding as mapping function. To do so, the genotype string $\tilde{x}$ is partitioned into six words of length $L = 5$, each word coding a single decision variable x_i, see fig. 2.2. A standard binary encoding, see equ. 2.1, is able to decode a word $\tilde{x}_i$ into a real-valued decision parameter x_i, if the lower and upper bounds of the range are given ($l_i = -1, u_i = 1$). In equ. 2.1 $\tilde{x}_{i,k}$ gives the value of the k-th bit of the word $\tilde{x}_i$.

$$x_i = \Gamma(\tilde{x}_i) = l_i + \frac{u_i - l_i}{2^L - 1} \cdot \sum_{k=0}^{L} \left(\tilde{x}_{i,k} \cdot 2^k\right) \tag{2.1}$$

This way the genotype given in fig. 2.2 can be decoded to $n = 6$ real-valued variables $x = \{1.0; -1.0; 0.36; -0.36; -0.55; 0.81\}$. The length of a word L limits the precision.

In this case the precision of the binary representation is limited to $\frac{u_i - l_i}{2^L - 1} \approx 0.065$. It is important to note that the evolutionary operators of crossover and mutation usually ignore the word boundaries.
There are multiple extensions to GAs that use alternative solution representation utilizing non-binary alphabets, double chromosomes, real-valued vectors or even program trees, which leads to Genetic Programming, see sec. 2.7.

Initialization

Often random initialization is used for GAs, because random initialization is said to generate a diverse sampling of the complete search space. This diversity is necessary for the GA to search the complete search space. In case of small population sizes or nonuniform initial distribution the GA may be limited to a local search. Therefore, bigger initial population sizes or an improved initial distribution like D-optimal design may improve the general performance of a GA.
Additionally, problem specific knowledge may be used to find a suitable initial distribution close to an assumed global optimum to reduce the necessary computational effort of the optimization process.

Solution Evaluation and Constraints

Solution evaluation is usually straight forward by using the target function to set the fitness value of the GA individual. But to save computational effort for extremely time consuming target function evaluations, it can be necessary to employ 'lazy' evaluation. This technique can be used if a partial evaluation of the target function is sufficient to guess the fitness of an individual. This guess can be sufficient to guide the evolutionary search while still far away from the optimum and to omit the rest of the time consuming evaluation in case of an extremely bad individual. An alternative approach uses model surrogates for the true fitness function evaluation to save computational effort and to increase the quality of the solutions found, see sec. 2.6 for more details on this approach.
Another problem is how to implement constraints not only in GAs but in EAs in general. In the field of EA constraints can be distinguished into two basic types: hard and soft constraints. Hard constraints, on the one hand, may not be violated in any circumstances. If violated, hard constraints may even cause the target function to be incomputable and therefore prevent fitness assignment for the EA individuals. Soft constraints, on the other hand, may be violated, i.e. the target function can still be evaluated and the solution may be interesting, although conflicting with some constraints. There are several approaches how to deal with constraints:

- **Legal Representation/Operators**: this method is suited for both hard and soft constraints and uses specialized representations and evolutionary operators. In this approach the representation and operators are designed in such a way that only legal phenotypes can be generated. The previous encoding and the associated mapping function in equ. 2.1 gives an example of how to meet a simple range constraint simply by including the range in the mapping function $\Gamma()$. Depending on the complexity of the constraints this approach becomes less and less feasible, since it may reduce the search space and some regions of the search space may become unreachable.

- **Repair Mechanisms**: this method is also suited for both types of constraints and transforms an infeasible solution to a feasible one before evaluation. This way the GA is still able to reach every point in the search space, but the search space may be become neutral, this means that some alteration in the genotype do not affect the phenotype or the fitness of an individual.
- **Penalty**: this approach is only suited for soft constraints and penalizes an individual violating a constraint by decreasing the fitness of the individual proportional to the amount of violation. In this method all portions of the search space can still be reached, but a final solution is not guaranteed to be feasible. This effect can be countered by higher penalties, but it may also lead to a reevaluation of the constraints, if the infeasible solution shows beneficial properties.
- **Lethal Punishment**: this approach brings the penalty to the extreme and may be used for both hard and soft constraints. In this case every infeasible solution is removed from the population, preventing the individual from being selected as parent by killing it off right after creation. Again like the previously mentioned legal representation/operators method this approach may discard useful genetic material and even may be prohibitively slow in some problem instances.
- **Multi-objective Optimization**: this method is similar to the penalty approach and can only be applied to soft constraints. This approach transforms the constraints to additional objectives, i.e. the minimization of constraint violation. This does not really simplify the optimization problem, but it may lead to some new solution alternatives previously not considered feasible.

See also [49] for more details on this topic.

2.2.2 Selection

The standard GA uses parent selection to guide the search toward the global optimum. Depending on the selection method used the selection process bears different selection pressures on the population. In case of a multi-modal search space, too high selection pressures and too small population sizes may lead to premature convergence.

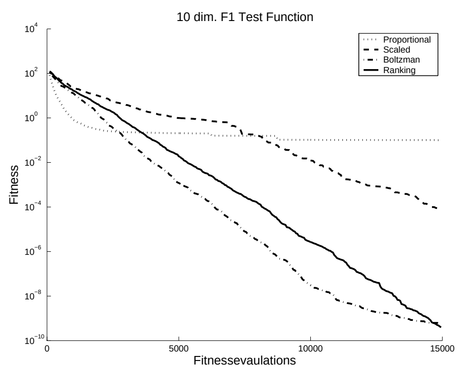


Figure 2.3: Comparing four methods to calculate selection probabilities.

Best Selection, $(\mu \overset{+}{,} \lambda)$-Selection

The best selection strategy selects the $|P_p(t)| = \mu$ best individuals from a population of $|P(t)| = \lambda$ individuals. This selection strategy is also used for Evolution Strategies, see sec. 2.5, and exerts the strongest selection pressure on the GA.

Tournament Selection

The tournament selection method iteratively selects the best individual from a randomly selected subset of the population of size $q \leq |P(t)|$ until μ individuals have been selected. The choice of q gives the selection pressure ranging from pure elitism $q = |P(t)|$ to random selection $q = 1$.

Probabilistic Selection Methods

Some selection methods require a selection probability p_i for each individual of the population, which sums up to one, $\sum_{i=0}^{n} p_i = 1$. There are multiple ways how to calculate this selection probability p_i from the fitness ϕ_i of a given individual. The fitness ϕ_i is to be maximized in this example, which allows easier mappings to the selection probabilities p_i:

- **Proportional selection probability**: requires positive fitness values.

$$p_i = \frac{\phi_i}{\sum_{j=0}^{n} \phi_j} \tag{2.2}$$

- **Scaled selection probability**: eliminates the impact of an offset on the fitness values by subtracting the fitness of the worst individual from every fitness value ϕ_i.

$$p_i = \frac{\phi_i - min_{k=0}^{n}(\phi_k)}{\sum_{j=0}^{n}(\phi_j - min_{k=0}^{n}(\phi_k))} \tag{2.3}$$

- **Boltzmann selection**: eliminates the effect of an offset, but also the effect of a multiplicative factor on the fitness function by using the standard deviation σ_ϕ of the fitness of the population.

$$p_i = \frac{e^{\frac{q \cdot \phi_i}{\sqrt{\sigma_\phi}}}}{\sum_{j=0}^{n} \left(e^{\frac{q \cdot \phi_j}{\sqrt{\sigma_\phi}}} \right)} \tag{2.4}$$

- **Ranked selection probability**: uses the rank of the individuals $rank_i$ calculated by sorting the individuals according to their fitness values. With additional Max and Min values for the resulting selection probability the selection probability can easily be calculated.

$$p_i = Min + (Max - Min) \cdot \frac{rank_i - 1}{|P(t)| - 1} \tag{2.5}$$

Fig. 2.3 illustrates the effect of the different methods to calculate the selection probability on the 10 dimensional F1 test function, where the fitness is to be minimized, using an elitist generational GA with population size 50, roulette-wheel selection and real-valued solution representation.
With the selection probability calculated, there are again multiple ways how to select the parents from the current population $P(t)$:

- **Roulette-Wheel Selection**: is a very common selection method for GAs and simulates a toss in a roulette-wheel to select an individual. Each individual is assigned a segment on the roulette-wheel proportional to its selection probability p_i, see fig. 2.4. This selection scheme is repeated until μ individuals have been selected.

- **Stochastic-Universal Sampling**: is similar to roulette-wheel selection but uses g equidistant pointers instead of a single one, see fig. 2.5. This way random fluctuations in the composition of the selected individuals due to noise are reduced.

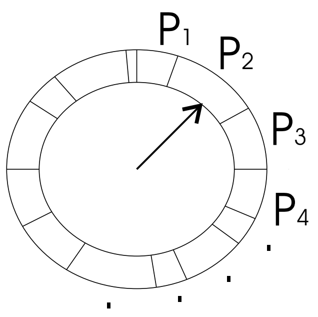


Figure 2.4: The Roulette-Wheel selection method.

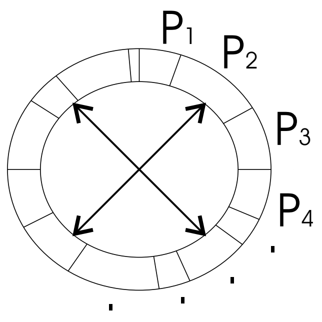


Figure 2.5: The Stochastic-Universal Sampling method ($g = 4$).

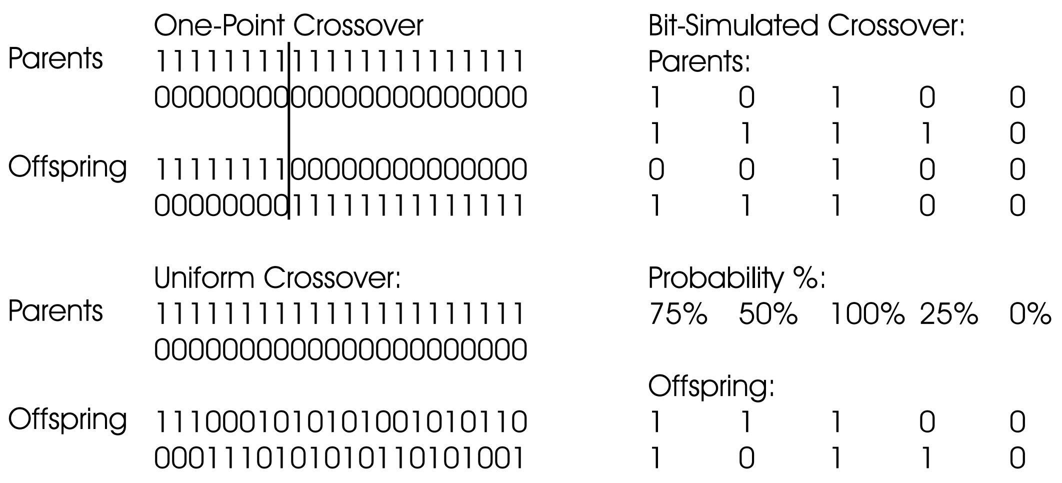


Figure 2.6: Crossover methods on binary genotypes

Extended Selection Methods

To prevent premature convergence, special extended selection methods have been developed. Fitness Sharing for example punishes individuals that are too similar to each other to maintain population diversity by reducing their fitness proportional to the distance to other individuals [15]. Another method is mating restriction, which can be considered a special selection operator for selecting mating partners that are similar to each other [6].

2.2.3 Recombination/Crossover

During reproduction offspring are either generated as perfect clones of the parents or through recombination of the parents. The crossover probability p_c gives the chance for recombination. Recombination tries to combine solution traits of multiple parents to generate novel solutions, which hopefully represent a suitable combination of positive traits. Usually, crossover is considered to be the primary operator in GAs. This assumption is often based on the Schemata Theorem, which was already introduced by Holland [18].
In this description we will limit to recombination/crossover operators to binary genotypes. Operators suitable for real-valued representations will be discussed in sec. 2.5.

- **One-Point crossover**: randomly selects a crossover point in the genotype of the parents, splits the genotype at that point and exchanges the elements to generate two offspring. To maintain the size of the genotype the crossover point has to be the same for both parents, compare upper left part of fig. 2.6

for an example, else the genotype needs to be of variable length.

- **N-Point crossover**: selects N crossover points in the genotype of the parents instead of just one [5].

- **Uniform crossover**: determines for each bit position in the genotype of an offspring randomly which parent to use to set the value of the bit [42], see lower left part of fig. 2.6 for an example.

- **Bit-Simulated crossover**: takes k parents and calculates the average value μ_i for each bit position over the whole set of parents, see right hand side of fig. 2.6. This vector of average values gives a vector $\bar{\mu}$ of probabilities for a bit at position i. $\bar{\mu}$ is then used to initialize the genotype of the offspring [43]. This vector $\bar{\mu}$ resembles the Vector V in Population Based Incremental Learning, see sec. 2.4.

The crossover operators outlined here can also be applied to k parents instead of just two parents.

2.2.4 Mutation

Mutation is often considered to be a secondary operator in GA, limited to recovering lost traits in the population. But depending on the representation used and the causality of the search space, mutation can easily advance to the most important operator, especially in case of real-valued representations used in Evolution Strategies, which are described in sec. 2.5. The mutation probability p_m gives the chance of mutation.

A simple mutation operator on a binary representation could invert a randomly chosen bit, see Invert-Bit Mutation in fig. 2.7, or swap two bits of different values in the representation, see Swap-Bits Mutation in fig. 2.7. But mutations can also lead to major alterations like inversion of larger sections of the genotype or translocation of a given part of the genotype to a different location.

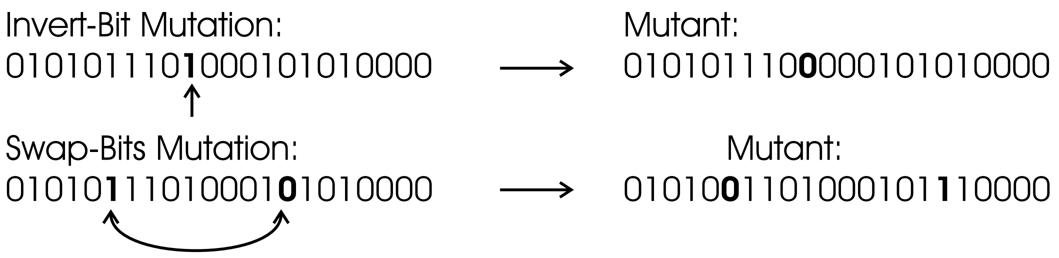


Figure 2.7: Mutation methods on binary genotypes

2.2.5 Generation Strategies

Three basic types of GA generational strategies can be distinguished, the generational GA, the steady-state GA and the generation gap GA. Depending on the generation strategy the selection pressure on the population may vary.

Generational GA

In case of the generational GA, the next generation $P(t+1)$ is given by the offspring population $P(t+1) = P_c(t)$. Thus, each generation the population is completely replaced by newly generated individuals. To increase the convergence rate of the generational GA, elitism is often used, with adds the k best individuals of the old population $P(t)$ to the next generation $P(t+1)$. This way a monotonic increase of fitness is guaranteed during the course of optimization.

Steady-State GA

The steady-state GA only creates one new individual per generation and eventually inserts this individual into the population by using a replacing strategy. Possible alternatives for the replacing strategy are, to replace a parent individual of the new individual to replace the worst individual in the population, or stochastically select the individual to be replaced proportional to the inverse of the fitness.

Generation Gap GA

The generation gap GA basically interpolates between the generational GA and the steady-state GA [20]. A parameter k gives the size of the population $P(t)$ to be replaced in the next generation $P(t+1)$ by new individuals. In case of $k = |P(t)|$ the generation gap GA is equal to the generational GA, since the population is completely replaced by new individuals. And in case of $k = 1$ the generation gap GA behaves like the steady-state GA. Again there are multiple alternatives how individuals are to be selected that participate in the next generation or that are to be replaced by new individuals.

2.2.6 Termination Criteria

There are multiple ways to terminate a GA. Here we just want to list the most common approaches to terminate a GA:

- Terminate after a given period of time. This approach ensures that the optimization process terminates after a predefined time period, depending on the requirement of a given application environment. Alternatively, this approach can be use to compare two algorithms regarding efficiency, in case the complexity of the optimization algorithm is higher than the computational effort necessary for the fitness evaluations.

- Terminate after a given number of function calls. This allows the comparison of algorithms based on the efficiency utilizing the samples of the search space to guide the search, in case the computational effort is mainly defined by the fitness evaluations, e.g. real world experiments or time consuming computational effort.
- Terminate after a sufficient solution quality is reached. Useful for many practical optimization problems, but may fail to terminate, if the targeted solution quality is too optimistic.
- Using a halting window, which terminates the GA, if the GA fails to improve the currently known best solution for k successive generations.

2.3 CHC Adaptive Search Strategy

The standard GA is usually based on five basic concepts: First, random initialization of $P(0)$. Second, selection of parents biased toward selecting the better individuals. Third, selection for the next generation $P(t+1)$ is often unbiased. Fourth, the recombination operator especially in case of binary representations is usually not disruptive. And finally, a low rate of mutation is used to maintain population diversity.
The CHC Adaptive search (CHC) strategy suggested by Eshelman [9] breaks with at least four of these traditional concepts of GAs, but the general structure is the same see alg. 5. While the initialization is still random, the CHC selects individuals for the next generation rather than for reproduction. Second, a new bias is introduced against mating individuals that are too similar. Third, CHC uses a highly disruptive crossover operator similar to the uniform crossover. Finally, instead of utilizing mutation for generating diversity in the population the CHC partially reinitializes the population if convergence is detected. Convergence is detected by using either an equals operator on $P(t)$ and $P(t+1)$ or by using the difference threshold d. The difference threshold is initially set to $d = L/4$, which L the length of the binary genotype, and after using the *diverge()* operator to $d = r \cdot (1.0 - r) \cdot L$ with r the divergence rate.

```
t = 0;
d = L/4;
initialize(P(t));
evaluate(P(t));
while isNotTerminated() do
    $P_p$(t) = selectParentsFrom(P(t));
    $P_c$(t) = reproduction($P_p(t)$);
    mutate($P_c$(t));
    evaluate($P_c$(t));
    P(t+1) = buildNextGenerationFrom($P_c$(t), P(t));
    if (P(t+1).equals(P(t)) then
        d = d -1;
    end
    if ($d < 0$) then
        diverge(P(t+1));
        d = $r \cdot (1.0 - r) \cdot L$;
    end
    t = t +1;
end
```

Algorithm 5: General scheme of the CHC Adaptive Search.

2.4 Population Based Incremental Learning

Following the work of Syswerda on Bit-Simulated crossover for GA, Baluja has created the Population Based Incremental Learning (PBIL) algorithm as an abstraction of the standard GA [2, 1, 27, 26]. The concept of PBIL rests on the idea that a GA population of solutions could also be represented by some statistics on the gene pool of the population. For a binary genotype consisting of l bits a probability vector with l real values $0 \leq V_i \leq 1$ is introduced. The probability vector $\bar{V}$ represents the chance to create a true bit a position i. Each generation n new individuals are created using the current probability vector, all individuals are then evaluated and the best individual $\hat{x}$ is used to update the probability vector using the learn rate LR.

$$V_{i,t+1} = V_{i,t} \cdot (1.0 - LR) + \hat{x}_i \cdot LR \tag{2.6}$$

Mutations on the probability vector can also occur to prevent premature convergence in local optima. In addition, $\bar{V}$ can be updated using more than one positive example and also using negative examples with a negative learn rate LR.

PBIL is not limited to binary genotypes, but has been extended to deal with real-

```
t = 0;
initialize($V_t$);
/*create new population using $V$ */
initializeFrom(P(t), $V_t$);
evaluate(P(t));
while isNotTerminated() do
    initializeFrom(P(t), $V_t$);
    evaluate(P(t));
    $\hat{x}$ = getBestFrom(P(t));
    /*update each component $i$ of the probability vector $V$ */
    for (int i = 0; i < $\hat{x}$.length; i++) do
        $V_{i,t+1} = V_{i,t} \cdot (1.0 - LR) + \hat{x}_i \cdot LR$;
    end
    mutateProbabilityVector($V_t$);
    t = t +1;
end
```

Algorithm 6: General scheme of the PBIL algorithm

valued representations [40, 14], permutations [44] and even program trees [36, 37]. The Bayesian Optimization Algorithm (BOA) extends the standard PBIL approach to identify dependencies between variables [32]. More recently PBIL algorithms became known as ‘Estimation of Distribution Algorithms’. A nice introduction to PBIL methods and optimization by probabilistic models is given in [31, 25].

2.5 Evolution Strategies

Evolution Strategies (ES) were developed by Rechenberg and Schwefel to solve numerical optimization problems in technical engineering [35, 38, 39]. ES are specialized on real-valued search spaces, applying very sophisticated mutation operators and often acting like a localized search with strong selection pressure and small population sizes. Therefore, ES have developed to be a optimization strategy, which is not as closely connected to the natural example as GA.
ES apply a so called $(\mu \overset{+}{,} \lambda)$-strategy starting from a random initial population like in GA. But instead of performing a stochastic selection ES select the μ best individuals from the current population as parents for λ offspring. The offspring are often generated as mutants of the parents omitting the crossover operator. After the λ offspring are evaluated, the next generation is either $P(t+1) = P_c(t)$ in case of a (μ, λ)-strategy or $P(t+1) = P_c(t) \cup P(t)$ in case of a $(\mu + \lambda)$-strategy, see alg. 7 for details.

```
t = 0;
initialize(P(t=0));
evaluate(P(t=0));
**while** *isNotTerminated()* **do**
    $P_p$(t) = selectBest($\mu, P(t)$);
    $P_c$(t) = reproduce($\lambda, P_p(t)$);
    mutate($P_c$(t));
    evaluate($P_c$(t));
    **if** *(usePlusStrategy)* **then** P(t+1) = $P_c(t) \cup P(t)$;
    **else** P(t+1) = $P_c$(t);
    t = t +1;
**end**
```

Algorithm 7: General scheme of a $(\mu \overset{+}{,} \lambda)$-Evolution Strategy

2.5.1 Solution Representation

Evolution Strategies were designed for and are often limited to real-valued representations using a real-valued vector $\mathbf{X} = \langle x_1, x_2, ..., x_n \rangle$ as phenotype representing the decision variables. Any additional genotype is omitted in ES and all evolutionary operators act directly on the phenotype. But ES typically extend the individual by adding so called strategy parameters, which usually parameterize the mutation operators used in ES.
To code non real-valued decision variables again a mapping function is required. For example to optimize binary variables with ES real-valued decision parameters one could limit the range of the variables to $0 \leq x_i \leq 1$ and discretize the decision

parameters or interpret them as a vector of probabilities similar to the vector V in PBIL.

2.5.2 Selection

With ES typically a $(\mu \overset{+}{,} \lambda)$-strategy is used together with elite selection, which selects the μ best individuals to become parents to the next generation.

2.5.3 Recombination/Crossover

Although the crossover operator is of minor importance for ES, we will give some examples, even if they are usually applied in real-valued GAs. In the following examples $\mathbf{X}_1 = \langle x_1^1, ..., x_n^1 \rangle$ and $\mathbf{X}_2 = \langle x_1^2, ..., x_n^2 \rangle$ are the real-valued chromosomes of two parent individuals, while $\mathbf{X}_1'$ and $\mathbf{X}_2'$ will denote the two offspring generated from the parents.

- **Flat Crossover**: generates an offspring, by initializing the values of X' with uniform random values $x\text{'}_i^j = U(Min(x_i^1, x_i^2), Max(x_i^1, x_i^2))$ from the range given by the parents [34].

- **Discrete N-Point Crossover**: is equal to the mechanism used in bit-string crossover. N points $(\in \{1, 2, ..., n-1\})$ are selected, where the chromosomes of the parents are swapped to produce the offspring [48, 28].
  1-Point crossover:
$$\begin{aligned} \mathbf{X}'^1 &= (x_1^1, x_2^1, ..., x_i^1, x_{i+1}^2, ..., x_n^2) \\ \mathbf{X}'^2 &= (x_1^2, x_2^2, ..., x_i^2, x_{i+1}^1, ..., x_n^1) \end{aligned} \quad (2.7)$$

- **Intermediate Crossover**: calculates x_i' as the mean of x_i of all k parents, $x\text{'}_i^j = \frac{\sum_{j=0}^k x_i^j}{k}$.

- **Arithmetical Crossover** uses a linear combination of x_i of all k parents to set $x\text{'}_i^j = \sum_{j=0}^k \alpha_j \cdot x_i^j$. The linear factor α_i is a unique random variable with $\sum_{j=0}^k \alpha_j = 1$ and $\alpha_j > 0 \; \forall \; j$ [28].

- **BLX-α Crossover**: similar to the flat crossover the BLX-α crossover operator [10] initializes X' with values from an extended range given by the parents, $x\text{'}_i^j = U(x_{i,min} - I \cdot \alpha, x_{i,max} + I \cdot \alpha)$ with $x_{i,min} = Min(x_i^1, x_i^2)$, $x_{i,max} = Max(x_i^1, x_i^2)$ and $I = x_{i,max} - x_{i,min}$. Actually the BLX-0.0 crossover is equal to the flat crossover.

- **Discrete Uniform Crossover**: here the value of x_i' is chosen randomly from the value set $\{x_i^1, x_i^2\}$ given by the parents [30]. This crossover operator is related to the uniform crossover in binary GAs.

If ES are used together with self-adapting mutation operators, the crossover operators also need to deal with the additional strategy parameters. In that case typically a discrete crossover is applied to the decision variables **X** and an intermediate crossover is applied to the strategy parameters. But the more sophisticated the self-adaption strategy of the mutation operator, the less efficient and even disruptive is the crossover operator on the strategy parameters. Therefore, in some cases the crossover operator is omitted completely from the ES.

2.5.4 Mutation

Mutation is considered to be the main and sometimes even the only evolutionary operator in ES. Since ES restrict them selfs to real-valued search spaces very sophisticated mutation operators have been developed in the recent years.

- **Standard mutation** uses no additional strategy parameters $\langle x_1, x_2, ..., x_n \rangle$ and simply adds a random number to a randomly selected element x_i of the decision parameters.

- **Global mutation** uses a single strategy parameter and extends the representation to $\langle x_1, x_2, ..., x_n, \sigma \rangle$ and uses the strategy parameter σ to control the mutation step size:

$$\begin{aligned} \sigma' &= \sigma \cdot e^{\tau \cdot N(0,1)} \\ x_i' &= x_i + \sigma \cdot N_i(0,1) \end{aligned} \tag{2.8}$$

  where $\tau \propto 1/\sqrt{n}$ gives a learning rate, σ is bounded by a minimum value σ_{min} and $N(0,1)$ is a normally distributed random number with mean zero and standard deviation one. See in fig. 2.8 how different values of σ can be beneficial depending how far away the individual is from the optimum. Please note that suitable values of σ are not selected directly, but indirectly depending on the impact of σ on the offspring of a given individual.

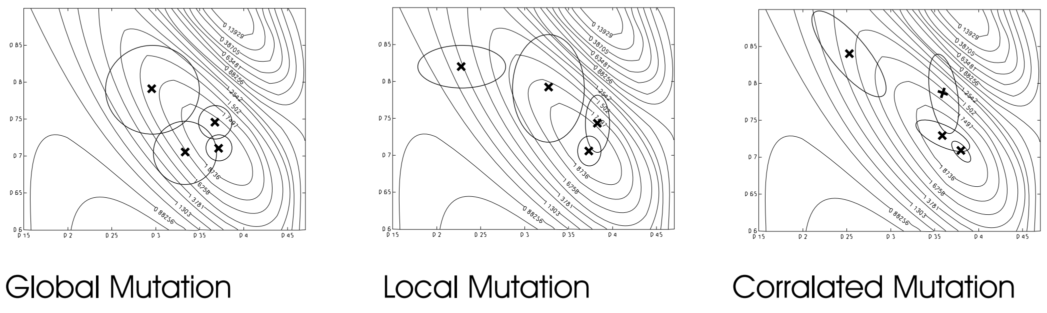


Figure 2.8: Three different mutation operators

- **Local mutation** adds a complete vector of strategy parameters to the representation $\langle x_1, x_2, ..., x_n, \sigma_1, ..., \sigma_n \rangle$ one for each decision variable. This allows to adapt the strategy parameters independently for each dimension of the problem space and to have different mutation step sizes for each variable:

$$\begin{aligned} \sigma'_i =& \quad \sigma_i \cdot e^{\tau' \cdot N(0,1) + \tau \cdot N_i(0,1)} \\ x'_i =& \quad x_i + \sigma_i \cdot N_i(0,1) \end{aligned} \tag{2.9}$$

  where $\tau' \propto 1/\sqrt{2n}$ gives an overall learning rate and $\tau \propto 1/\sqrt{2\sqrt{n}}$ gives a coordinate-wise learning rate. Again the values for σ_i have a lower bound of σ_{min}. See in fig. 2.8 how the σ_i's can be adopted to fit the local properties of the search space.

- **Correlated mutation** allows to adapt the orientation of the mutation ellipsoid introduced in the local mutation method by further adding orientation angles to the ES representation $\langle x_1, x_2, ..., x_n, \sigma_1, ..., \sigma_n, \alpha_1, ..., \alpha_k \rangle$ with $k = n \cdot (n-1)/2$. This mutation method uses an additional covariance matrix C which is defined as $c_{ii} = \sigma_i^2$, $c_{ij} = 0$ if i and j are not correlated and $c_{ij} = 1/2 \cdot (\sigma_i^2 - \sigma_j^2) \cdot tan(2\alpha_{ij})$ if i and j are correlated. The mutation mechanism follows as:

$$\begin{aligned} \sigma'_i =& \quad \sigma_i \cdot e^{\tau' \cdot N(0,1) + \tau \cdot N_i(0,1)} \\ \alpha'_j =& \quad \alpha_j + \beta \cdot N_j(0,1) \\ \mathbf{X}'_i =& \quad \mathbf{X}_i + \mathbf{N}(0, C) \end{aligned} \tag{2.10}$$

  again $\tau' \propto 1/\sqrt{2n}$ and $\tau \propto 1/\sqrt{2\sqrt{n}}$ while $\beta \approx 5°$. Again the σ_i have a lower bound and if $|\alpha'_j| > \pi \Rightarrow \alpha'_j = \alpha'_j - 2\pi sign(\alpha'_j)$. See fig. 2.8 for an example how the direction of the mutation can be adopted to fit the local properties of the fitness landscape.

Even more sophisticated mutation strategies are the Covariance Matrix Adaptation (CMA) mutation operator [16] and the Main Vector Adaptation (MVA) mutation operator [33].

2.5.5 Generation Strategies

ES typically employ a generational strategy using the $(\mu \overset{+}{,} \lambda)$-strategy. But an extension allows to interpolate between the $(\mu + \lambda)$-strategy and a (μ, λ)-strategy. In the (μ, λ, κ)-strategy κ gives the lifetime of an individual. After an individual participated in κ generational cycles it dies and cannot survive into the next generation anymore. With $\kappa = 1$ this extended strategy equals the (μ, λ)-strategy and with $\kappa = \infty$ it equals the $(\mu + \lambda)$-strategy.

2.6 Model Assisted Evolution Strategies

Evolutionary Algorithms are usually known to require many target function evaluations. In some real-world optimization problems this is not practical, especially if target function evaluations are expensive. This can happen if the target function requires extensive simulations, which causes high computational effort, or if the target function requires real-world experiments, which are time consuming and expensive. To counter the high demand of target function evaluations of EAs a surrogate model can be used to approximate the true target function. Unfortunately, the model limits the data types to be processed often to real-valued search spaces. Therefore, this strategy is usually used in combination with ES and is called Model Assisted Evolution Strategies (MAES) [7, 8, 19, 45].

```
t = 0;
initialize(P(t=0));
**evaluateOnTrueTarget**(P(t=0));
trainModel(P(t=0));
**while** *isNotTerminated()* **do**
    $P_p$(t) = selectBest($\mu, P(t)$);
    $P_{\lambda_+}$(t) = reproduce($\lambda_+, P_p(t)$);
    mutate($P_{\lambda_+}$(t));
    **evaluateOnModel**($P_{\lambda_+}$(t));
    $P_c$(t) = selectBest($\lambda, P_{\lambda_+}(t)$);
    **evaluateOnTrueTarget**($P_c(t)$);
    updateModel($P_c(t)$);
    **if** *(usePlusStrategy)* **then** P(t+1) = $P_c(t) \cup P_p(t)$;
    **else** P(t+1) = $P_c$(t);
    t = t +1;
**end**
```

Algorithm 8: General scheme of a ($\mu \overset{+}{,} \lambda$)-Model Assisted Evolution Strategy

A general scheme of the strategy of model assisted evolution is given in alg. 8. In the generational cycle an intermediate population of offspring $\lambda_+ > \lambda$ is generated and evaluated on the local model of the target function, see fig. 2.9. This model was initialized and trained using the initial population, which was evaluated on the true target function. Then the λ best individuals are selected from the intermediate population of λ_+ offspring using the output of the model as surrogate for the true fitness function. This assumes that those individuals performing best on the surrogate function will also perform best on the true fitness function. The selected λ individuals are then evaluated on the true fitness function without any further alterations. The new samples of the true target function are then used to update the model for the next generation cycle.

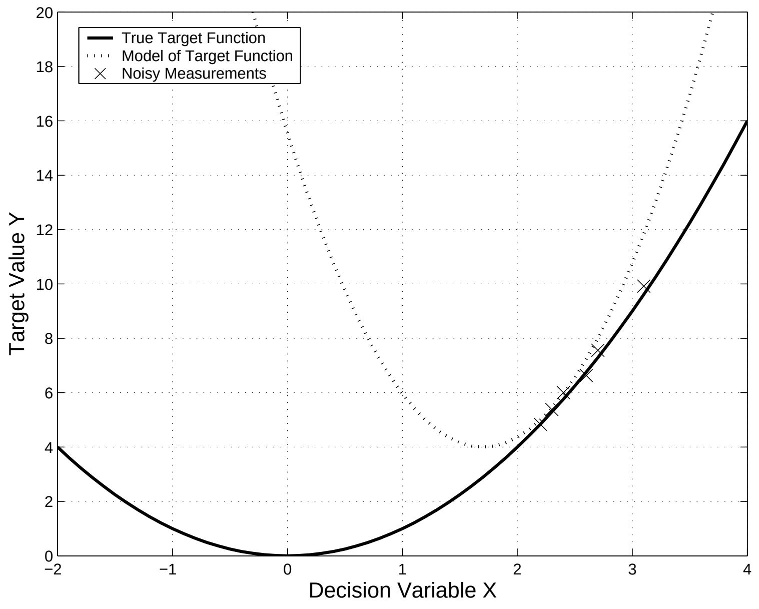


Figure 2.9: Modeling the true target function from evaluated individuals to save expensive fitness function evaluation

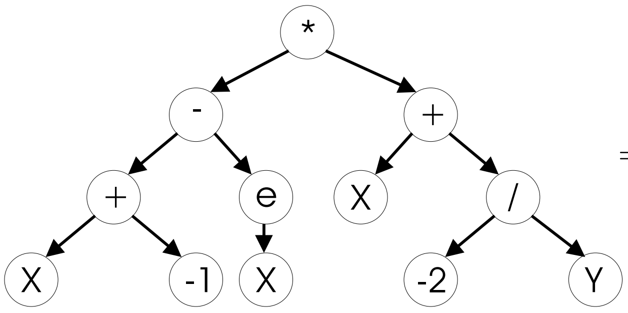
There are multiple alternative implementations of MAES, which may utilize the estimated prediction error of the model to determine whether or not an individual is to be evaluated with the surrogate function or the true target function. Others implement more sophisticated strategies to search for suitable samples to improve the model instead of focusing on increasing the fitness only.
Anyway, it has to be noted that sometimes the MAES is limited by the performance of the model rather than the Evolution Strategies. Depending on the model used either the dimension of the search space or the number of training samples can become the limiting element that increases the necessary computational effort to calculate or update the model. In some instances this effort might even exceed the effort for the true fitness function. Therefore, it is necessary, for example, to limit the number of samples to train the model to a local selection.
Besides the additional model everything from selection methods, generational strategies, representations used and crossover and mutation operators remains the same as in ES.

2.7 Genetic Programming

Although currently John Koza is dominating the literature about Genetic Programming (GP) and actually coined the term GP [22, 23, 24], the history of GP started much earlier. For example Cramer and Fujiki began in the mid eighties to work on automated programming [4, 13], while Friedberg started even earlier evolving machine language programs [11, 12].
GP adds a new data type to EAs to represent and optimize general computer programs with EAs. With GP it is possible to evolve program code to solve a given programming problem, to evolve a mathematical function for symbolic regression or to develop electrical circuits in Evolvable Hardware. Although this extension is straightforward, the results are usually not as promising as expected. This is due to the fact that the search space is extremely large and non-causal, i.e. solutions that are similar in genotype space are not necessarily similar in phenotype space. Especially the property of non-causality makes these kinds of optimization problems extremely hard to solve. Nonetheless, GP is the only optimization method that is able to deal with these kind of problem spaces, despite all its drawbacks.
Basically, the standard Koza style GP is quite similar to the standard GA except for the new solution representation. Therefore, we will not give the algorithm again but simply refer to the algorithm given previously, see alg. 4.

2.7.1 Solution Representation

The traditional Koza style GP uses a tree based representation for programs, which comes from the initial experiments using LISP, see fig. 2.10. This representation uses a directed acyclic graph with functions as nodes and terminals as leafs. The execution order is given by evaluating the left child node before the right node resulting in the equation given in equ. 2.11.
Depending on the choice of functions and terminals, the so called area, one can solve different kinds of optimization problems. In case of symbolic regression, for example, functions are typically mathematical functions like $\{e^x, sin(x), cos(x), (x+$



$$\Rightarrow (x - 1 - e^x) \cdot (x + \frac{-2}{y}) \tag{2.11}$$

Figure 2.10: The tree based GP geno- and phenotype

$y), (x-y), (x \cdot y), (x/y), ...\}$ while the terminals are either variables $\{x, x, z, ..\}$ or random constants $\{c_0, c_1, ..., c_n\}$. In case of programming tasks the functions are methods $\{if\{\}()(), execute()(), store(), ...\}$ while the terminals are either input values or simple execution tasks, which require no additional parameters.
The area of selected function and terminals typically has to meet two basic requirements to be applicable to GP:

- **Closure**: every function node must be able to process every possible output of all nodes in the area. For example the division operator needs to deal with zero as denominator without terminating the whole program. This problem can be resolved using a secure division operator '%' that returns one in case of zero as denominator.

- **Sufficiency**: the area must be sufficient to solve the problem, e.g. the area needs to contain a 'drive' command to control a mobile robot, otherwise the program cannot move the robot.

Some extensions of GP have softened the closure property by defining a strongly typed GP using nodes with flavor. In that case certain successive nodes may request a special data type, e.g. the if{} operator may request a boolean input as condition. The strongly typed GP requires special precautions for initialization, crossover and mutation but basically behaves like a standard GP in every other aspect.
A very common problem in GP is bloat. Bloat means that the program trees grow significantly in size while the fitness is stagnating. This leads to increased computation time due to the increased evaluation effort for each program tree. To counter bloat the overall tree size of a GP program tree is often limited.

Initialization

To initialize a GP program tree one usually starts with a random node as root node and recursively adds new random nodes to the successive nodes until every node that requires a successor has valid successor nodes. This recursive approach eventually terminates when a terminal is added as node and this way the recursive call is interrupted for that branch of the tree. There are two general methods how to generate a random GP program tree:

- **Full**: this method allows one to set a target depth d of the new tree and initializes evenly balanced trees. This is done by selecting only non terminals as following nodes as long as the current depth of the node is smaller than d and selecting only terminal nodes otherwise.

- **Grow**: this initialization method uses the recursive approach initially discussed. Unfortunately, this method may grow infinitely large trees if the ratio between terminals and non-terminals is bad or it may also generate extremely short program trees. Therefore, often an upper bound for the tree size is introduced, which causes the recursive growth method to select terminals only if violated, to limit the overall size of the GP program tree.

- **Ramped Half and Half**: this approach mixes the two previously mentioned methods to generate a diverse set of program trees of different sizes. The ramped half and half method alternately calls full and growth initialization while continuously increasing the upper bound for the tree size from the smallest three node program trees to the biggest program trees of d_{max} nodes, where d_{max} is to be specified by the user depending on the problem type.

2.7.2 Selection

The selection methods are basically the same as in GA, but typically Tournament Selection is used to impose a stronger selection pressure on the GP population, while at the same time allowing a diverse population.

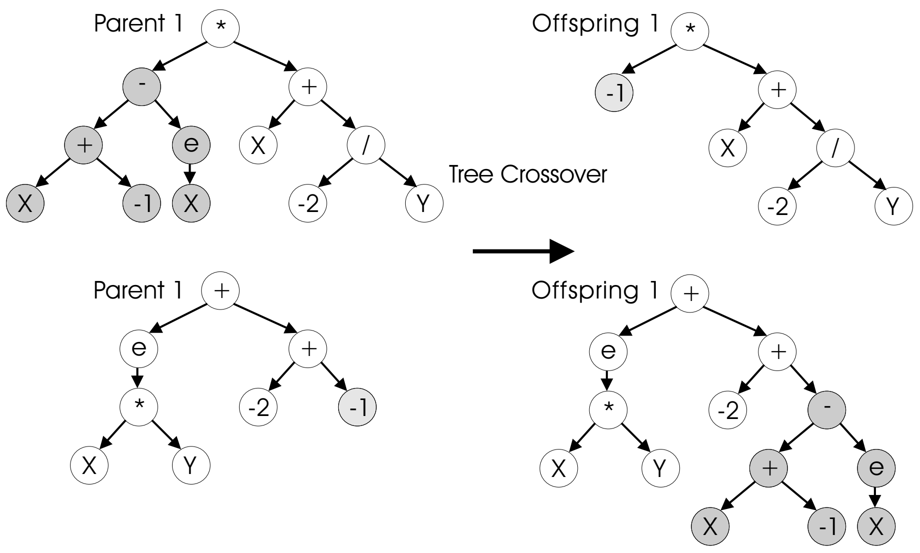


Figure 2.11: An exemplary GP crossover between two parents

2.7.3 Recombination/Crossover

Recombination on GP simply selects two random nodes from each parent and exchanges the associated subtrees to generate new offspring, see fig. 2.11. In case some nodes accept only special output types it is necessary to ensure that the resulting offspring are still able to evaluate correctly.

There are also special extensions to GP crossover to counter the effect of bloat, since crossover is able to generate extremely large GP program trees by exchanging a leaf node of one parent with a node close to the root node in the other parent, see for example fig. 2.11. One method uses pruning to replace all nodes that exceed the given maximum depth of the program tree d_{max} with terminal nodes. Depth-preserving crossover only exchanges nodes of equal depth between two parents and depth-dependent crossover selects nodes for crossover depending on the depth of the nodes to counter the effect of dominance of leaf nodes in case of random selection. Although GP crossover is said to be an important operator in GP and should act similar to the crossover operator in GAs combining building-blocks to find a good solution, usually the crossover probability in GP is not as high as in GAs.

2.7.4 Mutation

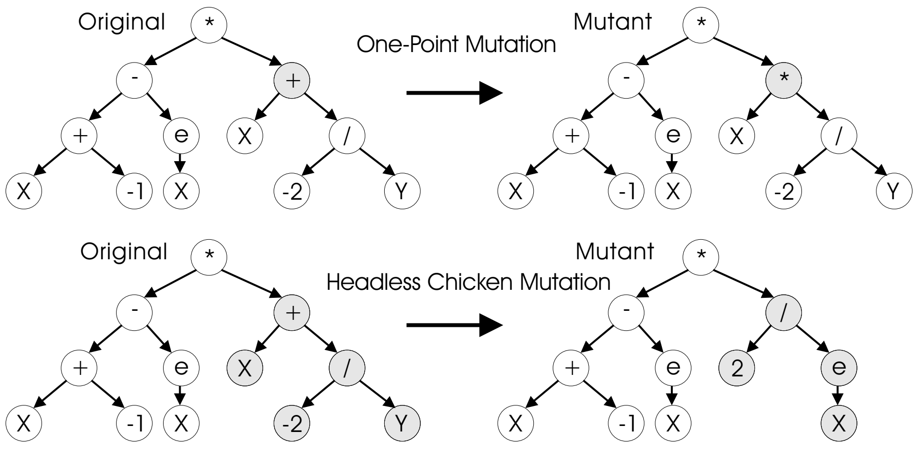


Figure 2.12: Two types of GP mutation

Mutation in GP plays a major role in exploring the search space and multiple alternatives have been suggested to perform mutation on a GP genotype. We will only outline two of the simplest here:

- **One-Point Mutation**: selects a random node in the GP program tree and replaces this node with a node of equal arity while the successive nodes remain unchanged, see upper part fig. 2.12. This mutation method is said to allow a gradual change in the GP genotype.
- **Headless Chicken Mutation**: this mutation method also selects a random node in the GP program tree but replaces the node and all successive nodes with a randomly generated subtree using the grow method, see lower part fig. 2.12. This mutation operator is said to be equally efficient as the GP

crossover methods mentioned previously. But with this mutation still special precautions are necessary to counter bloat, e.g. by limiting the size of the generated subtree.

2.7.5 Generation Strategies

Again the generation strategies are the same as discussed previously in the section about GA, see sec. 2.2. In GP typically a generational or steady-state strategy is used. The steady-state strategy is said to provide an improved convergence behavior for GP.

Chapter 3

JavaEvA - Tutorial

This tutorial on JavaEvA describes how to use the JavaEvA optimization toolbox at the level of the provided GUI elements. They give access to the parameters and operators of the optimization algorithms and the benchmark optimization problems already implemented in JavaEvA. If one wants to implement ones own optimization problem and solve it using JavaEvA, please refer to chapter 4 for a detailed explanation.
First, we give a general introduction on how to start the JavaEvA GUI and how to select and activate an optimization module. Then, the following chapters give more details on the parameters and the options of the individual optimization strategies available in the JavaEvA optimization toolbox.
To download the necessary files or to access the Java WebStart application visit our web pages at http://www-ra.informatik.uni-tuebingen.de/software/JavaEvA/.

3.1 First Steps with JavaEvA

In this section we describe how to start JavaEvA, how to select and parameterize an optimization algorithm and the statistics performed on the optimization process, and finally how to select and parameterize an optimization problem.

3.1.1 Getting Started

To start JavaEvA, there are multiple alternatives ranging from a Java WebStart application to a complete project environment, which allows one to define ones own optimization problems, as described in chapter 4.

- **Using the Java WebStart Application**: in case a Java Virtual Machine is already installed and activated, there is a link to the WebStart application in the download section of the JavaEvA web pages. Otherwise, there is a link

to a web page that describes how to install and activate the Java Virtual Machine. Additionally, there is another link that gives more details on the Java WebStart technology.
A simple click on the link to the Java WebStart application is sufficient to start JavaEvA. Unfortunately, the Java WebStart environment is rather restrictive regarding access to computer resources. Therefore, some options, like the UserDefinedProblem and loading problem data for the traveling salesman and the portfolio selection problem are not available using the Java WebStart application.

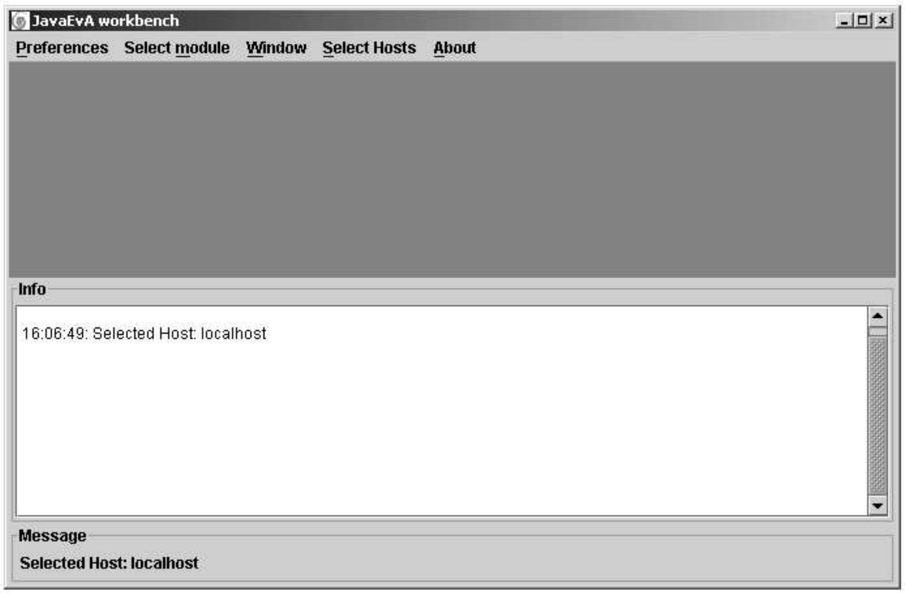
- **Download and start the JavaEvA.jar**: To use the JavaEvA.jar
  1. Download the JavaEvA.jar to the hard disk.
  2. Start JavaEvA either by
     (a) double clicking the JavaEvA.jar in a file explorer using 'java' or 'javaw' as target.
     (b) by using the console command 'java -jar JavaEvA.jar' or 'java -cp JavaEvA javaeva.client.EvAClient' if a suitable display is defined and available to show the Java GUI elements.

  Typically, JavaEvA will uncompress some data files into the same folder where the JavaEvA.jar was started. This is necessary to allow easy access to problem data files and images.

- **Download and use the JavaEvAExample project**:
  1. Download and uncompress the JavaEvAExample.zip from our web page to the hard disk.
  2. Open a new Java project with a Java IDE including the /src folder as sources and JavaEvA.jar as library.
  3. One can start JavaEvA by
     (a) executing the main method from JOptExampleGUI.java.
     (b) using the ant target 'run' in case the IDE supports ant.

Regardless, what method one uses, the very first GUI element to show up is the JavaEvA workbench, also known as the EvAClient, see fig. 3.1.
There are multiple drop-down menus for the EvAClient:

- **Preferences**: to choose from three different GUI styles.
- **Select Module**: to select the available optimization modules, ranging from random search to the most recently developed Model-Assisted Evolution Strategies.



16:06:49: Selected Host: localhost

Selected Host: localhost

Figure 3.1: The JavaEvA main GUI frame, called EvAClient, directly after starting JavaEvA.

- **Window**: due to the generic object editor the number of windows that are on the screen at the same time can become quite numerous. To navigate between the open windows, one can choose from this list which window should be on top.
- **Select Host**: JavaEvA was implemented with a client/server structure to allow optimization of computationally expensive optimization problems. Currently, one sees the GUI element of the local JavaEvA client that runs on ones own machine, while the optimization processes are typically running on the JavaEvA server, which may run on a different machine.
  To solve computationally expensive optimization tasks it is advisable to run the JavaEvA server not on ones own PC, but on a more powerful machine. The name of the machine, where the actual JavaEvA server application is located, is displayed in the upper message box of the main frame. It defaults to the local machine if no other server is available.
  Currently, JavaEvA servers need to be started manually:
  1. To make the remote machine known to the JavaEvA client, include the name of the machine in the /resources/JProxyServerList.props file on the local PC.
  2. To start a JavaEvA server use the console command 'java -cp JavaEvA.jar javaeva.server.EvAServer' on the remote machine. The JavaEvA server

runs without any GUI elements.

3. Now one can select JavaEvA server on the remote machine as host.

This procedure may seem difficult, but typically JavaEvA defaults to a local JavaEvA server, if no host is given. Currently we are testing an automated mechanism to start a server on a remote machine, which will be available soon.

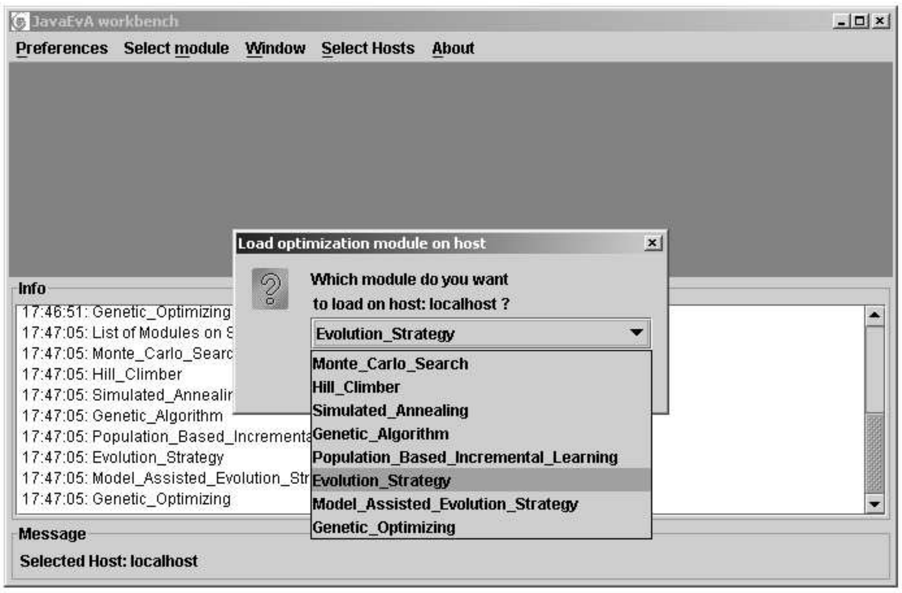
3.1.2 Select an Optimization Module

To select an optimization module activate the 'Load Module' menu item. Here one can choose between multiple optimization modules, which are available on the JavaEvA server:

- **Monte Carlo Simulation**: for details see 3.2.
- **Hill Climber**: for details see 3.3.
- **Simulated Annealing**: for details see 3.4.
- **Population Based Incremental Learning**: for details see 3.6.
- **Genetic Algorithm and Genetic Programming**: for details see 3.5.
- **Evolution Strategy**: for details see 3.7.
- **Model Assisted Evolution Strategy**: for details see 3.8.
- **Genetic Optimization**: for details see 3.9.

For this tutorial select the Evolution Strategy from the available optimization modules, see fig. 3.2.
After one has selected the Evolution Strategy, a new window opens for the selected optimization module, see fig. 3.3. It is divided into three elements: the control buttons at the top of the frame and a tabbed panel with an element for the module parameters and an element for the statistics parameters. The control buttons are:

- **Start Optimization**: for starting the optimization process.
- **Stop**: to interrupt the optimization.
- **Restart Optimization**: to resume a previously stopped optimization process.



Which module do you want
to load on host: localhost ?

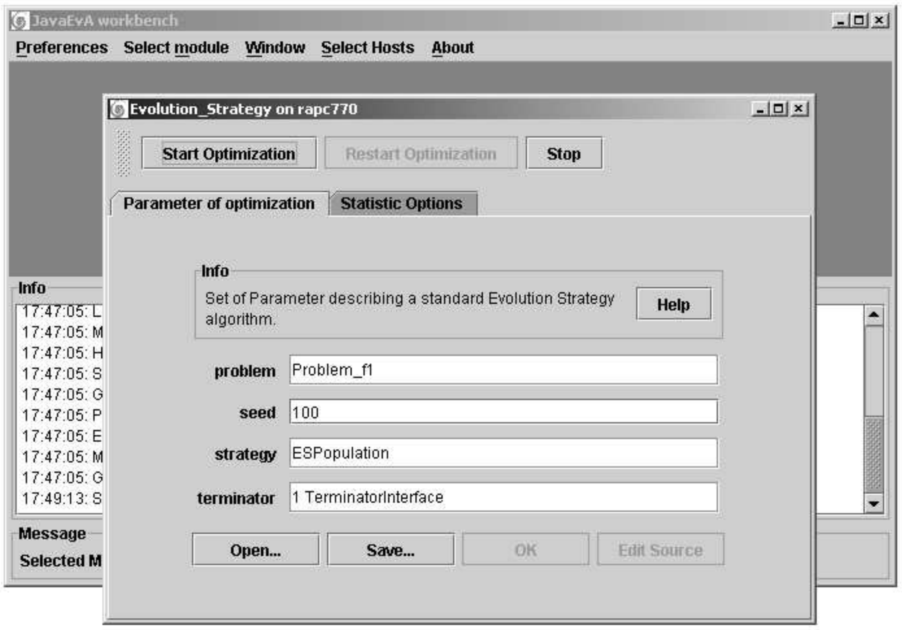
Evolution_Strategy

Monte_Carlo_Search
Hill_Climber
Simulated_Annealing
Genetic_Algorithm
Population_Based_Incremental_Learning
Evolution_Strategy
Model_Assisted_Evolution_Strategy
Genetic_Optimizing

```
17:46:51: Genetic_Optimizing
17:47:05: List of Modules on S
17:47:05: Monte_Carlo_Searc
17:47:05: Hill_Climber
17:47:05: Simulated_Annealir
17:47:05: Genetic_Algorithm
17:47:05: Population_Based_Incrementa
17:47:05: Evolution_Strategy
17:47:05: Model_Assisted_Evolution_Str
17:47:05: Genetic_Optimizing
```

Selected Host: localhost

Figure 3.2: Selecting the Evolution Strategy from the available optimization modules.



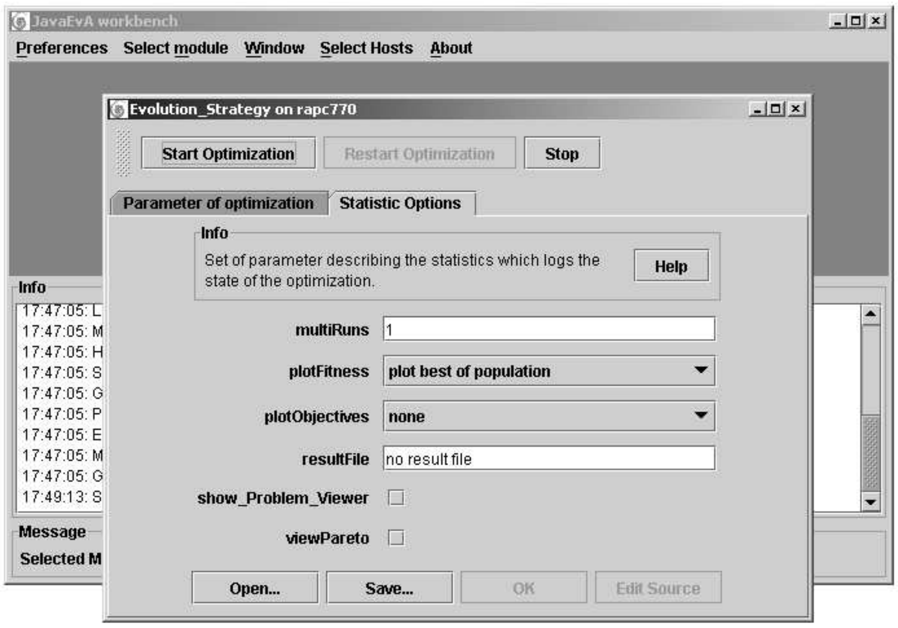
Set of Parameter describing a standard Evolution Strategy algorithm.

problem	Problem_f1
seed	100
strategy	ESPopulation
terminator	1 TerminatorInterface

```
17:47:05: L
17:47:05: M
17:47:05: H
17:47:05: S
17:47:05: G
17:47:05: P
17:47:05: E
17:47:05: M
17:47:05: G
17:49:13: S
```

Selected M

Figure 3.3: The GUI element for the optimization module selected Evolution Strategy with the optimization parameters.



Set of parameter describing the statistics which logs the state of the optimization.

```
17:47:05: L
17:47:05: M
17:47:05: H
17:47:05: S
17:47:05: G
17:47:05: P
17:47:05: E
17:47:05: M
17:47:05: G
17:49:13: S
```

multiRuns	1
plotFitness	plot best of population
plotObjectives	none
resultFile	no result file
show_Problem_Viewer	☐
viewPareto	☐

Figure 3.4: The GUI element for the optimization module selected with the general statistics options currently selected.

Regarding the tabbed panel, the individual parameters for the different optimization algorithms are described in the following subsections. The optimization parameters define the optimization problem to solve and the parameters of the optimization strategy. As the optimization parameters are specific for each module, they are described in detail in sec. 3.2, 3.3, 3.4, 3.6, 3.5, 3.7, 3.8 and 3.9.

The Statistics Options shown in fig. 3.4 allow to parameterize the statistics performed on the optimization runs, like the number of multi runs, what data to plot and where to store the results of the optimization process:

- **Multi Runs**: as most optimization strategies implemented in JavaEvA are non-deterministic, it is often necessary to perform multiple runs for each optimization strategy and take the mean of the obtained results to allow the comparison of the performance for each strategy.

- **Plot Fitness**: choose what kind of data is to be plotted over the number of fitness evaluations. The alternatives are to plot the best fitness, worst fitness or both best and worst fitness. Typically, only the best fitness values achieved are averaged over multiple runs to evaluate the performance of an optimization algorithm.

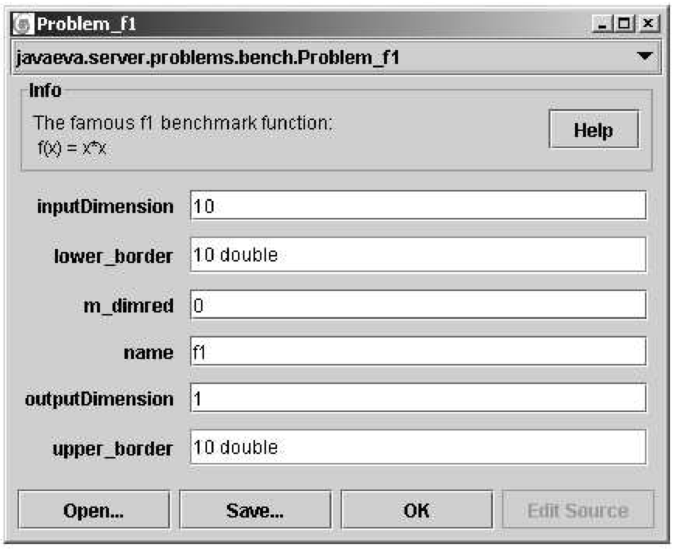
- **Plot Objectives**: this adds an additional plot window that plots the objective values of the currently best solution over the number of fitness evaluations.

Unfortunately, this option is only available for real-valued search spaces and the plots of the objective values may become confusing if used in a multi run environment.

- **Result File**: allows to specify the name of an output file, where the raw data of the optimization process is to be stored. This allows one to calculate ones own statistics on the performance of the optimization algorithms, if this is required for a particular application problem.
- **Show Problem Viewer**: this item is currently under revision.
- **View Pareto-Front**: this item is currently under revision.

Some of the elements are currently under revision, because we believe that at least some options belong to the particular optimization problem instead of the optimization algorithm. We consider it to be more appropriate to let the user decide what kind of data he wants to plot depending on his application problem or what kind of data to store.

3.1.3 Select an Optimization Problem



javaeva.server.problems.bench.Problem_f1

Info
The famous f1 benchmark function:
f(x) = x*x

inputDimension	10
lower_border	10 double
m_dimred	0
name	f1
outputDimension	1
upper_border	10 double

Figure 3.5: The GUI element for the optimization problem.

One property each optimization algorithm provides is the choice of the optimization problem, see fig. 3.3. Click this element to open the optimization problem GUI element, see fig. 3.5. On top of the GUI element there is a choice element, which

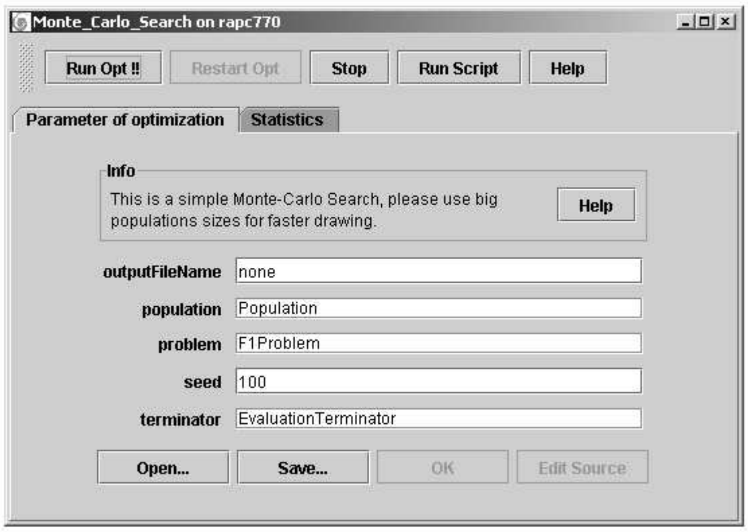
allows one to select from multiple alternative optimization problems available. This choice element is typical for the GUI elements in JavaEvA. In case alternative implementations of a given operator or method are available, this choice element allows one to choose from the available elements.
Below the choice element there is a short description of the currently selected class and a link to an additional help page if available. Then, there is a list of editable properties of the given class. In case of the real-valued F1-function these parameters are limited to:

- **Input Dimension**: allows to set the dimensionality of the optimization problem.
- **Output Dimension**: sometimes one may also be able to choose the output dimension of the problem.
- **Lower/Upper Border**: in case of real-valued optimization problem one may choose the allowed range for the decision variables.

Typically, every editable property is described in a small tool tip. To see the tool tip on a specific property, hold the mouse pointer over the property for several seconds. Further descriptions on selected problems can be found in the following sections.

3.2 Tutorial on Monte-Carlo Search

As discussed before the Monte-Carlo Search is a blind optimization strategy, but it still may give insights into the structure and the complexity of a given optimization problem. Further it may be used as worst case reference approach to compare any EA approach to.



This is a simple Monte-Carlo Search, please use big populations sizes for faster drawing.

outputFileName: none
population: Population
problem: F1Problem
seed: 100
terminator: EvaluationTerminator

Figure 3.6: The Monte-Carlo GUI frame

3.2.1 User's Description

When selecting the Monte-Carlo Search one has only few parameters to specify, see fig. 3.6:

- **Output File Name**: gives the name of the output file where the optimization results are stored. Since this optimizer belongs to the Genetic Optimization family, the basic data output is currently best, mean and worst fitness per generation. Any additional data needs to be specified in the problem implementation and is therefore problem dependent.

- **Population**: here one can specify the population size of the optimization algorithm. In case of the Monte-Carlo search this has no impact on the optimization performance of the algorithm, but may increase speed of the fitness plots, because it draws one line per generation, which would equal the number of fitness evaluations in case of a population size of one.

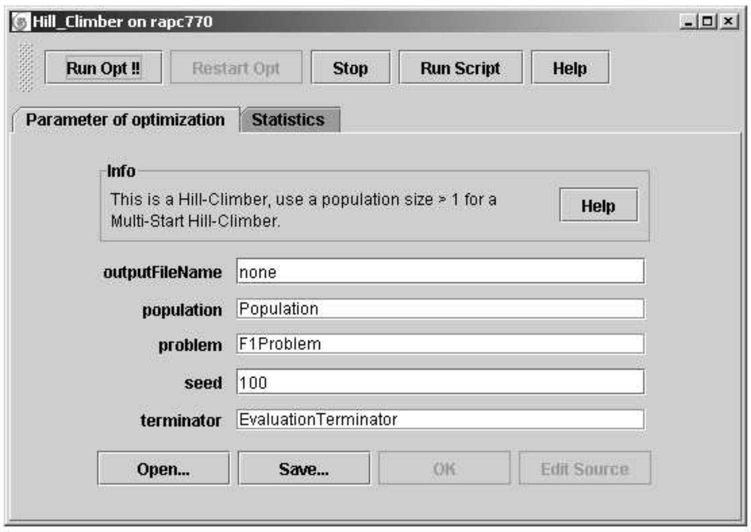
- **Problem**: here one can specify the problem to optimize and also the representation, initialization method, mutation and crossover rates and methods. Since the Monte-Carlo search solely relies on initializing random solutions, the representation and the initialization mechanism is the only element that has an impact on the performance of this optimization approach.
  Please refer to sec. 3.9 for further details and examples on problem implementations.

- **Seed**: specifies the starting value for the random number generator, which allows reproducible experiments in case of stochastic optimization strategies. Set the seed to zero to use a random starting condition.

- **Terminator**: gives the termination criterion. In the Genetic Optimization framework currently the termination criterion is limited to choosing the maximal number of fitness evaluations, the maximum number of iterations or a target fitness value.

3.3 Tutorial on Hill-Climber

As discussed before the Hill-Climber Search is a greedy local search method, which is prone to premature convergence in multi-modal search spaces. Therefore, it is necessary in multi-modal search space to use a multi-start approach by setting a population size >> 1. Since the Hill-Climber relies on mutation, it is necessary to use the highest possible mutation rate $p_m = 1.0$ for the individuals and to choose an appropriate mutation operator depending on the problem and the representation used.



This is a Hill-Climber, use a population size > 1 for a Multi-Start Hill-Climber.

outputFileName	none
population	Population
problem	F1Problem
seed	100
terminator	EvaluationTerminator

Figure 3.7: The Hill-Climber GUI frame

3.3.1 User's Description

When selecting the Hill-Climber Search one has only few parameters to specify, see fig. 3.7:

- **Output File Name**: gives the name of the output file where the optimization results are stored. Since this optimizer belongs to the Genetic Optimization family, the basic data output is currently best, mean and worst fitness per generation. Any additional data needs to be specified in the problem implementation and is therefore problem dependent.

- **Population**: here one can specify the population size of the optimization algorithm. In case of the Hill-Climber search the population size gives the number of multi-start hill-climbers to use in case of multi-modal search spaces.

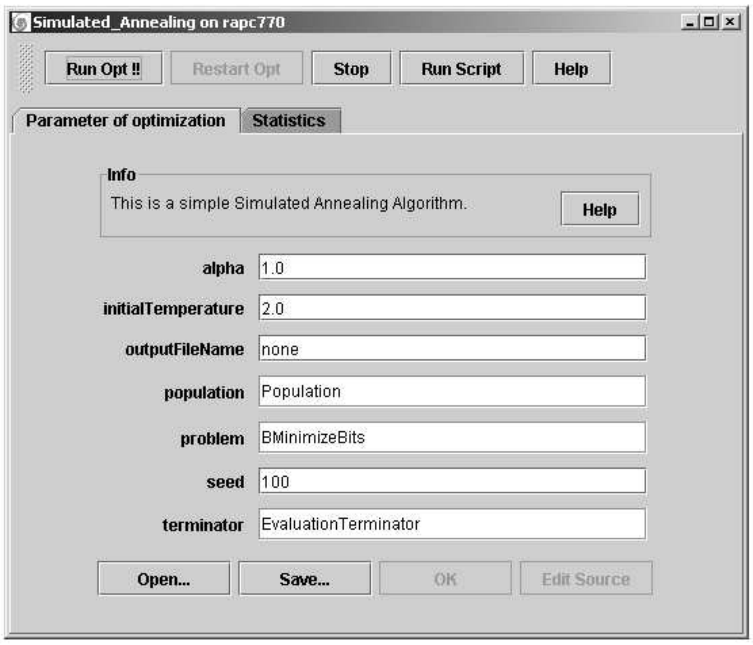
- **Problem**: here one can specify the problem to optimize and also the representation, initialization method, mutation and crossover rates and methods. Since the Hill-Climber relies only on the mutation operator, the representation and the choice of the mutation operator has a significant impact on the performance of the Hill-Climber. Again it is necessary to set the mutation probability p_m to the maximum value of $p_m = 1.0$.
  Please refer to sec. 3.9 for further details and examples on problem implementations.

- **Seed**: specifies the starting value for the random number generator, which allows reproducible experiments in case of stochastic optimization strategies. Set the seed to zero to use a random starting condition.

- **Terminator**: gives the termination criterion. In the Genetic Optimization framework currently the termination criterion is limited to choosing the maximal number of fitness evaluations, the maximum number of iterations or a target fitness value.

3.4 Tutorial on Simulated Annealing

Simulated Annealing is similar to the Hill-Climber, but since the replacement scheme is not as strict Simulated Annealing is not as prone to premature convergence as the Hill-Climber. Therefore, the population size is not such a critical parameter. But still the mutation operator needs to be selected carefully.
Compared to the Hill-Climber additional parameters are necessary for the Simulated Annealing approach to parameterize the cooling schedule.



This is a simple Simulated Annealing Algorithm.

alpha	1.0
initialTemperature	2.0
outputFileName	none
population	Population
problem	BMinimizeBits
seed	100
terminator	EvaluationTerminator

Figure 3.8: The Simulated Annealing GUI frame

3.4.1 User's Description

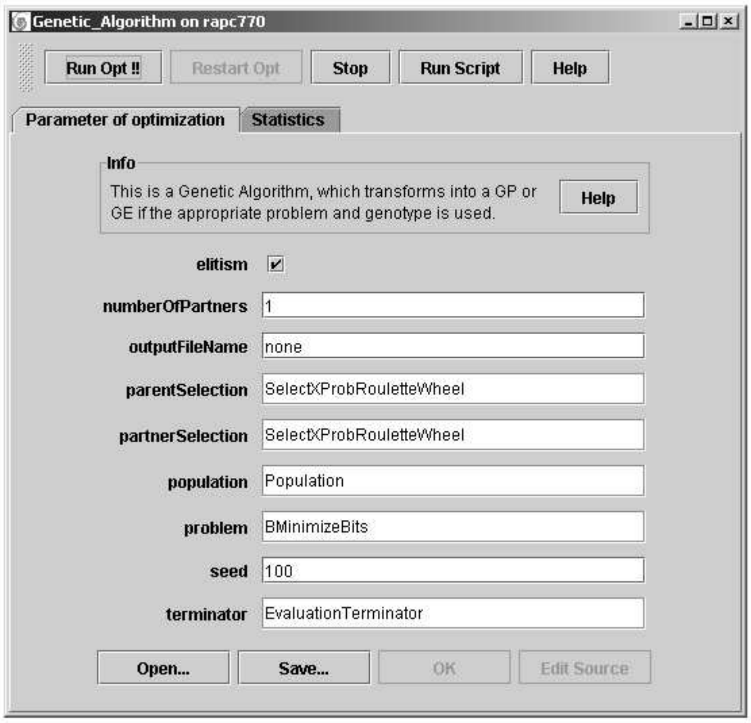
When selecting the Simulated Annealing strategy one has only few parameters to specify besides the parameters the give the cooling schedule, see fig. 3.8:

- **Output File Name**: gives the name of the output file where the optimization results are stored. Since this optimizer belongs to the Genetic Optimization family, the basic data output is currently best, mean and worst fitness per generation. Any additional data needs to be specified in the problem implementation and is therefore problem dependent.
- **Population**: here one can specify the population size of the optimization algorithm. In case of Simulated Annealing this parameter is not as critical as in case of the Hill-Climber.

- **Problem**: here one can specify the problem to optimize and also the representation, initialization method, mutation and crossover rates and methods. Since Simulated Annealing relies only on the mutation operator, the representation and the choice of the mutation operator has a significant impact on the Simulated Annealing algorithm. Further it is necessary to set the mutation probability p_m to the maximum value of $p_m = 1.0$.
  Please refer to sec. 3.9 for further details and examples on problem implementations.
- **Seed**: specifies the starting value for the random number generator, which allows reproducible experiments in case of stochastic optimization strategies. Set the seed to zero to use a random starting condition.
- **Terminator**: gives the termination criterion. In the Genetic Optimization framework currently the termination criterion is limited to choosing the maximal number of fitness evaluations, the maximum number of iterations or a target fitness value.
- **Initial Temperature**: gives the starting temperature T for the algorithm, compare alg. 3.
- **Alpha**: gives a linear cooling rate for the Simulated Annealing algorithm, see also alg. 3.

3.5 Tutorial on Genetic Algorithms and Genetic Programming

As described before, Genetic Algorithms and Genetic Programming are a population based search strategy inspired by the principle of natural evolution. They are guided by a selection mechanism that prefers better individuals and creating a new generation of individuals by random mutation and crossover between multiple parents.
The main parameters for Genetic Algorithms are the population size, the selection mechanism and the solution representation used. Depending on the solution representation used one has to select proper mutation and crossover operators and find suitable values for the associated crossover and mutation probabilities p_c and p_m.
Note that since the general strategy of Genetic Algorithms and Genetic Programming is basically the same, the Genetic Algorithms module includes Genetic Programming in JavaEvA. Simply choose a Genetic Programming type problem, choose a tree based representation and proper mutation and crossover operators and start the optimization. In this this case the type of the algorithm is not given by the search strategy but the data type used.



This is a Genetic Algorithm, which transforms into a GP or GE if the appropriate problem and genotype is used.

elitism	✓
numberOfPartners	1
outputFileName	none
parentSelection	SelectXProbRouletteWheel
partnerSelection	SelectXProbRouletteWheel
population	Population
problem	BMinimizeBits
seed	100
terminator	EvaluationTerminator

Figure 3.9: The Genetic Algorithms and Genetic Programming GUI frame

3.5.1 User's Description

In case of Genetic Algorithms one has to select the selection mechanisms for parent and partner selection, the number of partners for crossover, the population size and whether elitism is to be used, see fig. 3.9:

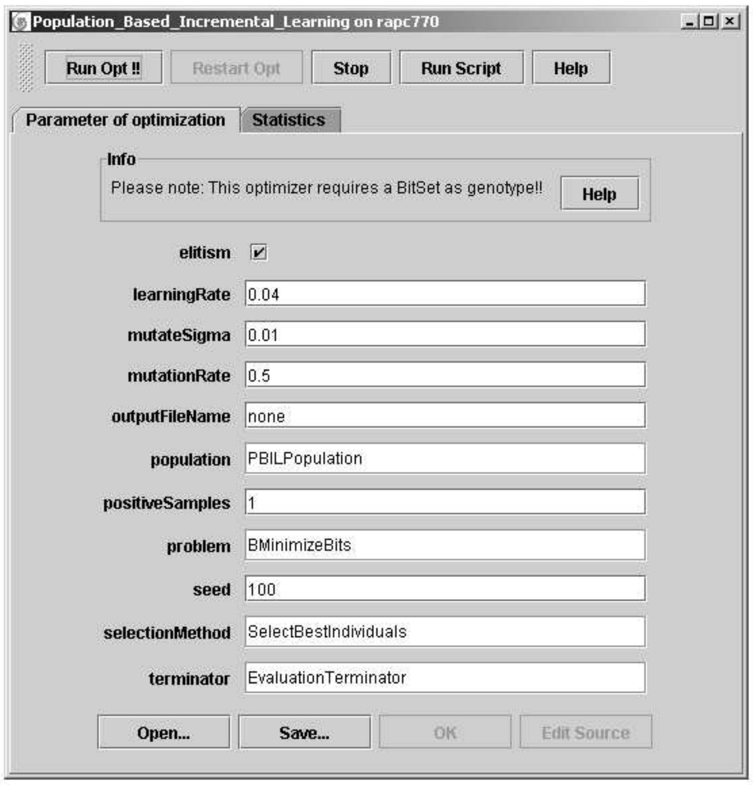
- **Output File Name**: gives the name of the output file where the optimization results are stored. Since this optimizer belongs to the Genetic Optimization family, the basic data output is currently best, mean and worst fitness per generation. Any additional data needs to be specified in the problem implementation and is therefore problem dependent.
- **Population**: here one can specify the population size of the optimization algorithm. In case of Genetic Algorithms the population size can be critical, depending on the problem instance. Try different population sizes ranging from ten individuals to several hundred or even more to find suitable values for a given optimization problem.
- **Problem**: here one can specify the problem to optimize and also the representation, initialization method, mutation and crossover rates and methods. Please refer to sec. 3.9 for further details and examples on problem implementations.
- **Seed**: specifies the starting value for the random number generator, which allows reproducible experiments in case of stochastic optimization strategies. Set the seed to zero to use a random starting condition.
- **Terminator**: gives the termination criterion. In the Genetic Optimization framework currently the termination criterion is limited to choosing the maximal number of fitness evaluations, the maximum number of iterations or a target fitness value.
- **Elitism**: this flag indicates whether or not elitism should be used, see alg. 4. If elitism is activated, the best individual of the current population $P(t)$ survives into the next generation $P(t+1)$. Elitism increases the selection pressure toward the best known individual, which leads to an increased convergence rate on the one hand, but on the other hand may also cause premature convergence.
- **Parent Selection**: this is the selection method to select the parents of the next generation. If n individuals are to be generated for the next generation, we select n singles to give birth to the next generation. This selection mechanism may have a significant impact on the performance of the GA since it gives the selection pressure and thus the speed of convergence and the chance of premature convergence in multimodal search spaces.

- **Partner Selection**: the previously selected parents would stay single if we did not select additional partners for each single. Typically parent and partner selection would be the same, but having separate selection mechanisms allows for mating restriction schemes, which select proper partner depending on the previously selected parent.
- **Number of Partners**: gives the number of crossover partners for reproduction. Some crossover operators like Bit-Simulated crossover may require more than one crossover partner. Other operators may ignore additional partners in case they are limited to recombining only two individuals.

Further details on selection operators and problems, the associated solution representations, mutation and crossover operators are given in sec. 3.9.

3.6 Tutorial on Population Based Incremental Learning

Population Based Incremental Learning (PBIL) belongs to the Density Estimating Algorithms (DEA). These algorithms try to estimate the optimal distribution of alleles to achieve best fitness values. Our PBIL implementation is currently limited to estimating the density distribution for binary genotypes. If non-binary genotypes are used the algorithm will throw an exception and will decline the optimization. In case of binary genotypes the distribution estimate can be reduced to the previously introduced probability vector V, see alg. 6.
If proper solution representations have been selected, PBIL starts with an unbiased probability vector V. In each generation PBIL generates new individuals using V, evaluates the new individuals and updates V depending on the best individuals in the population. Therefore, the parameters of the update rule are essential to the performance of PBIL.



Please note: This optimizer requires a BitSet as genotype!!

elitism	✓
learningRate	0.04
mutateSigma	0.01
mutationRate	0.5
outputFileName	none
population	PBILPopulation
positiveSamples	1
problem	BMinimizeBits
seed	100
selectionMethod	SelectBestIndividuals
terminator	EvaluationTerminator

Figure 3.10: The Population Based Incremental Learning GUI frame

3.6.1 User's Description

When selecting the PBIL strategy one has to choose a proper population size and parameters for the update rule, see fig. 3.10:

- **Output File Name**: gives the name of the output file where the optimization results are stored. Since this optimizer belongs to the Genetic Optimization family, the basic data output is currently best, mean and worst fitness per generation. Any additional data needs to be specified in the problem implementation and is therefore problem dependent.

- **Population**: here one can specify the population size for the PBIL algorithm.

- **Problem**: here one can specify the problem to optimize, the representation and the initialization method. Since the current implementation of PBIL is limited to binary genotypes, the choice of a proper solution representation is critical. The choice of crossover and mutation operators on the other hand can be neglected, because PBIL only relies on initialization of the genotype using the current probability vector V and not on the standard evolutionary operators.
  For some optimization problems it may be necessary to define a problem specific initialization scheme for PBIL, since the standard initialization value for V is $V_i = 0.5$. This leads to an equal number of ones and zeros for an individual in the initial population. In case of knapsack like problems such an initialization is wasteful. The performance of PBIL can be significantly increased if the probability vector V is initialized properly depending on the problem.
  Please refer to sec. 3.9 for further details and examples on problem implementations.

- **Seed**: specifies the starting value for the random number generator, which allows reproducible experiments in case of stochastic optimization strategies. Set the seed to zero to use a random starting condition.

- **Terminator**: gives the termination criterion. In the Genetic Optimization framework currently the termination criterion is limited to choosing the maximal number of fitness evaluations, the maximum number of iterations or a target fitness value.

- **Elitism**: this flag indicates whether or not elitism should be used. If elitism is activated, the best individual of the current population $P(t)$ survives into the next generation $P(t+1)$. Elitism increases the selection pressure toward the best known individual, this leads to increased convergence rate, but may also cause premature convergence.

- **Positive Samples**: gives the number of positive samples to be selected from the current population and used to update the current distribution estimate. The more positive samples are used the higher the speed of convergence.
- **Selection Method**: gives the selection strategy to select the positive samples. Again the choice of the selection method defines the speed of convergence for the algorithm.
- **Learning Rate**: gives the learning rate used to update the probability vector V, see alg. 6. The higher the learning rate the higher the speed of convergence and thus the chance of premature convergence.
- **Mutation Rate**: the probability p_m to mutate the probability vector V. This mutation is introduced to limit the chance of premature convergence and to escape local optima.
- **Mutate Sigma**: gives the size of mutation on a single randomly selected element i of the vector V.

$$V_i' = V_i + \sigma \cdot N(0,1) \tag{3.1}$$

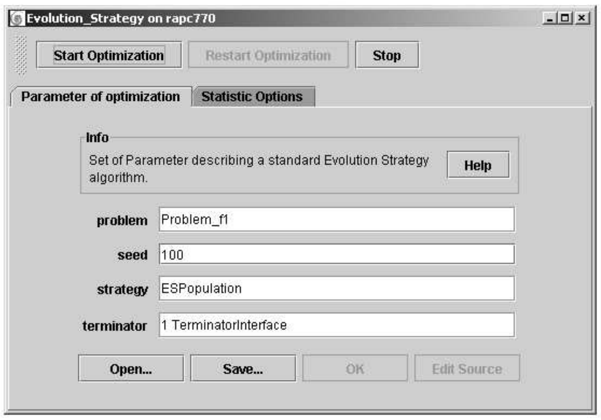
Again, take care to select a proper solution representation, since currently the JavaEvA implementation of PBIL is limited to binary genotypes.

3.7 Evolution Strategies (ES)

Evolution Strategies are most suitable for most real-valued optimization problems. The ES described here refers to the theory of ES given in section 2.5.

3.7.1 User's Description

After loading the Evolution Strategies module you get the main parameter panel, which shows the parameters for the ES optimization algorithm (figure 3.11).



Set of Parameter describing a standard Evolution Strategy algorithm.

problem Problem_f1

seed 100

strategy ESPopulation

terminator 1 TerminatorInterface

Figure 3.11: The Evolution Strategies GUI frame.

The following properties of an ES are editable:

- **Problem**: The problem to be solved.
- **Seed**: A seed value for the random number generator. For Seed = 0 the seed value is itself randomly selected.
- **Strategy**: Properties of the ES population.
- **Terminator**: The termination criterion for the algorithm.

First you have to choose a problem, which has to be optimized by the ES. All ES problems have real valued objective variables. There is a set of popular test functions available. e.g.:

- **Problem_f1**: Sphere test function

$$f_{Sphere}(\vec{x}) = \sum_{i=1}^{n} x_i^2 \tag{3.2}$$

$-5.12 \leq x_i \leq 5.12$; $n = 10$;

- **Problem_f15**: Weighted Sphere test function

$$f_{WSphere}(\vec{x}) = \sum_{i=1}^{n} i \cdot x_i^2 \tag{3.3}$$

$-5.12 \leq x_i \leq 5.12$; $n = 10$; $min(f_{WSphere}) = f_{WSphere}(0,..,0) = 0$

- **Problem_Rastrigin**: Rastrigin test function

$$f_{Rastrigin}\left(\vec{x}\right) = 10 \cdot n + \sum_{i=1}^{n}(x_i^2 - \cos\left(2\pi x_i\right)) \tag{3.4}$$

$-32.768 \leq x_i \leq 32.768$; $n = 10$; $min(f_{Rastrigin}) = f_{Rastrigin}(0,..,0) = 0$

- **Problem_f8**: Rosenbrock test function

$$f_{Rosen}(\vec{x}) = \sum_{i=1}^{n}(100 \cdot (x_{i+1} - x_i)^2 + (x_i - 1)^2) \tag{3.5}$$

$-5.12 \leq x_i \leq 5.12$; $n = 10$;

- **Problem_Ackley**: Ackley test function

$$\begin{aligned} f_{Ackley}(\vec{x}) &= 20 + e - 20 \exp\left(-0.2 \cdot \sqrt{\frac{1}{n} \cdot \sum_{i=1}^{n} x_i^2}\right) \\ &\quad - \exp\left(\frac{1}{n}\sum_{i=1}^{n}\cos\left(2\pi x_i\right)\right) \end{aligned} \tag{3.6}$$

$-32.768 \leq x_i \leq 32.768$; $n = 10$;

- **(Problem_Branin** Branin test function

$$f_{Branin}(x_1, x_2) = 10 + 10 \cdot (1 - \frac{1}{8\pi}\cos x_0) + (\frac{5}{\pi}x_0 - \frac{5.1}{4\pi^2}x_0^2 + x_1 - 6)^2 \tag{3.7}$$

$-5 \leq x_1 \leq 10$;$-5 \leq x_2 \leq 15$;

- **Problem_f2**:Schwefel's test function 1.2

$$f_{Schwefel}(\vec{x}) = \sum_{i=1}^{n}\left(\sum_{j=1}^{i} x_j\right)^2 \tag{3.8}$$

$-5.12 \leq x_i \leq 5.12$; $n = 10$;

- **Problem_Step**: Step test function

$$f_{Step}(\vec{x}) = \sum_{i=1}^{n} \lfloor x_i \rfloor \qquad (3.9)$$

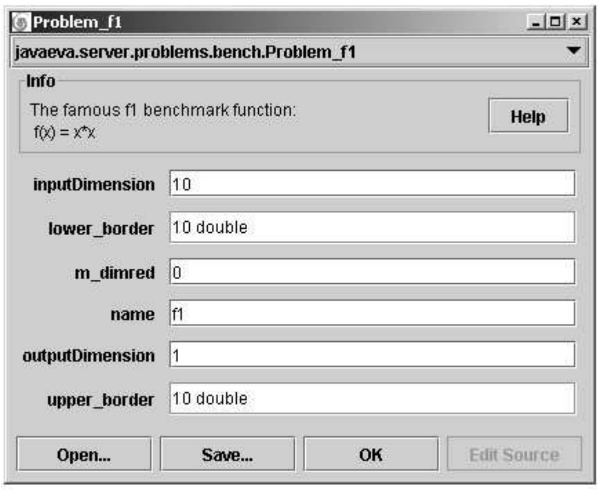
$-5.12 \leq x_i \leq 5.12$; $n = 10$;

- **Problem_Griewangk**: Griewank test function:

$$f_{Griewank}(\vec{x}) = 1 + \frac{1}{200} \sum_{i=1}^{n} x_i^2 - \prod_{i=1}^{n} \cos\left(\frac{x_i}{\sqrt{i}}\right) \qquad (3.10)$$

$-100 \leq x_i \leq 100$; $n = 10$;

Figure 3.12 shows a typical problem. Via the **Help** button you get additional information describing the problem. The editable properties of the problems are self-explanatory and connected with **tooltips**.


Problem_f1

javaeva.server.problems.bench.Problem_f1

Info

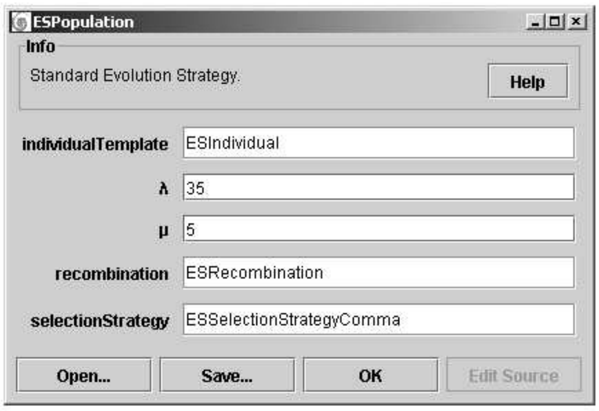
The famous f1 benchmark function:
f(x) = x*x

inputDimension	10
lower_border	10 double
m_dimred	0
name	f1
outputDimension	1
upper_border	10 double

Figure 3.12: The ES problem parameter frame.

The **showViewer** flag, if available, causes a problem specific visualization of the solutions of the problem during the optimization run (try e.g. the lens problem). With the **plotObjectives** flag one may plot the real valued objective values of the current best solution in a seperate frame.
Next one must specify the **population parameters** of the ES, which are the most important ones (figure 3.13).

The following properties of an ES population are editable:

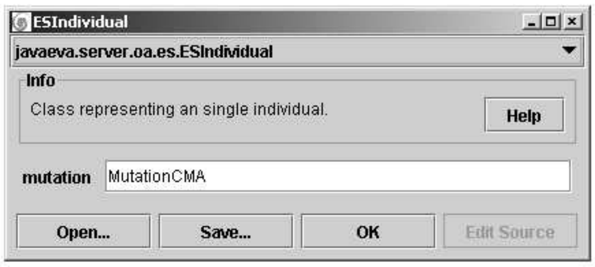


Standard Evolution Strategy.

individualTemplate: ESIndividual
λ: 35
μ: 5
recombination: ESRecombination
selectionStrategy: ESSelectionStrategyComma

Figure 3.13: The ES population parameter frame

- **individualTemplate**: A prototype of an individual (contains mutation operator).
- λ: The population size of the parents.
- μ: The population size of the children.
- **Recombination**: A recombination operator.
- **SelectionStrategy**: A fitness based selection operator.



javaeva.server.oa.es.ESIndividual

Class representing an single individual.

mutation: MutationCMA

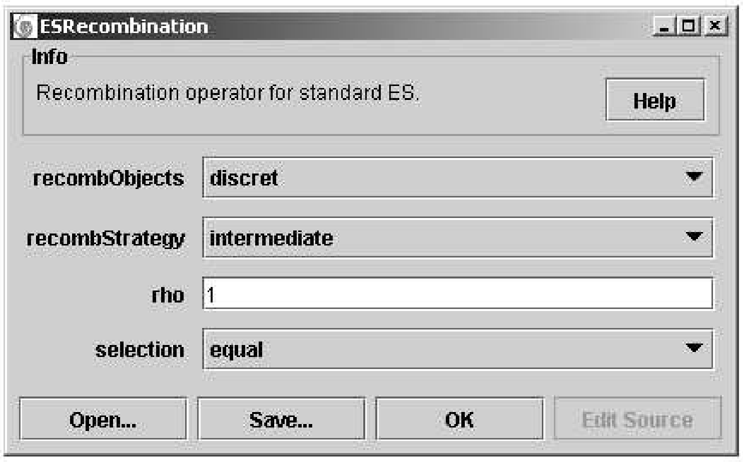
Figure 3.14: The ES individual parameter frame.

The only editable content of an individual is the **Mutation** operator, which is the most important for ES. The following mutation operators are available:

- **MutationCMA**: Covariance Matrix Adaptation (CMA).
- **MutationMVA**: Main Vector Adaptation (MVA).

- **MutationSuccessRule**: The 1/5 success rule.
- **MutationMSRGlobal**: The global step size adaptation.
- **MutationRandom**: A totally randomized mutation operator without step size adaption.

Next one may edit the recombination operator (figure 3.15). The properties are self-explanatory. For $\rho = 1$ (only one parent individual) recombination is inactive.


ESRecombination

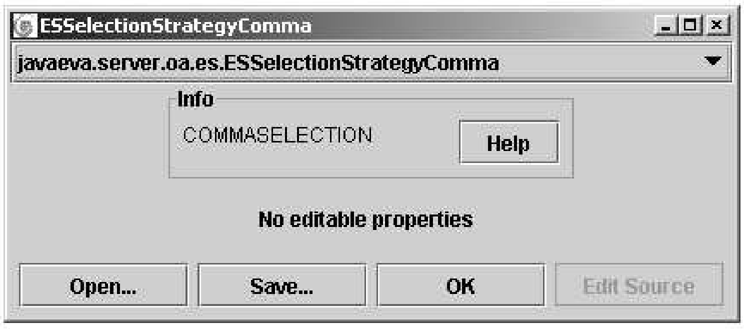
Info

Recombination operator for standard ES.

recombObjects	discret
recombStrategy	intermediate
rho	1
selection	equal

Figure 3.15: The ES recombination parameter frame

Finally you have to select the fitness based **Selection** operator 3.16. The fol-


ESSelectionStrategyComma

javaeva.server.oa.es.ESSelectionStrategyComma

Info

COMMASELECTION

No editable properties

Figure 3.16: The ES selection parameter frame

lowing mutation operators are available:

- **ESSelectionStrategyPlus**: Selection from parents plus offspring individuals.

- **ESSelectionStrategyComma**: Selection from offspring individuals only.
- **ESSelectionStrategyMedian**: This selection strategy is commendable for steady state optimizations with $\lambda = 1$.

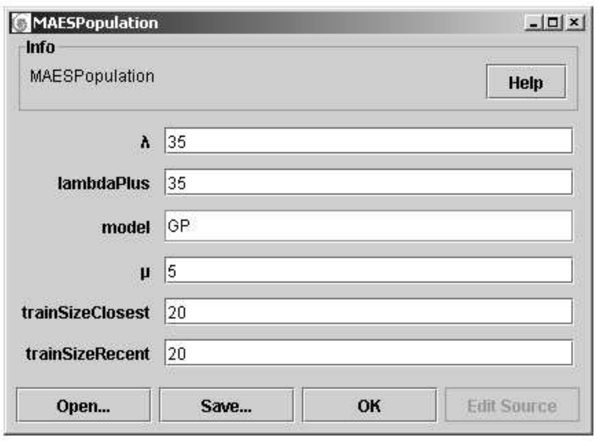
After one has set all necessary parameters,one may start the optimization process. This GUI framework can be used to compare the impact of different parameter settings on the performance of the ES.

3.8 Model Assisted Evolution Strategies

Compared to standard Evolution Strategies the Model Assisted Evolution Strategies reduce the number of true fitness evaluations significantly, by using a model of the true target function as surrogate function. Assuming the model can be evaluated and trained more efficiently than the true target function, the Model Assisted Evolution Strategies can save considerable amounts of computation effort, for example in design optimization where typically extensive simulations are necessary for a target function evaluation.

3.8.1 User's Description

As discussed before in section 2.6 the Model Assisted Evolution Strategy (MAES) is a extension of a standard Evolution Strategy. For this reason the main parameter panel of the optimization panel has the same appearance as the one of the standard ES panel. The only difference is in the strategy parameter panel (MAESPopulation) (see picture 3.17).



MAESPopulation

λ	35
lambdaPlus	35
model	GP
μ	5
trainSizeClosest	20
trainSizeRecent	20

Figure 3.17: The model assisted ES population parameter frame

The model assisted ES (MAES) population parameter has the following additional properties:

- **LambdaPlus**: The size of the model pre-selected individuals: $\lambda_{Plus} >= \lambda$. For $\lambda_{Plus} = \lambda$ you have no model impact on the optimization process.
- **Model**: The regression model for fitness prediction.
- **Train**: The model size is given by the number of last evaluated individuals,which are used to train the model.

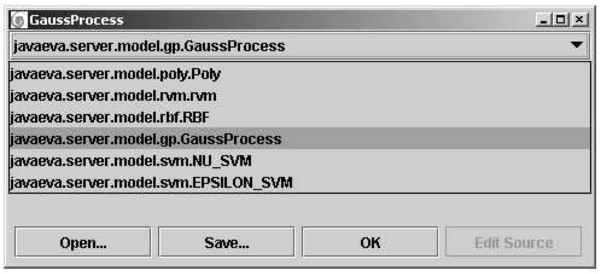

GaussProcess
javaeva.server.model.gp.GaussProcess
javaeva.server.model.poly.Poly
javaeva.server.model.rvm.rvm
javaeva.server.model.rbf.RBF
javaeva.server.model.gp.GaussProcess
javaeva.server.model.svm.NU_SVM
javaeva.server.model.svm.EPSILON_SVM
Open...
Save...
OK
Edit Source

Figure 3.18: A parameter frame for the used regression models in MAES.

In JavaEvA different regression models for fitness approximation are available:

- **Poly**: Classical polynominal Regression Models.
- **GaussProcess**: Gaussian Processes.
- **NU_SVM**: Support Vector Machines for ν-regression.
- **RVM**: Relevance Vector Machines.
- **RBF**: Radial-Basis-Function networks.

The editable parameters of the models are self-explanatory.

3.9 Tutorial on Genetic Optimization

The module Genetic Optimization (GO) tries to give a general approach to Evolutionary Algorithms (EA) and other iterative stochastic optimization strategies. The concept of GO is based on a special scheme of EA. In this scheme the optimization algorithms and the data types that are to be optimized are clearly separated. This scheme is inspired by Michalewicz book 'Genetic Algorithms + Data Structures = Evolution Programs' [28]. With Genetic Optimizing we even go a bit further by not only separating the optimization algorithms and the data but by separating the optimization algorithm and the optimization problem completely. Thus, making the solution representation, i.e. the EA individual, a property of the optimization problem and also the choice of the evolutionary operators of mutation and crossover, since they belong to the EA individual.

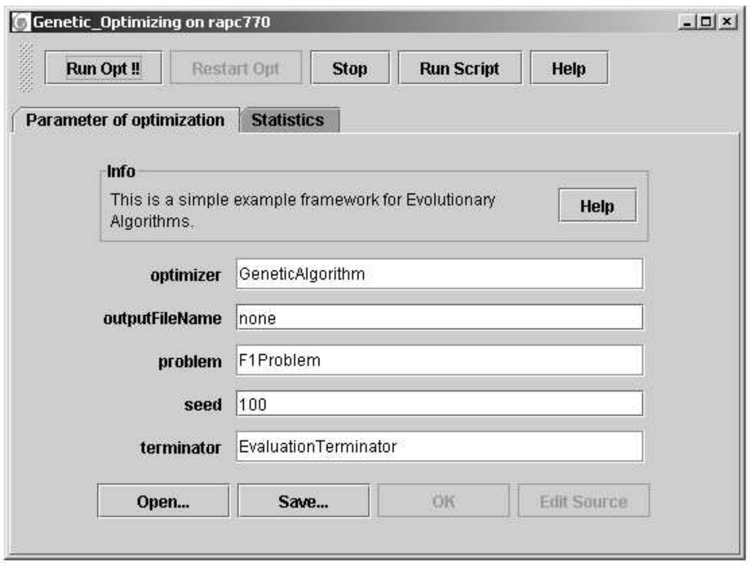
The optimization algorithm is given a reference to the optimization problem to solve, which provides methods to initialized an EA population and to evaluate it. The optimization algorithm is of course able to manage a population using selection, clone, deletion and addition operators on the population or multiple populations. To generate new individuals or to alter existing ones the optimization algorithms may access methods like clone, crossover and mutate on EA individuals via interfaces. But the choice of the actual operators applied is left to the individuals. The optimization algorithm may even be totally ignorant of the actual solution representation used.

In case the optimization algorithm requires specific solution representations like for example the current implementation of the Population Based Incremental Learning approach, it is able to cast the EA individuals in the population to a more specific interface. This may fail when the user decided to use a different solution representation and the optimization algorithm may terminate. Currently, we can't prevent such unfortunate combinations, but are working to resolve this issue.

Similar to setting the proper solution representation and operators from within the optimization problem, the Genetic Optimization module tries to leave as many problem specific choices to the optimization problem as possible. For example each optimization problem has to take care of a suitable graphical visualization of a solution found, of logging the statistical data necessary for the given application problem or of deciding on a suitable local search heuristic. But the general requirements for the implementation of an optimization problem are currently rather few. We require general initialization methods for the problem, for an initial population and evaluation methods on the level of populations and individuals. Additional input/output methods are often optional.

A local search and application specific additional statistics on the optimization process currently has to taken care of within the evaluation methods. But we are currently working on a more general framework for so called Memetic Algorithms and logging problem specific data.

We believe that this scheme gives a user on the one hand a lot of freedom regarding the choice of optimization strategies, see sec. 3.9.1 for a list of available elements that can be combined, and regarding the implementation of a given optimization problem on the other hand, see sec. 4.3 for an example for a own problem implementation.



This is a simple example framework for Evolutionary Algorithms.

optimizer: GeneticAlgorithm
outputFileName: none
problem: F1Problem
seed: 100
terminator: EvaluationTerminator

Figure 3.19: The Genetic Optimization GUI frame

3.9.1 User's Description

As already described the two main properties of the Genetic Optimization module are the optimization algorithm and the optimization problem, the others are pretty much standard.

- **OutputFileName**, gives the name of the output file where to store the optimization results. The basic data output is currently best, mean and worst fitness per generation. Any additional data needs to be specified in the problem implementation and is therefore problem dependent.

- **Seed**, specifies the starting condition for the random number generator, which allows reproducible experiments in case of stochastic optimization strategies. Set the seed to zero to use a random starting condition.

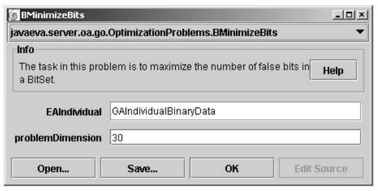
- **Terminator**, gives the breaking criteria. The breaking criteria are currently limited to choosing the maximal number of fitness evaluations, the maximum number of iterations or a target fitness value.

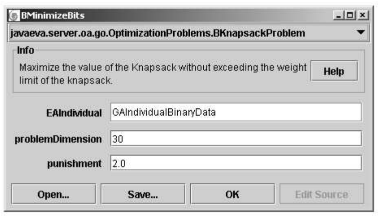
- **Problem**, some problem instances will be discussed in detail in the subsection GO Optimization Problems of this section.
- **Optimizer**, most of the optimization algorithms available under Genetic Optimization will be described in detail in the subsection GO Optimization Algorithms of this section.

In the following subsections we will give exemplary descriptions of the available elements ranging from optimization strategies, selection methods, mutation and crossover operators to optimization problems. But we will only explain non standard elements of GO in detail, in case of standard methods like roulette-wheel selection or ES mutation operators we would like to refer the user to the general description given in chap. 2.
First we will give examples of some optimization problems implemented for the GO module in JavaEvA. Then we will explain the available data types, the EA individual implementations and the available mutation and crossover operators. Finally we will give details on the optimization strategies and the selection methods.

GO Optimization problems

In the context of the GO module the optimization problem defines the data type that is to be optimized and also the type of EA individual to be used. Therefore, the mutation and crossover operators and the mutation/crossover rates have to be set in the optimization problem via the EAIndividual.
Per definition all optimization problems in GO are to be minimized.

GUI Element	Function and parameters
 BMinimizeBits javaeva.server.oa.go.OptimizationProblems.BMinimizeBits Info The task in this problem is to maximize the number of false bits in a BitSet. EAIndividual: GAIndividualBinaryData problemDimension: 30 	The MinimizeBits problems is very simple, the solution representation is a binary string, therefore the leading 'B' in the name of the problem. The problem is to minimize the number of non-zero bits in the solution. • **EAIndividual**, allows you to choose the solution representation selecting from EA individuals that comply to the Interface-DataTypeBinary. The GUI element for the EA individual allows you to select mutation/crossover operators and rates. • **Problem Dimension**, gives the length of the solution.


BMinimizeBits

javaeva.server.oa.go.OptimizationProblems.BKnapsackProblem

Info

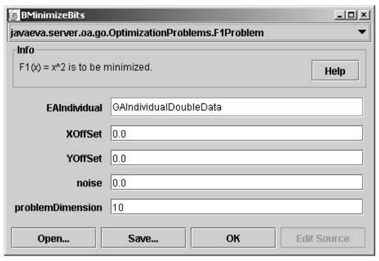
Maximize the value of the Knapsack without exceeding the weight limit of the knapsack.

EAIndividual	GAIndividualBinaryData
problemDimension	30
punishment	2.0

The Knapsack problem is also a binary problem and is given by the task to select a limited number of elements into a knapsack to achieve a maximum value of the knapsack, while at the same time not exceeding a maximum weight for the knapsack. The value and the weight of the available elements in this problem instance are hard coded. The maximum weight of the knapsack is given as 5000 and the maximal value of a selection of elements not exceeding this weight, and therefore the global optimum, is 5100. Because GO requires minimization problem the value is negative and we add an offset off 5100 to the fitness such that the global optimum is actually at zero to allow log scale for the fitness values.

- **EAIndividual**, choose the solution representation and EA mutation/crossover operators and rates.
- **Problem Dimension**, is inherited from BMinimizeBits, but is not used in this context.
- **Punishment**, gives the rate by which to penalize deviations from the weight constraint, see sec. 2.2 for details on the penalty strategy to meet constraints.

The Knapsack problem is a nice example for a multi-modal search space where a Hill-Climber would typically fail. Because a problem specific initialization scheme is currently not implemented, the Monte-Carlo search is unable to find good solutions, while both PBIL and GA are hindered by unfavorable initial populations. Please take care to select a suitable punish rate to prevent infeasible solutions. Punishment for exceeded weight should be greater than the additional gain in value.



javaeva.server.oa.go.OptimizationProblems.F1Problem

Info

F1(x) = x^2 is to be minimized.

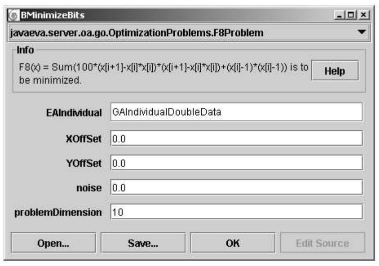
EAIndividual	GAIndividualDoubleData
XOffSet	0.0
YOffSet	0.0
noise	0.0
problemDimension	10

The F1Problem is a simple n-dimensional real-valued benchmark function minimizing the target function:

$$f(x) = \sum_{i=0}^{n} x_i^2 \tag{3.11}$$

The trailing 'F' is used to indicate the data type float used for this problem instance.

- **EAIndividual**, choose the solution representation complying to the InterfaceDataTypeDouble and EA mutation/crossover operators and rates.
- **X Offset**, adds an offset to all decision variables x_i. Such an offset can be used to detect a bias of the optimization algorithm toward a certain region of the search space.
- **Y Offset**, adds an offset to the output value y of the target function. This allows you to detect whether or not the optimization algorithm is sensible to the range of the target values.
- **Noise**, gives the level of noise to be added to the target function. This allows you to investigate the impact of noise on the performance of the optimization algorithm.
- **Problem Dimension**, n gives the dimension of the optimization problem.



javaeva.server.oa.go.OptimizationProblems.F8Problem

Info

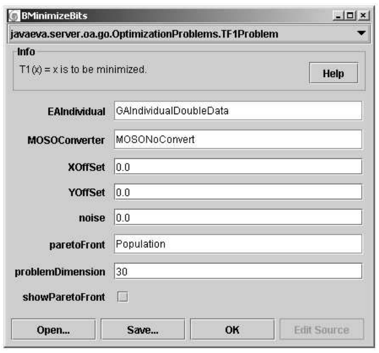
F8(x) = Sum(100*(x[i+1]-x[i]*x[i])*(x[i+1]-x[i]*x[i])+(x[i]-1)*(x[i]-1)) is to be minimized.

EAIndividual	GAIndividualDoubleData
XOffSet	0.0
YOffSet	0.0
noise	0.0
problemDimension	10

The F8Problem is also called the Rosenbrock function and is given by minimizing the target function:

$$f(x) = \sum_{i=2}^{n} \left(100 \cdot \left(x_{i-1} - x_i^2\right)^2 + \left(x_i - 1\right)^2\right) \tag{3.12}$$

The general properties are the same as in case of the F1Problem.



BMinimizeBits

javaeva.server.oa.go.OptimizationProblems.TF1Problem

Info

T1 (x) = x is to be minimized.

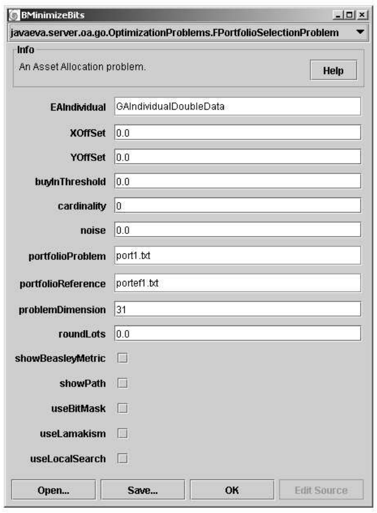
EAIndividual	GAIndividualDoubleData
MOSOConverter	MOSONoConvert
XOffSet	0.0
YOffSet	0.0
noise	0.0
paretoFront	Population
problemDimension	30
showParetoFront	☐

The TF1Problem is an example for a multi-objective optimization problem and is given as

$$\begin{aligned} f_1(x) &= x_1, \\ f_2(x) &= g(x)h(f_1(x), g(x)), \\ g(x) &= 1 - \frac{9}{n-1}\sum_{i=2}^{n} x_i, \\ h(f_1, g) &= 1 - (f_1/g)^2 \end{aligned} \tag{3.13}$$

with $n = 30$ and $x \in [0, 1]^n$.

- **EAIndividual**, choose the solution representation complying to the InterfaceDataTypeDouble and EA mutation/crossover operators and rates.
- **X Offset**, adds an offset to all decision variables x_i.
- **Y Offset**, adds an offset to the output value y of the target function.
- **Noise**, gives the level of noise to be added to the target function.
- **Problem Dimension**, $n = 30$ gives the dimension of the optimization problem.
- **MOSOConverter**, allows you to convert the multi-objective optimization problem into a single-objective optimization problem, for example by weight aggregation.
- **Pareto-Front**, gives an example how to log the result of the optimization process from inside the problem. The internal pareto-front of the problem allows you to log a pareto-front even if a single-objective optimization algorithm is used.
- **Show Pareto-Front**, plots the obtained pareto-front either the one logged by the problem or the one provided by a multi-objective optimization algorithm.



javaeva.server.oa.go.OptimizationProblems.FPortfolioSelectionProblem

Info

An Asset Allocation problem.

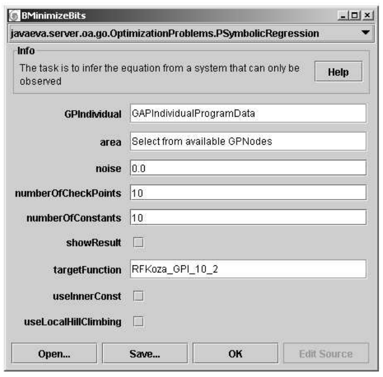
EAIndividual	GAIndividualDoubleData
XOffSet	0.0
YOffSet	0.0
buyInThreshold	0.0
cardinality	0
noise	0.0
portfolioProblem	port1.txt
portfolioReference	portef1.txt
problemDimension	31
roundLots	0.0
showBeasleyMetric	☐
showPath	☐
useBitMask	☐
useLamakism	☐
useLocalSearch	☐

The FPortfolioSelectionProblem is a multi-objective real-world application problem from the field of financial engineering where portfolios are to be found that

$$\begin{aligned} \text{minimize :} \quad & \sigma_p = \sum_{i=1}^{N} \sum_{j=1}^{N} w_i \cdot w_j \cdot \sigma_{ij}, \quad (3.14)\\ \text{maximize :} \quad & \mu_p = \sum_{i=1}^{N} w_i \cdot \mu_i, \\ \text{subject to :} \quad & \sum_{i=1}^{N} w_i = 1 \quad \text{and} \\ & 0 \leq w_i \leq 1 \end{aligned}$$

more details on this problem and how to solve it can be found in [41].

- **EAIndividual**, choose the solution representation complying to the InterfaceDataTypeDouble and EA mutation/crossover operators and rates.
- **Portfolio Problem and Reference**, contain the parameters of the available assets and the unconstrained reference solution [3].
- **Cardinality Constraints**, restrict the maximal number of assets used in the portfolio, $\sum_{i=1}^{N} \text{sign}(w_i) = K$.
- **Buy-in Threshold Constraints**, give the minimum amount that is to be purchased, i.e. $w_i \geq l_i \quad \forall \quad w_i > 0; i = 1, .., N$.
- **Roundlot Constraints**, give the smallest volumes c_i that can be purchased for each asset, $w_i = y_i \cdot c_i; \quad i = 1, .., N$ and $y_i \in \mathbb{Z}$.
- **Use BitMask**, together with a GAESIndividualBinaryDoubleData allows are more efficient search.
- **Lamarkism**, writes repaired phenotypes back into the individuals gentoype.
- **X Offset, Y Offset, Problem Dimension and Local Search**, are not active in this environment.



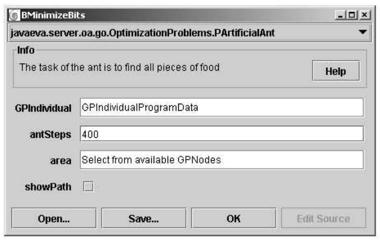
javaeva.server.oa.go.OptimizationProblems.PSymbolicRegression

The task is to infer the equation from a system that can only be observed

GPIndividual	GAPIndividualProgramData
area	Select from available GPNodes
noise	0.0
numberOfCheckPoints	10
numberOfConstants	10
showResult	☐
targetFunction	RFKoza_GPI_10_2
useInnerConst	☐
useLocalHillClimbing	☐

The PSymbolicRegression problem is to search for a mathematical function $\hat{f}(x)$ that fits a given target function $f(x)$. This is a standard GP problem instance, see [22] for further details.

- **EAIndividual**, choose the solution representation complying to the InterfaceDataTypeProgram and EA mutation/crossover operators and rates.
- **Target Function**, allows you to select the target function $f(x)$.
- **Area**, enables you to select the nodes to use for the symbolic regression.
- **Show Result**, is another example for a problem specific viewer. This one plots the target function $f(x)$ and the currently best $\hat{f}(x)$.
- **Number of Check Points**, gives the number of check points for the fitness function.
- **Number of Constants**, gives the number of ephemeral constants.
- **Use Inner Constants and Local Hill-Climbing**, are not active in this environment.



javaeva.server.oa.go.OptimizationProblems.PArtificialAnt

Info

The task of the ant is to find all pieces of food

GPIndividual: GPIndividualProgramData

antSteps: 400

area: Select from available GPNodes

showPath: ☐

The PArtificalAnt problem is given by finding a program that controlls the behavior of an artificial ant to collect all food particles in a given environment with a limited number of steps. The Artificial Ant Problem on the Santa-Free trail is another typical GP problem instance, see [22] for further details.

- **EAIndividual**, choose the solution representation complying to the InterfaceDataTypeProgram and EA mutation/crossover operators and rates.
- **Ant Steps**, gives the number of steps allowed to be performed for each ant.
- **Area**, enables you to select the nodes to use for ant program.
- **Show Path**, shows the path of the best performing ant. The environment is the Santa-Fee trail in a toroidal world. The uncollected food particles are black while the collected particles are red. The ant starts in the upper left corner and the resulting path the ant is colored from light green to dark purple. In the lower section you can see the evolved program code.

Available GO data types and EA Individuals

The available data types that can be currently optimized in the GO module range from simple bit-strings, integer and double arrays to permutations and even general program structures. From the problem point of view the phenotype data can be accessed using the InterfaceDataTypeX. In case there are multiple ways to represent a given phenotype the alternative instances will be discussed in the following list.

- **InterfaceDataTypeBinary**, are for bit-string phenotypes. Currently there are two EA individuals that can represent this data type, the GAIndividualBinaryData and the ESIndividualBinaryData. The ESIndividualBinaryData offers two alternative encodings, either by using a hard boundary to decided whether the real-valued genotype x_i is to be interpreted as boolean true or false or by using the real-valued vector x as vector of probabilities similar to the vector V in PBIL used to generate the phenotype.

- **InterfaceDataTypeInteger**, are for integer arrays. Again you can either used a ESIndividualIntegerData, which rounds the real-valued decision vector x to integer values or you can use a GAIndividualIntegerData. For the GAIndividualIntegerData you can either choose to use a standard binary encoding for integer values or to use a gray encoding for integer values.

- **InterfaceDataTypeDouble**, represent double arrays. Here you can choose using GAIndividualDoubleData again using either standard binary or gray encoding to decoded double values from the binary genotype. Alternatively, you can choose ESIndividualDoubleData that can omit any mapping function for the real-valued decision parameters.

- **InterfaceDataTypePermuation**, represent permutations. Currently there is only one implementation for permutation data, the OBGAIndividualPermutationData.

- **InterfaceDataTypeProgram**, code general computer programs, currently stored as program trees. You can choose between using the GPIndividualProgramData, which is a Koza style program tree representation, and the GEIndividualProgramData, which uses a binary genotype to represent the program and uses a grammar to decode the genotype into a phenotype.

These interfaces can be combined to generate more complicated data structures. The GAESIndividualBinaryDoubleData is just one example, which is especially well suited for searching for sparse real-valued vectors x.
To access the genotypes of a given EA individual for mutation or crossover typically the InterfaceXIndividual methods are used. In the following paragraphs we will discuss some mutation/crossover operators that can be applied to the different interfaces respectively.

- **InterfaceGAIndividual**, the GA individual relies on a bit-string genotype. Therefore, typical EA operators include BitFlipMutation, BitSwapMutation, one-point- or N-point-crossover, UniformCrossover and BitSimulatedCrossover.

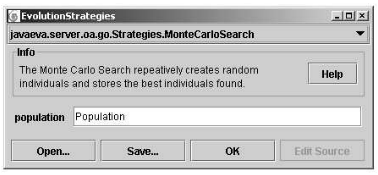
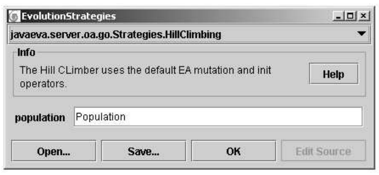
- **InterfaceESIndividual**, the ES individual is based on a vector of real-valued decision parameters. The strategy parameters depend on the mutation operator used and are typically stored in the mutation operators themselfs. Thus, it is necessary to make a deep clone to the mutation operators when an individual is to be duplicated. Typical mutation operators for ES individuals include one-point mutation, global- or local mutation, MainVector- and Covariance-Matrix Adaptation. Real-valued crossover operators which are usually more important for GA strategies using a real-valued genotype, also called real-valued GAs, include DiscreteCrossover, IntermediateCrossover, Arithmetical-Crossover, FlatCrossover, BLX-α-Crossover and BitSimulatedCrossover.

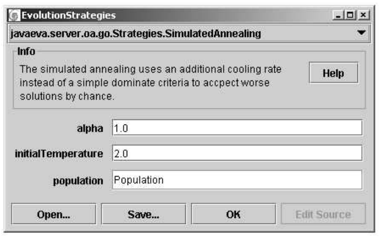
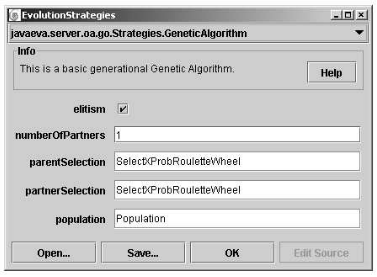
- **InterfaceGPIndividual**, the GP individual is based on a Koza style program tree. Currently there are only few default EA operators available for this representation.
- **InterfaceOBGAIndividual**, the Order Based GA (OBGA) individual is suited to represent permutation data and allows mutation operators like Flip-Mutation and InverseMutation and crossover operators like PMX- and PMX-UniformCrossover.

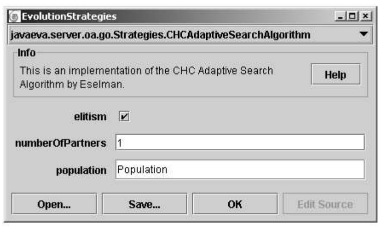
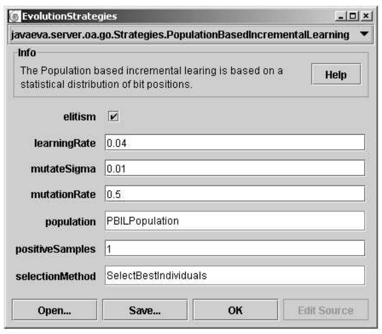
Please take care to select a suitable mutation/crossover operator for the individuals. In case you select a ES mutation operator for a binary-string based GA individual, the mutation operator will fail to find a suitable genotype and will decline processing the individual.

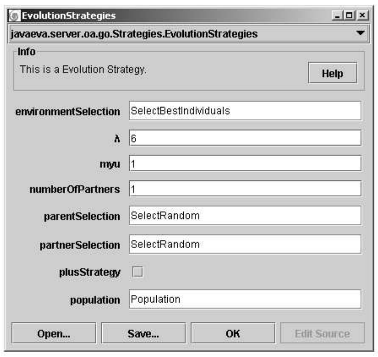
Available GO Optimization Algorithms

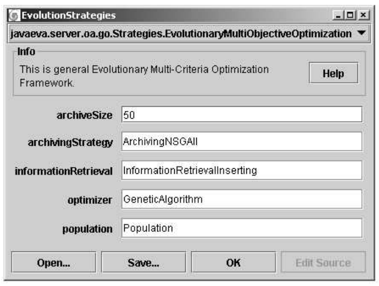
The following list will give a complete enumeration of all available optimization strategies in the GO module.

GUI Element	Function and parameters
 EvolutionStrategies javaeva.server.oa.go.Strategies.MonteCarloSearch Info The Monte Carlo Search repeatively creates random individuals and stores the best individuals found. population: Population	This is a simple random search strategy. See also sec. 2.1.1 and sec. 3.2 for further explanations. • **Population**, allows you to set the population size.
 EvolutionStrategies javaeva.server.oa.go.Strategies.HillClimbing Info The Hill CLimber uses the default EA mutation and init operators. population: Population	This is a greedy Hill-Climber strategy. See also sec. 2.1.2 and sec. 3.3 for further explanations. • **Population**, the population size gives the number of Hill-Climbers for a multi-start Hill-Climber.

 EvolutionStrategies javaeva.server.oa.go.Strategies.SimulatedAnnealing Info The simulated annealing uses an additional cooling rate instead of a simple dominate criteria to accpect worse solutions by chance. alpha: 1.0 initialTemperature: 2.0 population: Population 	This is a Simulated Annealing strategy. See also sec. 2.1.3 and sec. 3.4 for further explanations. • **Population**, the population size gives the number of multi-starts for the Simulated Annealing approach. • **Initial Temperature**, gives the starting temperature T for the algorithm. • **Alpha**, gives a linear cooling rate for the Simulated Annealing algorithm.
 EvolutionStrategies javaeva.server.oa.go.Strategies.GeneticAlgorithm Info This is a basic generational Genetic Algorithm. elitism: ✓ numberOfPartners: 1 parentSelection: SelectXProbRouletteWheel partnerSelection: SelectXProbRouletteWheel population: Population 	This is a general Genetic Algorithm strategy, also suitable for Genetic Programming. See also sec. 2.2 and sec. 3.5 for further explanations. • **Population**, allows you to set the population size. • **Elitism**, this flag indicates whether or not elitism is to be used. • **Parent Selection**, this is the selection method to select the parents for the next generation. • **Partner Selection**, this selection method selects suitable partners for crossover for each previously selected parent. • **Number of Partners**, gives the number of crossover partners to select for reproduction.

 EvolutionStrategies javaeva.server.oa.go.Strategies.CHCAdaptiveSearchAlgorithm Info: This is an implementation of the CHC Adaptive Search Algorithm by Eselman. elitism: ✓ numberOfPartners: 1 population: Population	This is a rudimentary implementation of the CHC Adaptive Search. See also sec. 2.3 for further explanations. • **Population**, allows you to set the population size. • **Elitism**, this flag indicates whether or not elitism is to be used. • **Number of Partners**, gives the number of crossover partners for reproduction.
 EvolutionStrategies javaeva.server.oa.go.Strategies.PopulationBasedIncrementalLearning Info: The Population based incremental learing is based on a statistical distribution of bit positions. elitism: ✓ learningRate: 0.04 mutateSigma: 0.01 mutationRate: 0.5 population: PBILPopulation positiveSamples: 1 selectionMethod: SelectBestIndividuals	This is the Population Based Incremental Learning optimizer. Currently, it is only suited for solution representations based on binary genotypes. See also sec. 2.4 and sec. 3.6 for further explanations. • **Population**, allows you to set the population size. • **Elitism**, this flag indicates whether or not elitism is to be used. • **Positive Samples**, gives the number of positive samples to be selected from the current population. • **Selection Method**, gives the selection strategy to select the positive samples. • **Learning Rate**, gives the learning rate used to update the probability vector V. • **Mutation Rate**, the the probability p_m to mutate the probability vector V. • **Mutate Sigma**, gives the size of mutation on a single randomly selected element i of the vector V.

 EvolutionStrategies javaeva.server.oa.go.Strategies.EvolutionStrategies Info This is a Evolution Strategy. environmentSelection: SelectBestIndividuals λ: 6 myu: 1 numberOfPartners: 1 parentSelection: SelectRandom partnerSelection: SelectRandom plusStrategy: ☐ population: Population 	This is an Evolution Strategy, see also sec. 2.5 for further explanations. • **Population**, allows you to set the population size, but may collied with the settings for μ and λ. Therefore, do not use this property to parameterize the ES, but use the μ and λ properties instead. • **Plus Strategy**, this flag indicates whether or not the $\mu + \lambda$-strategy is to be. • μ, the number of individuals to select as potential parents. • λ, the number of offspring to generate from the parents. • **Environment Selection**, this is the selection method to select the μ parents P_p, typically the SelectBestIndividuals method. • **Parent Selection**, this is the selection method to select the parents from P_p, often SelectRandom. • **Partner Selection**, this selection method selects suitable partners for crossover from P_p for each previously selected parent, often SelectRandom. • **Number of Partners**, gives the number of crossover partners for reproduction.



javaeva.server.oa.go.Strategies.EvolutionaryMultiObjectiveOptimization

Info

This is general Evolutionary Multi-Criteria Optimization Framework.

archiveSize	50
archivingStrategy	ArchivingNSGAII
informationRetrieval	InformationRetrievalInserting
optimizer	GeneticAlgorithm
population	Population

This is a Multi-Objective EA. Our implementation simply adds an archive A to the optimizer. The archive $A(t)$ is updated each generation by adding new pareto-optimal solutions from the current population $P(t)$ to the archive. The archive can also influence the current population by reinserting the archive into the population, this increases the selection pressure toward the pareto-front.

- **Archive Size**, the size of the external archive A.
- **Archiving Strategy**, determines the method how to calculate $A(t + 1)$ from $(A(t) \cup P(t))$.
- **Information Retrieval**, determines how the current archive $A(t)$ influences the next generation $P(t + 1)$. The effect of the information retrieval method is similar to elitism in GA.
- **Population**, this is a short cut to set the population size.
- **Optimizer**, allows you to choose from multiple optimization strategies. Although often a GA is used, you can choose any optimization strategy, but most likely you will need to choose a multi-objective selection criterion for the optimization strategy.

Selection Operators

These are the available selection operators used in Genetic Algorithms, Population Based Incremental Learning and Evolution Strategies. The selection operators are typically used in single-objective optimization problems. In case of multi-objective optimization problems these selection operators default to select each individual based on a randomly selected single-objective.

- **SelectRandom**, this method randomly selects k individuals regardless of their fitness values.

- **SelectBest**, this method selects k time the best individual.
- **SelectBestIndividuals**, this method selects the k best individuals, see also p. 19.
- **SelectTournament**, this method uses the tournament selection method k times to select k individuals, see also p. 19.
- **SelectXProbRouletteWheel**, this method calculates selection probabilities and uses the roulette-wheel metaphor to select individuals, see also p. 20.

There are also some multi-objective selection methods suited for multi-objective optimization like

- **SelectMOMaxiMin**, uses the MaxiMin criteria to calculate the fitness value.
- **SelectMONonDominated**, which randomly selects from non-dominated individuals.
- **SelectMONSGAIICrowedTournament**, is based on the Non-dominated Sorting GA (NSGA) II to select individuals using tournament selection based on their pareto-level and the crowding factor.
- **SelectMOPESA**, is based on the Pareto Envelope-based Selection Algorithm (PESA) to select individuals using tournament selection based on the grid squeeze factor.
- **SelectMOPESAII**, is based on the Pareto Envelope-based Selection Algorithm (PESA) to select individuals using tournament selection based on region based grid squeeze factor.
- **SelectMOSPEAII**, is based on the Strength Pareto EA (SPEA) II to select individuals using tournament selection based on pareto strength.

Selection Probability Calculators

These are the available selection probability calculators that are used for probability based selection methods like the roulette-wheel selection method.

- **SelProbStandard** which uses a simple scaling method as described on p. 19.
- **SelProbStandardScaling** uses are more sophisticated scaling method which removes a possible offset form the fitness values, see p. 19.
- **SelProbBoltzman** is even more sophisticated by taking the mean and the standard deviation of the fitness values into account, see also p. 20.

- **SelProbRanking** uses a simple ranking to determine the selection probabilities, see also p. 20.
- **SelProbLinearRanking** uses a simple ranking with an additional linear scaling to control the selection pressure to determine the selection probabilities, see also p. 20.
- **SelProbNonLinearRanking** uses a simple ranking with an additional nonlinear scaling to control the selection pressure to determine the selection probabilities, see also p. 20.

Chapter 4

JavaEvA Applications

This chapter will explain how to implement your own optimization problem and how to optimize it using the JavaEvA package. The examples mentioned in this chapter can be downloaded from our web site [1].
The examples contain one parameter optimization problem instance for the Evolution Strategies module of JavaEvA, which can also be applied with the Model Assisted Evolution Strategy module, and another parameter optimization problem instance for the Genetic Optimization module. The project also provides exemplary entry points how to start the optimization process for the individual problem instances or the general JavaEvA optimization toolbox either using the main methods of the Java classes or using the additionally provided ant targets in the build.xml file in the ant folder.

4.1 How to install the Project

First, download and uncompress the JavaEvAExample.zip file from our web page to your hard disk. Open a new Java project with your favorite IDE including the /src folder as source and JavaEvA.jar as library. You can start the standard JavaEvA optimization toolbox by executing the main method from JOptExampleGUI.java.
Alternatively, you can use the Ant target 'run' to start the JavaEvA application. This requires ant be installed on your system[2].

4.2 ES Application Example

In this section we will show how to integrate the Evolution Strategies module in an application. First an exemplary fitness function for a toy design problem is given. Then this optimization problem is integrated into the ES module for optimization.

[1]See *http://www-ra.informatik.uni-tuebingen.de/software/JavaEvA/download.html*
[2]See *http://ant.apache.org/* for further details on Apache Ant.

An additional problem specific viewer is implemented and a small application named 'WeldOpt' is build, see fig. 4.1.
It is not necessary to read the whole section about the ES application example, for an experienced user the actual source code of the optimization problem and the alg. 4.1 should be sufficient to understand how to use the EA module immediately.

4.2.1 General Problem Definition

The exemplary optimization problem for the ES module is given by he task to weld a bar to a wall such that it is sturdy but also inexpensive. The dimension of the bar and the thickness of the weld is to be optimized.
Sturdiness is defined by the stress in the structure created by a defined force acting at the end of the bar. The shear stress, bending stress and the deflection at the end of the weld is calculated as follows:

$$\begin{aligned}
\tau &= \sqrt{\tau'^2 + \tau'\tau''\frac{l}{R} + \tau''^2} \\
\tau' &= \frac{P}{\sqrt{2}hl} \\
\tau'' &= \frac{MR}{J} \\
M &= P(L + \frac{l}{2}) \\
R &= \sqrt{\frac{l^2 + (h+t)^2}{4}} \\
J &= \sqrt{s}hl(\frac{l^2}{6} + \frac{(h+t)^2}{2}) \\
\sigma &= \frac{6PL}{t^2 b} \\
\delta &= \frac{6PL^3}{Et^3 b} \\
I &= \frac{tb^3}{12}
\end{aligned}$$

The cost function combines the cost of soldering, an the cost of the bar itself:

$$C = 67412.93h^2 l + 2935.852tb(L + l)$$

Basically this problem definition leads to a multi-objective optimization problem. Although JavaEvA also includes Multi-Objective Evolutionary Algorithms (MOEAs), for the sake of simplicity weight aggregation is used to transform the multi-dimensional objective function into a single objective one. This is done by building a weighted sum of the objectives:

$$f_{sum} = w_C C + w_\tau \tau + w_\delta \delta + w_\sigma \sigma$$

A real world constraint requires the thickness of the weld (h) must be smaller than the thickness of the bar (b). This constraint can be met using a penalty function by adding extremely large values for solutions which violate this constraint:

$$f_{constraints} = \begin{cases} \infty, & h \leq b \\ 0, & else \end{cases}$$

Therefore the final fitness function is given by:

$$f = f_{sum} + f_{constraints}$$

4.2.2 Problem Implementation

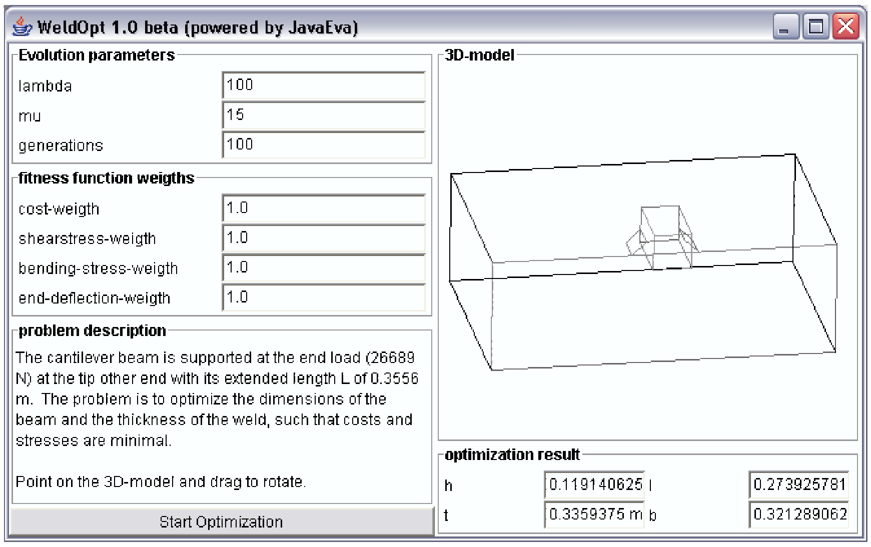
The ES module can handle real-valued optimization problems which extend the AbstractESProblem class.

The AbstractESProblem class for the ES Module

The most important method to be implemented is doEvalution(double[] params). This function returns a real-valued vector (double[]) containing the fitness value(s). In this optimization problem params would contain 4 values: h,l,t and b. doEvalution(params) would return a one-dimensional vector containing the fitness value calculated as described above.
The method GetInputDimension() must return the dimension of the search space of the optimization problem. This is very important for the initialization of the ES, therefore it is necessary to implement it correctly. The corresponding method setInputDimension() sets the search space in case it is variable.
The method equals() checks if another instance of the problem is equal to the actual instance.
Finally, the getName() method returns a descriptive String giving the name of the problem.

Setting Bounds for a real-valued Optimization Problem

Real-valued optimization problem often include upper and lower bounds on the decision variables. It's practical to do this in the constructor of the problem, or in the setInputDimension() method if appropriate. The methods setlower_border(double[] lowerborder) and setupper_border(double[] upperborder) can be used for this purpose. The variables lowerborder and upperborder respectively are real-valued vectors of the same dimension as the search space.



Evolution parameters

lambda	100
mu	15
generations	100

fitness function weigths

cost-weigth	1.0
shearstress-weigth	1.0
bending-stress-weigth	1.0
end-deflection-weigth	1.0

problem description

The cantilever beam is supported at the end load (26689 N) at the tip other end with its extended length L of 0.3556 m. The problem is to optimize the dimensions of the beam and the thickness of the weld, such that costs and stresses are minimal.

Point on the 3D-model and drag to rotate.

optimization result

h	0.119140625	l	0.273925781
t	0.3359375 m	b	0.321289062

Figure 4.1: : The WeldOpt application interface.

Adding a Problem specific Viewer

To visualize the optimization result in an intuitive way a problem specific viewer can be implemented. Since only the optimization problem knows the meaning of the individual decision variables, the optimization problem is the right place to define such a visualization method.
In this example the wall and the attached bar can be visualized as a simple wire frame model. The description of the drawing routines are beyond the scope of this documentation, but there are several alternative Java packages available which are dedicated to sophisticated drawing routines, like Java3D [3].
The most important thing is that the problem decides when it has to be drawn or updated. In the assosicated source code

```
if (fitness < bestfitness) {
..
}
```

can be found in the doEvaluation() method. The best fitness value produced by any call to doEvaluation() method of this instance of the problem is logged. In case a better fitness value is produced the solution can be printed via System.out.print(), drawn in a dedicated frame, opened in a new popup window etc.

[3]See *http://java.sun.com/products/java-media/3D/*.

Listing 4.1: buildESprocessor builds a full parametrized Evolution Strategy

```
public ESProcessorSolo buildESprocessor(
                int lambda, int mu, int maxgenerations,
                ESMutation mutationoperator,
                AbstractESProblem esproblem) {
    ESProcessorSolo esprocessor = new ESProcessorSolo();
    ESPara esparameters = new ESPara();
    final ESPopulation espopulation = new ESPopulation();
    GenerationTerminator Term = new GenerationTerminator();
    Term.setGenerations(maxgenerations);
    TerminatorInterface[] terminators = {Term};
    esparameters.setTerminator(terminators);
    esparameters.setStrategy(espopulation);
    esparameters.setProblem(esproblem);
    espopulation.setLambda(lambda);
    espopulation.setMy(mu);
    espopulation.setModulParameter(esparameters);
    ESIndividual esindividual = new ESIndividual();
    if (mutationoperator instanceof MutationCMA) {
      ((MutationCMA)mutationoperator).setConstraints(true);
    }
    esindividual.setMutation(mutationoperator);
    espopulation.setIndividualTemplate(esindividual);
    espopulation.createInitiMAESPopulation();
    esprocessor.setModulParameter(esparameters);
    return esprocessor;
  }
```

4.2.3 Setting up the ES Application Algorithm

The setup of the ES algorithm is a bit complex because it's high flexibility. After the instantiation of the bare ESProcessor, ESParam, ESPopulation and the GenerationTerminator, these parts have to be initialized and 'connected' in the way and order as shown in the listing 4.1. Some fundamental concepts of modular approach of JavaEvA are shown in this listing. The object ESPara contains all parameters of the Evolutions Strategy module. It contains set() methods for the problem, evolutionary operators, population (a.k.a. strategies), terminators etc. All of these have to be set before the ESProcessor is being initialized.
A terminator watches the optimization process and terminates it, when certain conditions are fulfilled. The GenerationTerminator for example terminates the process after a given number of generations have passed. There are also Terminators which stop the optimization process after given number of function evaluations, or if a

lower fitness limit is reached.
The createInitiMAESPopulation() method in the ESPopulation initializes a initial population of size λ by cloning a template individual λ-times. Every individual is however initialized individually to ensure diversity of the population.
Finally, the optimization process can be run by calling the method optimize() of the ESProcessor.

4.3 Genetic Optimization Application Example

In this section we will explain how to implement your own optimization problem complying with the Genetic Optimization module. You will find the complete source code for this example in the additional JavaEvAExample.zip file for download on our web pages.
We will give an example on how to define the Lens Optimization problem, how to access the problem members via the GUI and how to log and display the optimization results. Finally, we will give you some examples how this optimization problem can be solved using JavaEvA.

4.3.1 General Problem Definition

The Lens Optimization problem is given by the question, how to shape a lens with the radius R to focus as much light as possible on the center of a screen in a distance of F from the lens, see fig. 4.2. The shape of the lens is given by the vector x, which gives the thickness of the lens in n segments. In this case n gives the problem dimension.
To simplify the mathematical equations we assume a thin lens. In that case the target function is given by the sum of the quadratic distances of the projection of each light ray per segment to the center of the screen:

$$f(x) = \sum_{i=1}^{n}((R - S/2) - S(j-1) - \frac{F}{S} \cdot (\epsilon - 1) \cdot (x[i] - x[i-1]))^2 \qquad (4.1)$$

with $R = 5$ the radius of the lens, $F = 20$ the focal length, $\epsilon = 1.5$ a material property and S the length of a segment $S = \frac{2 \cdot R}{n-1}$.
This target function is to be minimized. An optional constraint is to minimize

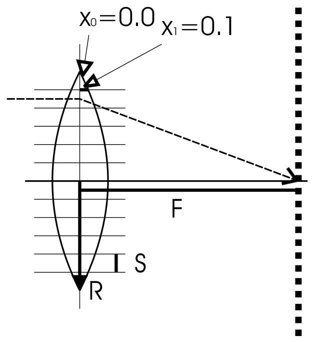


Figure 4.2: Lens Optimization problem.

the required material that can be calculated from the summed segment thickness or even more simple just the thickness in the middle of the lens. The required material is added as penalty to the target function $f(x)$.

$$f'(x) = f(x) + x[x.length/2] \tag{4.2}$$

For practicable results the search space is limited to x: $0.1 \leq x_i \leq 5$.

4.3.2 Problem Implementation

To solve this optimization problem using the GO module of JavaEvA one has to implement an optimization problem complying with the InterfaceOptimizationProblem, allow access to all important problem properties, care for the necessary output and logging methods and finally start the optimization environment to solve the problem. In the following sections we give details on how to implement the problem and how to take care that all the important data is displayed and logged.
To fit into the general framework of JavaEvA an optimization problem has to meet two other requirements too.
First, the optimization problem needs to implement the java.io.Serializable interface. There is nothing special about this, actually one does not need to implement anything for implementing this interface, but all properties of the optimization problem have to be serializable too. In case a property does not meet this requirement one can make this property transient, which prevents this property from being serialized. The java.io.Serializable interface is necessary to allow serialization of the current state of the optimizer and the optimization problem. This allows JavaEvA to restore ones application settings from a previous session and to distribute the optimization over multiple processors.
Second, the Generic Object Editor typically requires an empty constructor, in case a problem needs to load external data or requires proper initialization.

The InterfaceOptimizationProblem for the GO Module

To implement this optimization problem and to optimize it using the GO module the FLensProblem class has to implement the InterfaceOptimizationProblem of the GO module. This interface has three main methods that need to be implemented, see listing 4.2.

Listing 4.2: The GO interface for optimization problems.

```
public interface InterfaceOptimizationProblem {
  /** This method inits the Problem to log multi runs.*/
  public void initProblem();
  /** This method inits a given population.
   * @param population    The populations that is to be inited
   */
  public void initPopulation(Population population);
```

```
    /** This method evaluates a given population and set the
     * fitness values accordingly.
     * @param population    The population that is to be evaluated.
     */
    public void evaluate(Population population);
    /** This method evaluates a single individual and sets the
     * fitness values accordingly.
     * @param individual    The individual that is to be evaluated
     */
    public void evaluate(AbstractEAIndividual individual);
    .
    .
}
```

These are the three basic methods that are necessary to start an optimization process. Of course it is necessary to initialize the problem and to initialize the primordial population of solutions. The initialization of the population is necessary to specify the data type to use, to set the necessary bounds of the search space and to allow problem specific initialization of the individuals. The two evaluation methods are either on the level of individuals or on the level of populations. An optimization strategy usually uses the population based evaluation method. The evaluation of a population can simply be implemented as repetitive evaluation of each individual, but it also allows for coevolutionary environments, where the individuals may interact and perhaps even compete for limited resources. The population based method further allows you to log problem specific data like the best individual found so far, the history of the best solutions per generation or to take care of plotting the best solution. In the individual based evaluation method one can also take care of storing a sensible solution representation, e.g. after repair or normalization mechanisms have been applied, to allow easier access to the true solution representation.
Please note that the problem has to take care to increment the number of fitness evaluations. This is necessary because the problem is the only place that can decide whether or not it is able to take a short cut, because two individuals are very much alike, or if additional fitness evaluation are necessary because of an additional local search step.

Implementing the InterfaceOptimizationProblem

The first thing to implement should be the initialization methods. The init() method of the problem is of course problem specific and in this case limited to reseting the overall best result to null and to initializing the problem specific viewer.
The initPopulation() method has to take care that the individuals in the initial population are suitable formatted, see listing 4.3. This includes to select the proper data type for the individuals and mutation and crossover operators. This is usually done before initPopulation() is called by parameterizing the template individual accord-

ingly using the GUI or by direct access. But still the problem has the final decision on how to parameterize the template individual. In this example the problem sets the problem dimension to the template individual and also sets a suitable range for each decision parameter, which could also be set independently.
Next the population is cleared and each individual cloned and initialized from the template individual. In this example the default init() method is used on each individual, but you could also perform a problem specific initialization on each individual or use a D-optimal design for the whole population. Please note that the size of the population is usually not set from within the problem but somewhere else.
Finally, the problem viewer is initialized and again an init method is called on the population, which also uses the default init method on each individual. This command needs to be removed in case of a problem specific initialization, else this command would render the previous initialization useless.

Listing 4.3: The initPopulation(population) method for the FLensProblem.

```java
public void initPopulation(Population population) {
 AbstractEAIndividual tmpIndy;
 this.m_OverallBest = null;
 this.m_Template.setDoubleDataLength(this.m_ProblemDimension);
 // set the range
 double[][] range = new double[this.m_ProblemDimension][2];
 for (int i = 0; i < range.length; i++) {
  range[i][0] = 0.1;
  range[i][1] = 5.0;
 }
 this.m_Template.SetRange(range);
 population.clear();
 for (int i = 0; i < population.getPopulationSize(); i++) {
  tmpIndy = (AbstractEAIndividual)
    ((AbstractEAIndividual)this.m_Template).clone();
  tmpIndy.init(this);
  population.add(tmpIndy);
 }
 if (this.m_Show) this.initProblemFrame();
 population.init();
}
```

The evaluation method for populations basically calls the evaluation method for individuals on each individual of the population, see listing 4.4, and increments the functions calls for the population accordingly. Finally, some problem specific data logging or statistics could be performed. This example limits to updating the problem specific viewer.

Listing 4.4: The evaluate(population) method for the FLensProblem.

```
public void evaluate(Population population) {
 AbstractEAIndividual    tmpIndy;
 for (int i = 0; i < population.size(); i++) {
  tmpIndy = (AbstractEAIndividual) population.get(i);
  this.evaluate(tmpIndy);
  population.incrFunctionCalls();
 }
 if (this.m_Show) this.updateProblemFrame(population);
}
```

The evaluation method for individuals basically implements equ. 4.1, although it may seem a bit more complicated than the equation. But this is simply due to the fact that the computation is separated into two methods, see listing 4.5. The only problem worth mentioning is, that it is necessary to cast the objects stored in the population to AbstractEAIndividuals to set the fitness values and perhaps the UserDefinedObjects. Further on it is necessary to cast the AbstractEAIndividual to the proper InterfaceDataTypeX. For the real-valued lens problem this is the InterfaceDataTypeDouble, to access the problem specific decision variables.

Listing 4.5: The evaluate(individual) method for the FLensProblem.

```
public void evaluate(AbstractEAIndividual individual) {
 double[]          x;
 double[]          fitness;
 x = new double[((InterfaceDataTypeDouble) individual)
  .getDoubleData().length];
 System.arraycopy(((InterfaceDataTypeDouble) individual)
  .getDoubleData(), 0, x, 0, x.length);
 for (int i = 0; i < x.length; i++) x[i] = x[i] - this.m_XOffSet;
 fitness = this.doEvaluation(x);
 for (int i = 0; i < fitness.length; i++) {
  // add noise to the fitness
  fitness[i] += RandomNumberGenerator.gaussianDouble(this.m_Noise);
  fitness[i] += this.m_YOffSet;
  // set the fitness of the individual
  individual.SetFitness(i, fitness[i]);
 }
 if ((this.m_OverallBest == null) ||
  (this.m_OverallBest.getFitness(0) > individual.getFitness(0))) {
  this.m_OverallBest = (AbstractEAIndividual)individual.clone();
 }
}

public double[] doEvaluation(double[] x) {
 double fitness         = 0;
```

```
  double[] ret              = new double[1];
  // set a minimum value for the thickness of the lens
  for (int i = 0; i < x.length; i++) if (x[i] < 0.1) x[i] = 0.1;
  double[] tmpFit = this.testLens(x);
  for (int i = 0; i < tmpFit.length; i++)
    fitness += Math.pow(tmpFit[i], 2);
  if (this.m_UseMaterialConst)
    fitness = fitness + x[(int)(x.length/2)];
  ret[0] = fitness;
  return ret;
}

public double[] testLens(double[] x) {
  double      m_SegmentHight      = 2 * m_Radius / (x.length - 1);
  double[]    result              = new double[x.length-1];
  // Computation of fitness. Uses an approximation for very thin
  // lenses. The fitness is the sum over all segments of the
  // deviation from the center of focus of a beam running through
  // a segment.
  for (int i = 1; i < x.length; i++)
   result[i-1] = m_Radius - m_SegmentHight / 2 - m_SegmentHight
     * (i - 1) - m_FocalLength / m_SegmentHight * (m_Epsilon - 1)
     * (x[i] - x[i-1]);
  return result;
}
```

Adding Problem Members to the GUI

To enable the GenericObjectEditor to display general information about your problem and to allow editing of basic members of your problem definition you need to implement just a few methods, see listing 4.6. The getName() method returns the name that the GenericObjectEditor will use to name your class. The globalInfo() will return a text that the GenericObjectEditor will display at the top of the GUI frame as short description. Please limit to just a few words since the GenericObjectEditor often cuts the text short.
To allow easy GUI access to the basic problem properties you simply need to add get and set methods for each member. Do to so you have to obey a simple naming convention: the methods have to be named 'getMemberName' and 'setMemberName', 'MemberName' and the data type must be the same for both methods. To add a hint text implement a method called 'memberNameTipText' returning a descriptive string. This method again has to have the same string as member name but needs to start with a small letter and end with 'TipText'. See for example the implementation in listing 4.6 for the amount of noise on the target function evaluation.

This way you can add arbitrary variables to the GUI as long as the members are of a primitive data type that is known to the GenericObjectEditor, see right hand side of fig. 4.3 for the GUI element resulting for the FLensProblem class from the GenericObjectEditor.

Listing 4.6: Example how give a short description for the class in the GUI and how to access the class properties in the GUI

```
/** This method allows the GenericObjectEditor to
 * read the name of the current object.
 * @return The name.
 */
public String getName() {
 return 'Lens Problem';
}
/** This method returns a global info string
 * @return description
 */
public String globalInfo() {
 return 'Focusing of a lens is to be optimized.';
}
/** This method allows you to choose how much noise is to
 * be added to the fitness.
 * @param noise      The sigma for a gaussian random number.
 */
public void setNoise(double noise) {
 if (noise < 0) noise = 0;
 this.m_Noise = noise;
}
public double getNoise() {
 return this.m_Noise;
}
public String noiseTipText() {
 return 'Noise level on the fitness value.';
}
```

Additional Output and Logging Methods

The InterfaceOptimizationProblem also requests additional methods to output the result of the optimization process, see listing 4.7.
For example the getSolutionRepresentationFor() returns a possibly multi-line string that is a readable and easy understandable solution representation. The getStringRepresentation() methods returns a similar string for the optimization problem an its parameter. The drawIndividual() may open a new JFrame to show a graphical representation of the solution, while the getDoublePlotValue() returns a double value

that will be logged and averaged over several multi runs by the JavaEvA Statistics class, typically the target value.
The getAdditionalFileStringHeader() and the getAdditionalFileStringValue() allow you to log problem specific data per generation over multiple runs in a result file that is written by JavaEvA. The first method returns the tab separated header, while the second method returns the tab separated values.
Finally, the method getFinalReportOn() allows you to output a final resume on the performance and the result of the optimization process.
Although you can implement all these methods, most likely it would be best if you take care of the problem specific logging and statistic yourself, since we can not foresee all the possible requirements regarding documentation of the optimization process that your specific optimization problem needs.

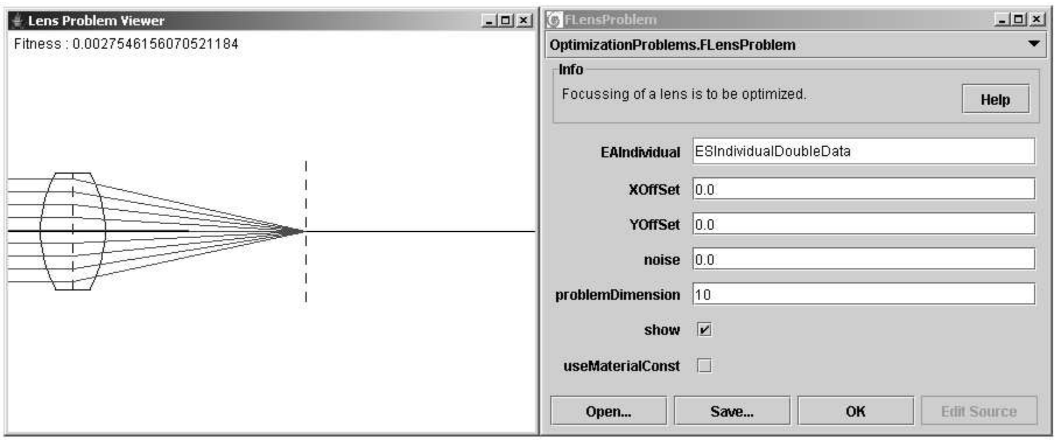
Listing 4.7: Additional output and logging method requested from the InterfaceOptimizationProblem

```java
/** This method allows you to output a string that describes a
 * found solution in a way that is most suitable for a given problem.
 * @param individual    The individual that is to be shown.
 * @return The description.
 */
public String getSolutionRepresentationFor(AbstractEAIndividual
   individual);
/** This method returns a string describing the optimization problem.
 * @return The description.
 */
public String getStringRepresentation();
/** Request a graphical representation for a given individual.
 */
public void drawIndividual(AbstractEAIndividual indy);
/** This method returns a double value that will be displayed in
 * a fitness plot. A fitness that is to be minimized with a global
 * min of zero would be best, since log y can be used. But the
 * value can depend on the problem.
 * @param pop   The population that is to be refined.
 * @return Double value
 */
public Double getDoublePlotValue(Population pop);
/** This method returns the header for the additional data that
 * is to be written into a file.
 * @param pop   The population that is to be refined.
 * @return String
 */
public String getAdditionalFileStringHeader(Population pop);
```

```
/** This method returns the additional data that is to be written
 * into a file.
 * @param pop   The population that is to be refined.
 * @return String
 */
public String getAdditionalFileStringValue(Population pop);
/** This method allows you to output a string that describes
 * the overall performance of the optimization process.
 * @param optimizer         The individual that is to be shown.
 * @return The description.
 */
public String getFinalReportOn(InterfaceOptimizer optimizer);
```

Adding a Problem specific Viewer

In this exemplary problem implementation we also added a problem specific viewer to the optimization problem by using the three methods initProblemFrame(), disposeProblemFrame() and updateProblemFrame(). Without going much into detail regarding the capabilities of Java for 2D or even 3D visualization, this example should help you to implement your own problem specific and customer orientated visualization. See left hand side of fig. 4.3 for the resulting problem viewer. The viewer displays the overall best individual found so far, the achieved target value, a graphical representation of the optimized lens and the resulting path of the individual light rays for each segment.



Fitness : 0.0027546156070521184

OptimizationProblems.FLensProblem

Info

Focussing of a lens is to be optimized.

EAIndividual ESIndividualDoubleData

XOffSet 0.0

YOffSet 0.0

noise 0.0

problemDimension 10

show ☑

useMaterialConst ☐

Figure 4.3: The Lens Optimization problem GUI and the problem specific viewer.

Adding the new Problem to the GUI

To integrate the new problem into the generic JavaEvA GUI several additional steps are necessary. First, it is necessary to add the new problem instance to JavaEvAGUI_Lite.props under InterfaceOptimizationProblem. Please, take care that no extra spaces are introduced behind the individual problem instances. This may cause the scanning algorithm to terminate prematurely. Make also sure to edit the JavaEvAGUI_Lite.props file that is in the working path of JavaEvA.
Second, it is necessary for the GenericObjectEditor that you provide your new classes with an empty constructor. Otherwise the GenericObjectEditor will not be able to to create a new instance of this object.
Finally, please note that due to the GenericObjectEditor any problems occurring during the initialization of the new object instance selected via the GUI will not be reported correctly. The GenericObjectEditor will terminate the new object instance notifying with a simple 'Cannot create an example of..' statement.

4.3.3 Optimizing the Problem

After having implemented the optimization problem you want to optimize it. Since we have implemented a problem for the GO module we will show a short cut how to start the GUI for the GO module with your optimization problem.
To do so you have to make a new instance of the GOStandaloneVersion, get the parameters of the optimization problem, which are basically the same as described in sec. 3.9 and set the optimization problem to the implemented FLensProblem, see listing 4.8 for details. Now you can visualize the GUI by using the initFrame() and setShow(true) command on the optimization module. Alternatively, you could start the optimization process directly by calling the doWork() method.
Instead of setting just the problem, you could also choose and parameterize the optimization algorithm used in the GO module. You could also parameterize the problem regarding EA representation type, mutation/crossover operators and rates on the programming level. You have access to all properties of the GO module discussed in sec. 3.9.

Listing 4.8: Example how start the GUI of the GO module to optimize your FLensProblem

```
public static void main(String[] args) {
  System.out.println('TESTING THE FLENSPROBLEM:');
  System.out.println('Working Dir ' +System.getProperty('user.dir'));
  FLensProblem        f       = new FLensProblem();
  GOStandaloneVersion program = new GOStandaloneVersion();
  GOParameters        GO      = program.getGOParameters();
 GO.setProblem(f);
```

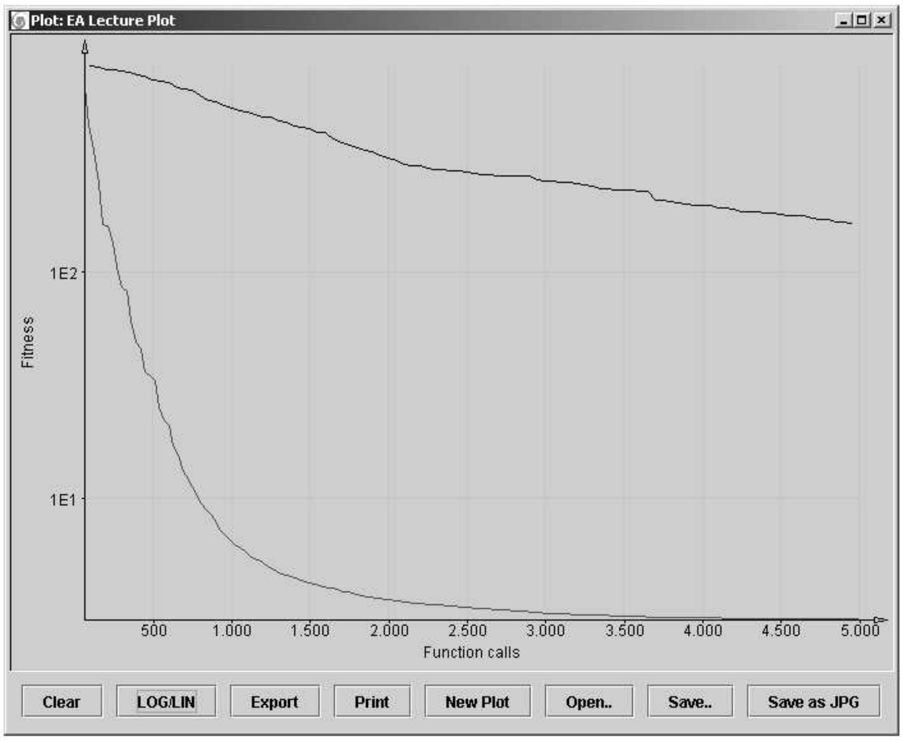


Figure 4.4: Comparing the performance of a GA with a binary genotype individual (upper plot) and a $(5, 30)$-ES with a real-valued individual and global mutation (lower plot) regarding the averaged best results over 20 multi-runs.

```
  RandomNumberGenerator.setseed(1);
  program.initFrame();
  program.setShow(true);
}
```

If you choose to use the GUI version you can immediately start to optimize the lens problem and comparing the behavior of the different EA approaches, the impact of different solution representations and mutation/crossover operators and rates. In fig. 4.4 you can see the result of comparing a standard GA with a binary genotype representation to a $(5, 30)$-ES with a real-valued representation. The real-valued ES clearly outperforms the GA.

Chapter 5

JavaEvA Frequently Asked Questions (FAQ)

In this chapter we will try to give some answers to frequently asked questions related to JavaEvA but also to the full version JOpt. We will not necessarily distinguish between these to versions since JavaEvA is a subset of JOpt and we expect most algorithms currently under development in JOpt will move to JavaEvA as soon as they are validated and tested at large. Some of the question may also be related to the 'developer edition' (source code of JavaEvA) which is available at request.
This list of frequently asked questions is far from being complete, but is updated continuously.

Q: How to install the Project?

First, download and uncompress the JavaEvAExample.zip file from our web page[1] to your hard disk. Open a new Java project with your favorite IDE including the /src folder as source and JavaEvA.jar as library. You can start the standard JavaEvA optimization toolbox by executing the main method from JOptExampleGUI.java.
Alternatively, you can use the Ant target 'run' to start the JavaEvA application. This requires ant be installed on your system [2].

Q: JavaEvA exits on start with NullPointerException in Toolkit.createImage()?

```
java.lang.NullPointerException
   at java.awt.Toolkit.createImage(Unknown Source)
   at javaeva.client.EvAClient.createSplashScreen(EvAClient.java:445)
   at javaeva.client.EvAClient.<init>(EvAClient.java:348)
   at javaeva.client.EvAClient.main(EvAClient.java:462)
 Exception in thread main
```

[1]See *http://www-ra.informatik.uni-tuebingen.de/software/JavaEvA/*

[2]See *http://ant.apache.org/* for further details on Apache Ant.

In the developer edition it is necessary to copy the resource folder into the working directory. This can be done manually or by using the 'compile' or 'compileAll' command from the provided ant file.

Q: I've made my own problem instance but it doesn't show up in the GUI?

It is necessary to add the new problem instance to JavaEvAGUI_Lite.props (JavaEvAGUI.props) under InterfaceOptimizationProblem. Please, take care that no extra spaces are introduced behind the individual problem instances. This may cause the scanning algorithm to terminate prematurely. Again make sure to edit the JavaEvAGUI_Lite.props (JavaEvAGUI.props) file that is in the working path of JavaEvA.

Q: The new problem instance shows in the GUI but when selected an error occurs?

The GenericObjectEditor requires an empty constructor for the new problem instance, else it is not able to create a new instance. Additionally, in case the constructor throws any exceptions the GenericObjectEditor will terminate the new object instance notifying with a simple 'Cannot create an example of..' statement.

Q: I've implemented my own problem for GO and the Optimizers are behaving strange not like they behave on the test functions?

There are two possible reasons for this. First, you need to take care that the populations are initialized correctly in the initPopulation() method. You need to select a suitable data representation, set appropriate data ranges and initialize the individuals with random values. Even if you prefer to initialize the individual with problem specific initial values, it is advisable not to initialize all individuals with the same values. Otherwise, the optimizers lack the initial diversity to explore the search space efficiently.
Second, perhaps you have overridden the getDoublePlotValue() method to log your own data. Typically, such data is gathered during the evaluate() method. But unfortunately not all individuals of the current population may enter the evaluate() method. For example in case of a GA with elitism, the elite individuals are not reevaluated, This may cause and elite GA the look like a GA without elitism. Therefore, it is advisable to perform the required statistics on the population given as parameter for the getDoublePlotValue() method instead on the population encountered during the evaluate() methods.

Q: I want to use the online source code editing method for the User Defined Problem, but it doesn't work?

That is a complicated problem. Despite on going efforts to make the online source code editing option as user friendly as possible, there are several pitfalls with this feature of JavaEvA.

1. Make sure that there is no compiled *.class file in the class path of the Java object you want use the source code edition feature on.
2. To enable the source code editor to find the actual source code of the object, make shure to start JavaEvA from the correct directory, where it can find the source. For example, if the project is installed in 'C:/JavaEvA/', the complied classes are in 'C:/JavaEvA/build/' and the source code is in 'C:/JavaEvA/src/', start JavaEvA in 'C:/JavaEvA/' using:
   java -cp 'C:/JavaEvA/build/' javaeva.client.EvAClient.
3. Finally, after the first conditions are met another problem can occur. Without a proper PATH variable pointing to 'javac' JavaEvA may be unable to compile the edited source code. Therefore, make sure alter the PATH variable to point to the bin directory of your JAVA SDK version.

In case it still doesn't work, it could be related to general compilation errors of the edited source code. Please remember that JavaEvA is no valid substitute for a full IDE. We made no special provision in JavaEvA to deal with syntax errors or online compilation errors.

Acknowledgments

This research and software development has been funded by the German federal ministry of research and education (BMBF) in the project "Entwicklung eines Systems zur automatisierten Herstellung und Charakterisierung von kristallinen Festkörpern in hohem Durchsatz" under contract No. 03C0309E and by the ALTANA Pharma AG, Konstanz, Germany.

Bibliography

[1] S. Baluja. An empirical comparison of seven iterative and evolutionary function optimization heuristics. Technical Report CMU-CS-95-193, 1995.

[2] S. Baluja and R. Caruana. Removing the genetics from the standard genetic algorithm. In A. Prieditis and S. Russel, editors, *The International Conference on Machine Learning*, pages 38–46, San Mateo, CA, 1995. Morgan Kaufmann Publishers.

[3] J. B. Beasley. OR-Library: distributing test problems by electronic mail. *Journal of the Operational Research*, 8:429–433, 1996.

[4] N. Cramer. A representation for the adaptive generation of simple sequential programms. In Grefenstette, editor, *Proceedings of the 1st International Conference on Genetic Algorithms*, pages 183–187, 1985.

[5] K. De Jong. An analysis of the behavior of a class of genetic algorithms. (Doctoral dissertation, University of Michigan) Dissertation Abstracts International, 36(10), 5140B. (University Microfilms No. 76-9381), 1975.

[6] K. Deb and D. Goldberg. An investigation of niche and species formation in genetic function optimization. In M. Kaufmann, editor, *Proceedings of the 5th International Conference on Genetic Algorithms and their Applications*, pages 42–50, 1989.

[7] M. A. El-Beltagy and A. Keane. Evolutionary optimization for computationally expensive problems using gaussian processes. In *Proceedings of the International Conference on Artificial Intelligence IC-AI'2001*, pages 708–714. CSREA Press, 2001.

[8] M. Emmerich, A. Giotis, M. Özdemir, K. Giannakoglou, and T. Bäck. Metamodel assisted evolution strategies. In *Parallel Problem Solving from Nature VII*, pages 362–370. Springer, 2002.

[9] L. Eshelman. The CHC adaptive search algorithm: How to have safe search when engaging in nontraditional genetic recombination. *Foundations of Genetic Algorithms*, 1:265–283, 1991.

[10] L. J. Eshelman and J. D. Schaffer. Real-coded genetic algorithms and interval-schemata. In L. D. Whitley, editor, *Foundations of Genetic Algorithms 2*, pages 187–202. Morgan Kaufmann, San Mateo, CA, 1993.

[11] R. Friedberg. A learning machine: Part I. *IBM Journal of Research and Development*, pages 2–13, 1958.

[12] R. Friedberg, B. Dunham, and J. North. A learning machine: Part II. *IBM Journal of Research and Development*, 3(3):282–287, 1959.

[13] C. Fujiki. Using the genetic algorithm to generate lisp source code to solve the prisoner's dilemma. In *International Conference on Genetic Algorithms*, pages 236–240, 1987.

[14] M. Gallagher, M. Frean, and T. Downs. Real-valued evolutionary optimization using a flexible probability density estimator. In W. B. et al., editor, *Proceedings Genetic and Evolutionary Computation Conference*, pages 840–846, San Francisco, CA, 1999. Morgan Kaufmann Publishers.

[15] D. Goldberg and J. Richardson. Genetic algorithms with sharing for multimodal function optimization. In Grefenstette, editor, *Proceedings of the 2nd International Conference on Genetic Algorithms*, pages 41–49, 1987.

[16] N. Hansen and A. Ostermeier. Adapting arbitrary normal mutation distributions in evolution strategies: The covariance matrix adaption. In *Proceedings of the 1996 IEEE International Conference on Evolutionary Computation*, pages 312–317, 1996.

[17] J. He and X. Yao. Towards an analytic framework for analysing the computation time of evolutionary algorithms. *Artificial Intelligence*, 145(1-2):59–97, 2003.

[18] J. Holland. *Adaption in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Systems.* The University Press of Michigan Press, Ann Arbor, 1975.

[19] Y. Jin, M. Olhofer, and B. Sendhoff. A framework for evolutionary optimization with approximate fitness functions. *IEEE Transactions on Evolutionary Computation*, 6(5):481–494, 2002.

[20] K. A. D. Jong and J. Sarma. Generation gaps revisited. In D. Whitley, editor, *Foundations of Genetic Algorithms*, volume 2, pages 19–28. Morgan Kaufmann, 1992.

[21] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.

[22] J. R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* MIT Press, Cambridge, MA, USA, 1992.

[23] J. R. Koza. *Genetic Programming II: Automatic Discovery of Reusable Programs.* MIT Press, Cambridge Massachusetts, May 1994.

[24] J. R. Koza, David Andre, F. H. Bennett III, and M. Keane. *Genetic Programming III: Darwinian Invention and Problem Solving.* Morgan Kaufman, Apr. 1999.

[25] P. Larranaga. A review on estimation of distribution algorithms. In P. Larranaga and J. Lozano, editors, *Estimation of Distribution Algorithms*, chapter 3, pages 57–100. Kluwer Academic Publisher, 2002.

[26] H. Mühlenbein, J. Bendisch, and H.-M. Voigt. From recombination of genes to the estimation of distributions II. continuous parameters. In *4th International Conference on Parallel Problem Solving from Nature*, pages 188–197, 1996.

[27] H. Mühlenbein and G. Paass. From recombination of genes to the estimation of distributions I. binary parameters. In *4th International Conference on Parallel Problem Solving from Nature*, pages 178–187, 1996.

[28] Z. Michalewicz. *Genetic Algorithms + Data Structures = Evolutions Programs.* Springer-Verlag, New York, 1992.

[29] M. Mitchell, J. H. Holland, and S. Forrest. When will a genetic algorithm outperform hill climbing. In J. D. Cowan, G. Tesauro, and J. Alspector, editors, *Advances in Neural Information Processing Systems*, volume 6, pages 51–58. Morgan Kaufmann Publishers, Inc., 1994.

[30] H. Mühlenbein and D. Schlierkamp-Voosen. Predictive models for the breeder genetic algorithm I: Continuous parameter optimization. *Evolutionary Computations*, 1(1):25–49, 1993.

[31] M. Pelikan, D. Goldberg, and F. Lobo. A survey of optimization by building and using probabilistic model. Technical Report 99018, IlliGAL, Septemper 1999.

[32] M. Pelikan, D. E. Goldberg, and E. Cantú-Paz. BOA: The Bayesian optimization algorithm. In W. Banzhaf, J. Daida, A. E. Eiben, M. H. Garzon, V. Honavar, M. Jakiela, and R. E. Smith, editors, *Proceedings of the Genetic and Evolutionary Computation Conference*, volume I, pages 525–532, Orlando, FL, 13-17 1999. Morgan Kaufmann Publishers, San Fransisco, CA.

[33] J. Poland and A. Zell. Main vector adaptation: A CMA variant with linear time and space complexity. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 1050–1055. Morgan Kaufman, 2001.

[34] N. J. Radcliffe. Equivalence class analysis of genetic algorithms. *Complex Systems*, 5(2):183–205, 1991.

[35] I. Rechenberg. *Evolutionsstrategie: Optimierung technischer Systeme nach Prinzipien der biologischen Evolution.* Fromman-Holzboog, Stuttgart, Germany, 1973.

[36] R. P. Salustowicz and J. Schmidhuber. Probabilistic incremental program evolution: Stochastic search through program space. In M. van Someren and G. Widmer, editors, *Machine Learning: ECML-97*, volume Lecture Notes in Artificial Intelligence 1224, pages 213–220. Springer-Verlag, 1997.

[37] K. Sastry and D. E. Goldberg. Probabilistic model building and competent genetic programming. In R. L. Riolo and B. Worzel, editors, *Genetic Programming Theory and Practise*, chapter 13, pages 205–220. Kluwer, 2003.

[38] H.-P. Schwefel. *Numerical Optimization of Computer Models.* John Wiley & Sons, Chichester, U.K., 1977.

[39] H.-P. Schwefel. *Evolution and Optimum Seeking.* John Wiley & Sons, New York, 1995.

[40] M. Sebag and A. Ducoulombier. Extending population-based incremental learning to continuous search spaces. In H.-P. Schwefel, editor, *5th International Conference on Parallel Problem Solving from Nature*, pages 418–427, Amsterdam The Netherlands, Mai 1998. 27-30 september.

[41] F. Streichert, H. Ulmer, and A. Zell. Evolutionary algorithms and the cardinality constrained portfolio selection problem. In D. Ahr, R. Fahrion, M. Oswald, and G. Reinelt, editors, *Operations Research Proceedings 2003, Selected Papers of the International Conference on Operations Research (OR 2003), Heidelberg, September 3-5, 2003.* Springer, 2003.

[42] G. Syswerda. Uniform crossover in genetic algorithms. In *Proceedings of the Third International Conference on Genetic Algorithms*, pages 1–9. Morgan Kaufmann, 1989.

[43] G. Syswerda. Simulated crossover in genetic algorithms. In D. Whitley, editor, *Foundations of Genetic Algorithms*, volume 2, pages 239–255. Morgan Kaufmann, 1993.

[44] S. Tsutsui. Probabilistic model-building genetic algorithms in permutation representation domain using edge histogram. In *Proceedings of the 7th International Conference on Parallel Problem Solving from Nature*, pages 224–233. Springer-Verlag, 2002.

[45] H. Ulmer, F. Streichert, and A. Zell. Model-assisted steady-state evolution strategies. In E. Cantú-Paz, J. A. Foster, K. Deb, D. Davis, R. Roy, U.-M. O'Reilly, H.-G. Beyer, R. Standish, G. Kendall, S. Wilson, M. Harman, J. Wegener, D. Dasgupta, M. A. Potter, A. C. Schultz, K. Dowsland, N. Jonoska, and J. Miller, editors, *Genetic and Evolutionary Computation – GECCO-2003*, volume 2723 of *LNCS*, pages 610–621, Chicago, 12-16 July 2003. Springer-Verlag.

[46] J. Wakunda. *Parallele Evolutionsstrategien mit der Optimierungsumgebung EvA*. PhD thesis, Fakultät für Informatik, Eberhard-Karls-Universität Tübingen, 2001.

[47] J. Wakunda and A. Zell. Eva - a tool for optimization with evolutionary algorithms. In *Proceedings of the 23rd EUROMICRO Conference*, Budapest, Hungary, September 1-4 1997.

[48] A. H. Wright. Genetic algorithms for real parameter optimization. In G. J. Rawlins, editor, *Foundations of genetic algorithms*, pages 205–218. Morgan Kaufmann, San Mateo, CA, 1991.

[49] T. Yu and P. Bentley. Methods to evolve legal phenotypes. In A. E. Eiben, T. Bäck, M. Schoenauer, and H.-P. Schwefel, editors, *Fifth International Conference on Parallel Problem Solving from Nature*, pages 280–291, Amsterdam, 1998. Springer.