

# Visualization of Evolutionary Algorithms

Master Thesis  
im Fach Computational Engineering

vorgelegt von

Zoltán Tóth  
geboren am 17. Mai 1976 in Dunaújváros, Ungarn

anfertigt am

Institut für Informatik  
Lehrstuhl für Programmiersprachen  
FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG  
(*Prof. Dr. H. J. Schneider*)

Betreuer: Dr.-Ing. Gabriella Kókai, PD Dr.-Ing. Mark Minas

Beginn der Arbeit: 03.12.2001  
Abgabe der Arbeit: 03.06.2002

Ich versichere, daß ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als angegebenen Quellen anfertigt habe und daß die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Ich bin damit einverstanden, daß die Arbeit veröffentlicht und daß in wissenschaftlichen Veröffentlichungen auf sie Bezug genommen wird.

Erlangen, 29. 5. 2002

# Aufgabenstellung

## Background:

Engineering applications provide a wide range of optimization problems for researchers working in this area. In most cases, the different tasks require different programming environments to achieve the best results.

Evolutionary algorithms (EAs) are general purpose function optimization methods which search for optima by making potential solutions (individuals) compete for survival in a population. The better an individual is, the better chance it has to survive. The search space is explored by modifying the potential solutions by genetic operators observed in nature: generally mutation and recombination.

The *GEA* (Generic Evolutionary Algorithms) system contains implementation of various evolutionary algorithms, genetic operators for the most common representation forms of individuals, various selection methods, and examples on the usage and expansion possibilities of the library.

## Task:

The task of the student is to design and implement a graphical user interface (GUI) called *GraphGEA* to the *GEA* system. The basic requirements are the following:

- *GEA* should stay independent of *GraphGEA*
- management (setting, checking, saving, restoring) of the parameters of the evolutionary algorithm
- interactive execution of the evolution process: start/stop, suspend/resume, step-wise execution etc.
- visualization of the results of the evolution process as plots, tables, figures etc.
  - course visualization: fitness graphs, best individuals of the generations (genotype and phenotype) etc.
  - state visualization: all individuals of the current generation (genotype and phenotype)
- on-line and off-line visualization

## Literature:

- C. Jacob. *Principia Evolvica – Simulierte Evolution mit Mathematica*. Dpunkt Verlag, 1997
- J. R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, Massachusetts, 1992
- M. Mitchell. *An Introduction to Genetic Algorithms*. MIT Press, Cambridge, Massachusetts, 1996
- I. Rechenberg. *Evolutionsstrategien: Optimierung Technischer Systeme nach Prinzipien der Biologischen Evolution*. Fromann-Holzboog, Stuttgart, 1973
- Z. Tóth, G. Kókai. *An Evolutionary Optimum Searching Tool* In proceedings of IEA/AIE 2001, LNAI 2070 pp. 19-24, Springer-Verlag, Berlin

## Abstract

The goal of this thesis is to design and develop an easy-to-use graphical user interface to the *Generic Evolutionary Algorithms* (*GEA*) programming library. The implemented user interface is capable of visualizing various aspects of the running evolution process. Further objectives are to provide a tool for controlling the evolution process interactively and a simple interface for setting the parameters of the optimization. The implemented software, called *GraphGEA*, realizes the visualization by reading and processing the log files of *GEA* and passing this information to the loaded displaying plug-ins. A data structure and its graphical extension had also been implemented to hold the often strongly dependent parameters of the evolution process. This data structure has a clear EBNF-definition and thus is easy to extend. A novelty of *GraphGEA* is the applied plug-in technology which makes the extension of the system with new visualization tools very easy. The evolution process is running as a child-process of the GUI and they communicate with pipes, signals and through a shared memory area.

## Zusammenfassung

Das Ziel dieser Arbeit ist, eine einfach benutzbare grafische Oberfläche zu der *Generic Evolutionary Algorithms* (*GEA*) Programmbibliothek zu planen und entwickeln. Die implementierte Benutzeroberfläche ist fähig, die verschiedenen Aspekte des laufenden Evolutionsprozesses zu visualisieren. Weitere Zielsetzungen sind, ein Werkzeug für die interaktive Steuerung des Evolutionsprozesses und eine einfache Schnittstelle für die Einstellung der Optimierungsparameter zur Verfügung zu stellen. Die eingeführte Software, *GraphGEA* genannt, verwirklicht die Visualisierung durch das Lesen und die Verarbeitung der Protokolldateien von *GEA* und die Weitergabe dieser Informationen an die geladenen Anzeigeplugins. Eine Datenstruktur und ihre graphische Erweiterung sind auch eingeführt worden, um die häufig stark voneinander abhängigen Parameter des Evolutionsprozesses zu speichern. Diese Datenstruktur hat eine saubere EBNF-Definition und ist folglich leicht zu erweitern. Eine Neuheit von *GraphGEA* ist die angewandte Plugintechnologie, die die Erweiterung des Systems um neue Visualisierungswerkzeuge sehr einfach macht. Der Evolutionsprozess wird in einem Kind-Prozess des GUI durchgeführt und die Kommunikation zwischen den beiden Programmen ist mittels Pipes, Signalen und eines gemeinsam genutzten Speicherbereich realisiert.

# Contents

<b>Introduction</b>	<b>3</b>
<b>1 Evolutionary Algorithms</b>	<b>5</b>
1.1 Genetic Algorithms . . . . .	6
1.1.1 GA Mutation . . . . .	7
1.1.2 GA Crossover . . . . .	7
1.1.3 Creation of the New Generation . . . . .	7
1.1.4 Selection Methods . . . . .	8
1.2 Genetic Programming . . . . .	12
1.2.1 GP Mutation . . . . .	12
1.2.2 GP Crossover . . . . .	13
1.3 Evolution Strategies . . . . .	14
1.3.1 ES Mutation . . . . .	14
1.3.2 ES Recombination . . . . .	14
1.3.3 Creation of the New Generation . . . . .	15
1.3.4 Meta-ES . . . . .	16
1.4 The Theoretical Basis of Probability Adaptation . . . . .	16
1.5 Possibilities of Visualization . . . . .	17
<b>2 The e_params Data Structure</b>	<b>19</b>
2.1 Requirements on the Data Structure . . . . .	19
2.2 Implementation . . . . .	20
2.2.1 Data Types . . . . .	21
2.2.2 Functions . . . . .	22
2.3 Input and Output Files . . . . .	24
2.3.1 Definition of a Parameters Structure . . . . .	24
2.3.2 Storing the Parameter Values . . . . .	28
2.3.3 Listing a Set of Parameters . . . . .	28
2.4 The Graphical Extension . . . . .	28
2.5 Final Notes . . . . .	31
<b>3 The GEA System</b>	<b>33</b>
3.1 The Structure of GEA . . . . .	34
3.1.1 Class Evolvable and the Representation Forms . . . . .	34
3.1.2 Representation of a Population . . . . .	35
3.1.3 Selection Methods . . . . .	36
3.1.4 Evolutionary Algorithms . . . . .	36
3.1.5 Class EA . . . . .	37

3.1.6	Plug-ins . . . . .	37
3.1.7	Problem-dependent Functions . . . . .	37
3.1.8	The GEA Executable . . . . .	37
3.2	Sample Application: the Traveling Salesman . . . . .	39
<b>4</b>	<b>GraphGEA</b>	<b>41</b>
4.1	Managing the Parameters . . . . .	41
4.2	Running the Evolution Process . . . . .	42
4.3	The Visualization Plug-ins . . . . .	44
4.3.1	Methods of Visualization . . . . .	45
4.3.2	The Implemented Plug-ins . . . . .	50
<b>5</b>	<b>Related Work</b>	<b>53</b>
<b>6</b>	<b>Summary</b>	<b>55</b>
	<b>Acknowledgments</b>	<b>57</b>
	<b>Appendices</b>	<b>59</b>
A	A Detail of GEA's Parameter Structure . . . . .	59
B	The TSP Problem . . . . .	62
	<b>Index</b>	<b>67</b>
	<b>References</b>	<b>69</b>

When the earth was still flat, and the clouds made of fire,  
And mountains stretched up to the sky, sometimes higher,  
Folks roamed the earth like big rolling kegs.  
They had two sets of arms. They had two sets of legs.  
They had two faces peering out of one giant head  
So they could watch all around them as they talked; while they read.  
And they never knew nothing of love. It was before the origin of love.

*Stephen Trask - The Origin of Love*

## Introduction

*Evolutionary algorithms* (EAs for short) are general purpose function optimization methods that search for optima by making potential solutions (*individuals*) compete for survival in a *population*. The better a potential solution is, the better chance it has to survive. The individuals are represented by means of a predefined data structure (*genotype*), and the evaluation considers the performance of the individual in its current environment (*phenotype*). The search space is explored by modifying the genotypes by *genetic operators* observed in nature: generally *mutation* and *recombination* [15, 23, 33].

Evolutionary algorithms have (among others) the following two advantages over other optimization methods: first, in most cases they converge to global optima, and second, the usage of the black-box principle (which only requires knowledge of a function's input and output to perform optimization on it) makes them easily applicable to functions whose behavior is too complex to handle with other methods.

The huge amount of practical applications presented on numerous conferences show that EAs represent a relatively new and important group of function optimization methods. Nevertheless, being stochastic processes, it is hard to understand the functioning of a particular algorithm and build a suitable model of it. An even more difficult problem is to choose the optimal algorithm and determine the values of its parameters for a given problem or problem class.

Visualizing the interiors of an algorithm can be a great help for the understanding of its inner processes and behavior. For example, it is very easy to see the effects of a parameter or a selection method on the diversity of the population in the different phases of the evolution process.

The visualization of evolutionary algorithms is useful in education, too. Not just because it is much easier to fascinate students with a nice and handy graphical user interface, but also because they can become acquainted with the most important features of evolutionary computation. They can experience with different parameter settings and see that the changes in the behavior of the process are really those which they have heard of or read in the literature.

The purpose of this thesis is to present a tool for visualizing evolutionary algorithms: the *GraphGEA* program. *GraphGEA* is a graphical user interface to the *Generic Evolutionary Algorithms Programming Library* (*GEA*) [37]. *GEA* is an easily applicable and extendible evolutionary programming

tool written in the C++ programming language. By interacting with the evolution process running in the background as its child process, the GUI shows the course and the status of the optimization in various configurable visualization windows. *GraphGEA* can be easily extended with new methods showing the interiors of the optimization, for these methods are realized as plug-ins of the system. The communication is implemented by means of the so-called pipe mechanism and UNIX IPC (inter-process communication).

Last but not least, an evolution process can have a great many parameters, the values of which are usually strongly interconnected or dependent. The *GraphGEA* program can just be used to manage optimization projects, for it assures that all necessary parameters of the selected algorithm and representation of individuals are correctly set.

In the following, Section 1 offers an overview of evolutionary algorithms. The presented systems use a special data structure to hold the parameters of the evolution process, a definition of this data structure can be found in Section 2. Section 3 contains the detailed exposition of the *GEA* system: the class hierarchy, the functioning of the various evolutionary algorithms, selection methods and genetic operators are described. There are also examples on the expansion of the system with new evolutionary algorithms or representation forms of individuals. Section 4 presents the *GraphGEA* program with a detailed description of the user interface and the visualization tools. In Section 5, some references to and comparisons with the related work can be found. Finally in Section 6 a summary of the work is given.



# 1 Evolutionary Algorithms

In this section an overview of evolutionary algorithms is given, focusing on details that are important for the *GEA* and *GraphGEA* systems.

*Evolutionary algorithms* (*EAs* for short) are general purpose function optimization methods which use the ‘survival-of-the-fittest’-model known from nature [8]. In this model *individuals* compete for resources in an environment, and *selection* assures that individuals which are better suited for the given environment will produce more offspring. Thus the preservation of good attributes is guaranteed.

Unlike most optimization methods, *EAs* consider several potential solutions at a time. These potential solutions, called *individuals* from now, form a *population*. The individuals interact with each other, thus they create new individuals to form a *new generation*.

An individual of the population is represented with a sort of data structure. The most common representation forms for individuals are *bit-string* and *real vector*. Each element of the vector is called a *gene*. The chain of genes is also called a *chromosome*. The values in it are the individual’s *genotype*. The appearance of an individual – which can be e.g. a permutation of certain numbers – is called *phenotype*. Evolutionary algorithms work on the level of the genotype, which means that they modify the encoded form of individuals. When *evaluating* an individual in its current environment, its phenotype is considered. The result of the *evaluation* is usually a real number, and the task of the evolutionary algorithm is either to maximize or to minimize this number. From the evaluation, a *fitness value* is computed, which is always greater for fitter individuals; this is required for some selection methods to work properly. This fitness value is considered when performing *selection*.

The creation of new individuals is done by applying certain *genetic operators* on the selected parents. The most common genetic operators are *reproduction*, *mutation* and *recombination*. Reproduction and mutation are unary operators. Reproduction simply copies the individual into the new generation, while mutation modifies its argument by randomly changing each gene of it with a certain probability. Recombination takes two or more individuals and creates new ones by exchanging parts of their gene-chains. Each genetic operator is applied with a certain probability. However, sometimes one operator is more efficient than the others and it is not easy (or at least it requires experiment) to set the probabilities correctly at the start of an evolution process. Davis offers a solution to this problem: let’s change the probabilities dynamically during the evolution process by observing the effectiveness of the operators. He calls this method the *adaptation of operator probability* [9].

A general procedure of an evolutionary algorithm can be seen on Algorithm 1. As all algorithms in this paper, it is formulated in a Pascal-like pseudo-language. Generally, the procedure of an evolutionary algorithm is the following: the structures in the initial population can be generated randomly or, if an initiative solution is known, then that can be used with random modifications.

In the simplest cases, the algorithm is terminated if a certain generation number is reached or the fitness value of the best individual of the population hasn't changed for a certain number of generations. In the body of the while-loop, the generation of the new population is absolutely algorithm-dependent, so these methods will be discussed at the specification of the algorithms.

---

**Algorithm 1** A general evolutionary algorithm

---

```

procedure ea()
begin
  t = 0           // current generation number
  initialize P(t) // starting (initial) population
  evaluate structures in P(t)
  while (termination condition not satisfied) do
    t = t + 1
    create structures from P(t-1) into P(t) using selection and genetic operators
    evaluate structures in P(t)
  endwhile
end

```

---

Several kinds of evolutionary algorithms are known, from which the most important ones are *genetic algorithms (GAs)* [10, 15] and *evolution strategies (ESs)* [32]. They were developed independently in the 1970s: GAs were introduced by John Holland and analyzed by his students (e.g. Kenneth De Jong) in the USA, and at the same time, evolution strategies were invented in Germany by Ingo Rechenberg. The main differences between these two kinds of EAs are the method of creating the new generation and the typical representation form for individuals. The typical representation form for individuals is bit-string for GAs and real vector for ESs. The two kinds of EAs also differ in the mode genetic operators are applied.

There is a special kind of genetic algorithms, namely *genetic programming (GP)*, introduced by John R. Koza [23]. The main invention of GPs is that branching structures can be evolved. Most of the methods are the same as in GAs, but there are special genetic operators designed for branching structures: e.g. recombination replaces subtrees of the selected individuals.

Evolution strategies can be divided into two classes as well: the so-called *plus and comma strategies*. In short, the difference between the two strategies is that when the comma strategy is used, parents die off after creating their offspring. In the case of the plus strategy, parents compete with their offspring for survival.

In the following, the characteristics of genetic algorithms, genetic programming and evolution strategies are described in detail. In Subsection 1.4, the theoretical founding of probability adaptation is described.

## 1.1 Genetic Algorithms

Genetic algorithms are the most popular sort of evolutionary algorithms, where the individuals are usually represented by a series of bits. The genetic operators are implemented in accordance with this representation form. Genetic algorithms have proven to be successful at searching mul-

No. of gene	1	2	3	4	5	6	7	8	9	10
Parent 1	1	0	0	1	0	1	1	0	0	1
Parent 2	1	0	1	1	0	0	1	1	1	0
Crossover points			↑ cp1	↑ cp2						
Descendant 1	1	0	1	1	0	1	1	0	0	1
Descendant 2	1	0	0	1	0	0	1	1	1	0

Figure 1: The functioning of the GA crossover

tidimensional spaces in order to solve, or solve approximately, a wide variety of problems [13, 26]. Here follows the description of the two most important genetic operators for GAs: *mutation* and *crossover*.

### 1.1.1 GA Mutation

Mutation randomly changes each bit of an individual with a certain probability ( $p_{mut}$ ). The change can be done by either flipping a bit or replacing its value with a newly generated random value. In both cases it is important that it is considered for each bit whether to change that bit or not.

### 1.1.2 GA Crossover

In the case of GAs, the recombination operator always takes two parents and creates two descendants, thus it is often called crossover as well. The main kinds of crossover are *single-point*, *multi-point* or *parametrized uniform crossover*.

For single-point and multi-point crossover, the crossover points (whose number is given) are chosen at random. For single-point crossover, this number is always one, and for multi-point crossover, it is given in a parameter  $N_{cp}$ . In point of fact, single-point crossover is a special case of multi-point crossover (i.e. when  $N_{cp}$  is set to 1). After choosing the crossover points, the parts of the individuals between these points are being exchanged. For example, if the GA works with individuals of length 10 and  $N_{cp}$  is set to 2, then the two crossover points (let's call them  $cp_1$  and  $cp_2$ ) are chosen from the range 0 – 10, inclusive. Suppose that  $cp_1 = 2$  and  $cp_2 = 4$ . Then the 3rd and 4th bits are exchanged in the two individuals. This can be seen in Figure 1.

Parametrized uniform crossover exchanges each bit of the parent individuals with probability  $p_{ue}$  to create the descendants. It works like the mutation operator, in the meaning that it considers each gene whether to exchange that gene or not.

### 1.1.3 Creation of the New Generation

The process of creating the new generation for a GA is quite simple, it is sketched on Algorithm 2. First a new empty population is created. Then a number of best individuals in the previous

generation is copied into the new population as determined by the elitism rate parameter. After that the remaining places are filled out in the population by selecting two parent individuals from the old population, performing mutation and crossover on them, and inserting either one or both of the descendant individuals into the new population as necessary. These operations (from the selection to the insertion) are repeated in a loop until the new population has enough individuals. In order to guarantee the monotonicity of the process, some of the best individuals can be copied (reproduced) into the new generation. The genetic parameter *elitism rate* ( $r_e$ ) holds the number of reproduced individuals. The other individuals are generated by selecting two parent individuals from the old generation and performing mutation and crossover on them. The selection method is a very important part of genetic algorithms, since selection assures that the fitness values of the individuals are constantly increasing during the evolution process. The various selection methods used by GAs are described in Subsection 1.1.4.

---

**Algorithm 2** The algorithm for creating the new generation for a GA

---

```

procedure ganextgen(oldpop)
begin
  create newpop as an empty population
  if ( $r_e > 0$ ) then
    copy the best  $r_e$  individuals from oldpop into newpop
  endif
  while (sizeof(newpop) < sizeof(oldpop)) do
    select  $p_1$  and  $p_2$  from oldpop to be the parents
    mutate  $p_1$  and  $p_2$ 
    perform crossover on  $p_1$  and  $p_2$  to get the descendants  $d_1$  and  $d_2$ 
    put  $d_1$  into newpop
    if (sizeof(newpop) < sizeof(oldpop))
      put  $d_2$  into newpop
    else
      dispose  $d_2$ 
    endif
  endwhile
  return newpop
end

```

---

#### 1.1.4 Selection Methods

Since there are a wide range of functions that can be optimized with genetic algorithms and these functions behave very differently, a number of selection methods have been developed to deal with the various functions [28]. For example if a function has many local optima and some of these optima are very close to the global optimum, then selection pressure should be kept low in order to explore the whole search space rather than founding one local optimum and get stuck at it. An example function with more local optima can be seen in Figure 2. It is  $f(x, y) = \frac{1}{|x|+|y|+0.5} + \frac{1}{|x-6|+|y-3|+0.8} + \frac{1}{|x+8|+|y-4|+0.9}$ , which has its global optimum in  $(0, 0)$  and local optima in  $(6, 3)$  and  $(-8, 4)$ . For easier functions, which are smooth and have no local optima, the selection pressure can be set high in order to achieve faster convergence. Selection pressure here

A sample function with more local optima

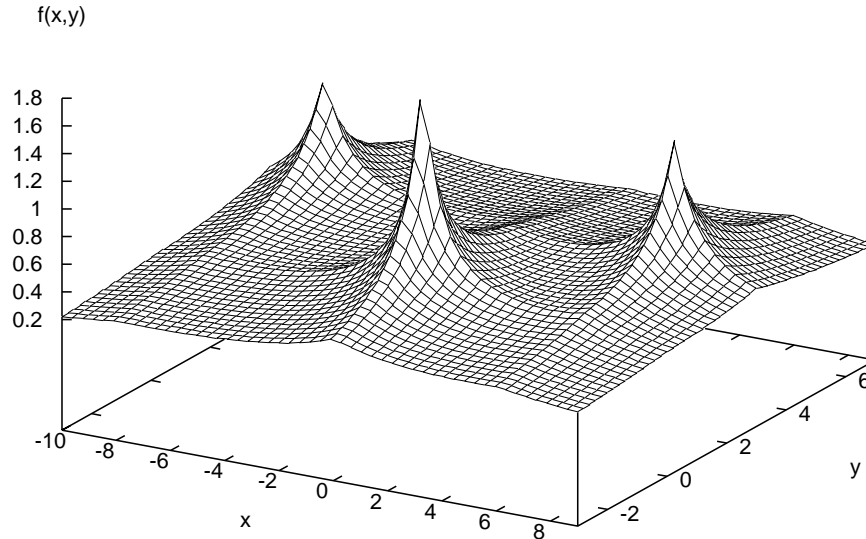


Figure 2: A sample function with more local optima

means a function of fitness value that determines the relationship between fitness values and the probability of an individual with that fitness value to get selected. The most important and widely used selection methods are the following:

**Fitness proportional selection with roulette wheel sampling.** For this method, each individual is assigned a slice of a roulette wheel; the size of the slice is proportional to the individual's fitness value. The wheel is spun, and the individual under the wheel's marker is selected. The problem with this method is that if the fitness variance in the population is low, then all individuals have nearly equal chance to be selected as a parent. Thus, better individuals cannot create more descendants than lower fit ones. To address this problem, each individual's fitness value is decreased by a certain portion (which is determined by the parameter  $0 \leq s_{red} \leq 1$ ) of the fitness value of the worst individual in the population. The effect of this parameter can be seen in Figure 3. To create graphs for Figures 3 and 4, a population of 51 individuals was taken. The individuals' fitness values were set to 50..100. With this population, 100.000 selections were made with each selection method. The graphs show the number of times an individual was selected with a given fitness value.

**Rank based selection.** Rank based selection, like fitness proportional selection, uses a roulette wheel to select individuals. Here the slice of the roulette wheel assigned to an individual is proportional to its rank in the sorted population. Thus, the worst individual gets 1 unit and

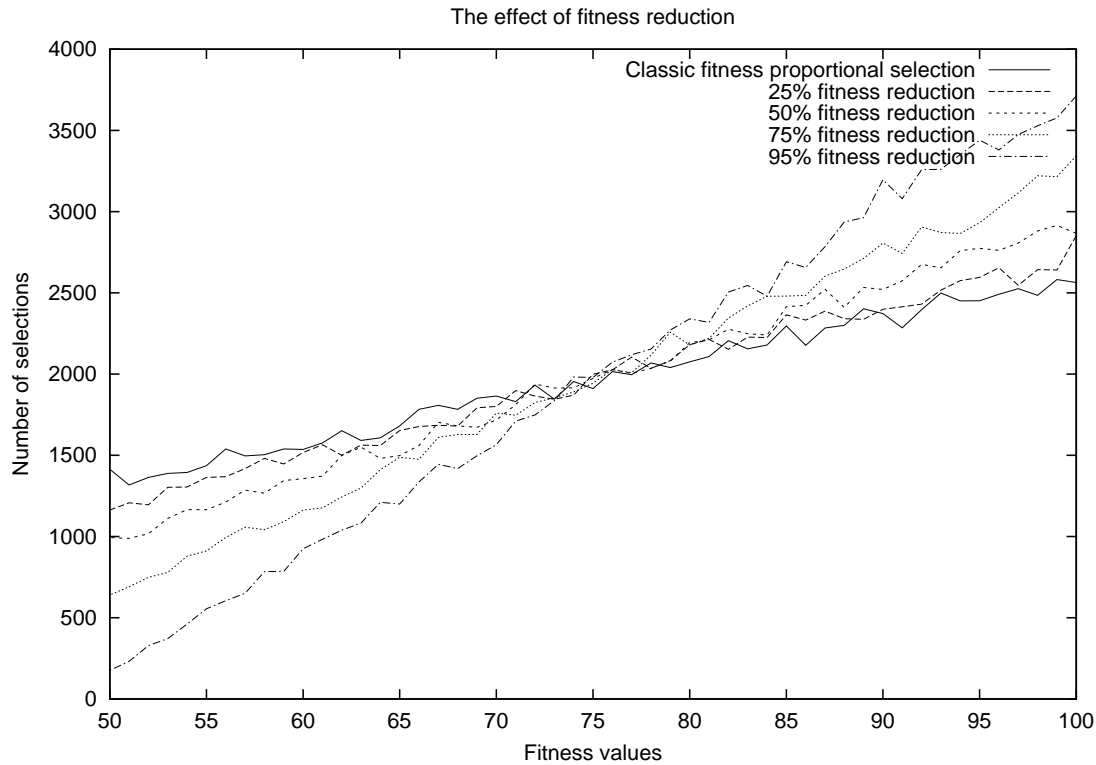


Figure 3: Effect of fitness reduction on the selection pressure at fitness proportional selection

the best one gets  $n$  units, where  $n$  is the population size.

**Tournament selection.** Tournament selection is similar to rank selection in terms of selection pressure, but it is computationally more efficient. Two individuals are chosen at random from the population. A random number  $r$  is generated between 0 and 1. If  $r < k$  (where  $k$  is a parameter, e.g. 0.75), then the fitter individual is selected. Otherwise, the less fit individual is selected.

**Best selection.** This method simply selects the best individual from the population.

**Random selection.** This method selects an individual at random.

**Sigma scaling.** Sigma scaling (like fitness reduction) is another method to address the problem mentioned at fitness proportional selection. When applying this method, the slice of the roulette wheel assigned to an individual is a function of the individual's fitness value, the population mean and the population standard deviation. The modified fitness value is computed with the following formula:

$$f^*(i) = \begin{cases} 1 + \frac{f(i) - \bar{f}}{2s} & \text{if } s \neq 0, \\ 1 & \text{if } s = 0 \end{cases},$$

where  $f^*(i)$  is the modified fitness value of individual  $i$ ,  $f(i)$  is the original fitness value of individual  $i$ ,  $\bar{f}$  is the mean fitness of the population, and  $s$  is the standard deviation of the population fitnesses. As it can be seen in Figure 4, Sigma scaling performs much like fitness

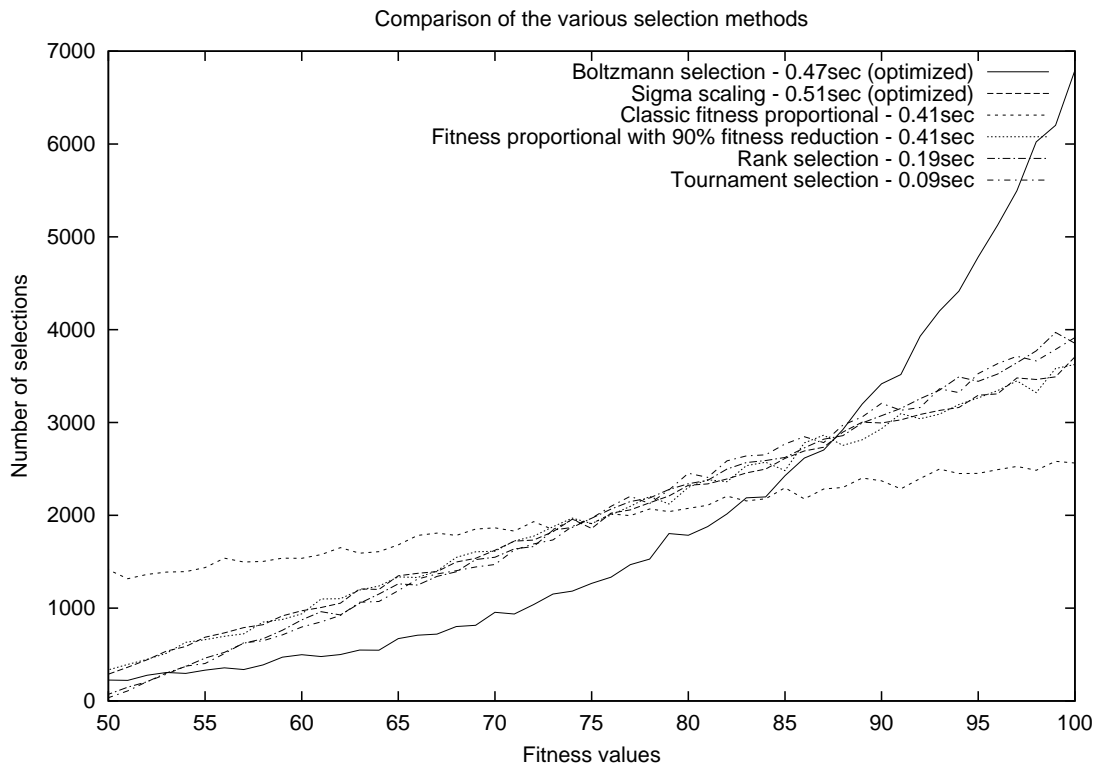


Figure 4: Comparison of the various selection methods regarding the selection pressure and required CPU time

proportional selection with fitness reduction of 90%, but it consumes a little more CPU time.

**Boltzmann selection.** Sigma scaling keeps the selection pressure more constant over a run. But sometimes it may be good to have a stronger selection later in the evolution process in order to strongly emphasize highly fit individuals. One approach for this idea is Boltzmann selection. In this method, the slice of the roulette wheel assigned to an individual  $i$  is computed by the formula

$$f^*(i) = \frac{\exp(f(i)/T)}{\langle \exp(f(i)/T) \rangle},$$

where  $f^*(i)$  and  $f(i)$  are as at Sigma scaling,  $\langle \dots \rangle$  denotes the average of the actual population, and  $T$  is a temperature which is constantly decreasing over a run. With lower values of  $T$ , the difference in  $f^*(i)$  between high and low fitnesses increases.

**Interactive selection.** This method has been introduced for the sake of interactive evolution. For example, graphical objects can be evolved with interactive evolution, where an appropriate fitness function cannot be formalized. In this case, the user of a program can choose between the individuals. Besides other areas, interactive selection is widely used in design and shape recognition [2, 5, 14, 25, 35, 41].

In Figure 4 a comparison of the selection methods can be seen. Each line shows the performance of a particular selection method. The CPU times show the required CPU time for applying the

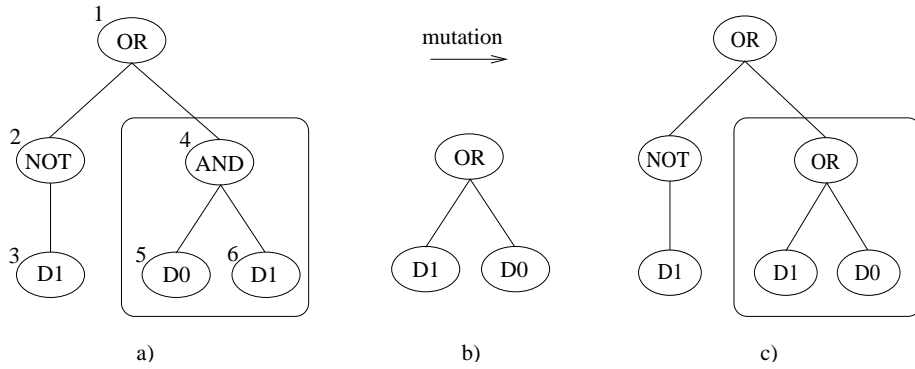


Figure 5: Sample trees for GP mutation

appropriate method 100.000 times. Used parameters:  $k = 0.75$  for tournament selection, and  $T = 15$  for Boltzmann selection. The CPU times seen in Figure 4 were measured on an IBM PC compatible computer equipped with a 600MHz Intel Pentium III processor.

## 1.2 Genetic Programming

It is difficult and restrictive to represent hierarchies of dynamically varying size and shape with fixed length vectors. Genetic programming uses the same algorithms for creating the new generation and selecting individuals as genetic algorithms. The difference between GAs and GP is that GP uses a tree-like representation form for individuals, thus it provides a way to find a function or a computer program of unspecified size and shape to solve a problem [23].

For example, genetic programming has been successfully applied to problems such as classification [1] and pattern recognition [24, 34], generation of maximal entropy sequences of random numbers [22], Boolean function learning [11, 27], simultaneous architectural design [29] and training of neural networks [30].

GP's genetic operators work with sub-trees of the individuals. The logical expressions used as examples for the next two subsections were taken from [23].

### 1.2.1 GP Mutation

GP mutation is presented here through an example. Suppose that the selected individual from the population of logical expressions is  $\text{OR}(\text{NOT}(\text{D1}), \text{AND}(\text{D0}, \text{D1}))$ , its tree can be seen in Figure 5a. The randomly selected crossover point is numbered with 4, thus the subtree that will be changed is  $\text{AND}(\text{D0}, \text{D1})$ , framed in Figure 5a. The newly generated subtree can be seen in Figure 5b, it corresponds to the logical expression  $\text{OR}(\text{D1}, \text{D0})$ . After replacing the selected part, the resulting logical expression is  $\text{OR}(\text{NOT}(\text{D1}), \text{OR}(\text{D1}, \text{D0}))$ , its tree can be seen in Figure 5c.

GP mutation has two extra parameters that control the range of the mutation. The first one determines the maximum depth of the exchanged subtree ( $\text{depth}_{del\_max}$ ), and the other tells the operator what is the maximum depth of the randomly generated subtree to put into the place of the exchanged one ( $\text{depth}_{rand\_max}$ ).



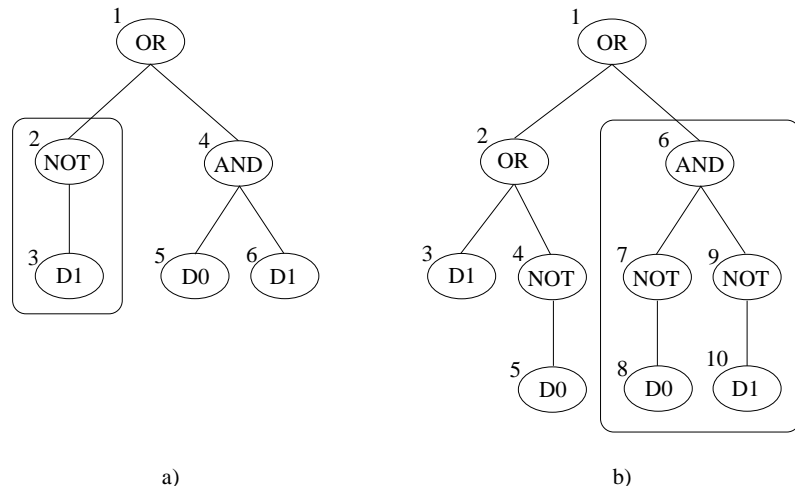


Figure 6: Parent trees for GP crossover

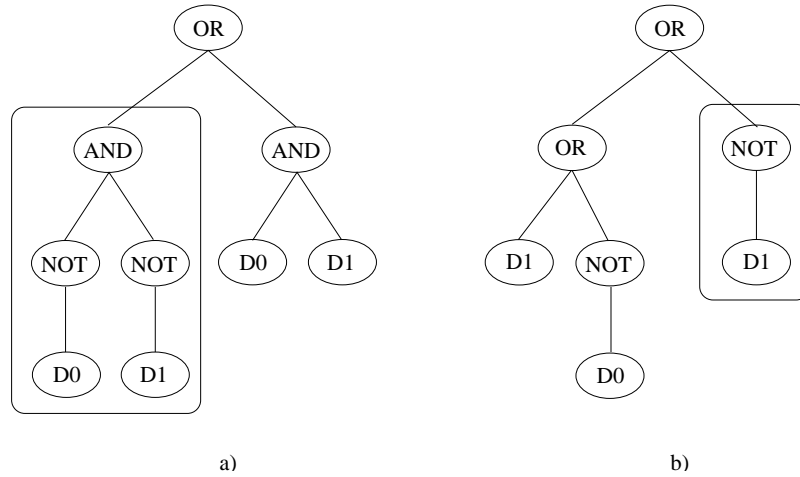


Figure 7: Offspring trees for GP crossover

### 1.2.2 GP Crossover

Another example helps presenting how the GP crossover (or GP recombination, as it is called in GAs) works. GP crossover exchanges randomly selected subtrees of the selected parent trees. Suppose that the logical expressions selected from the actual generation are  $\text{OR}(\text{NOT}(\text{D1}), \text{AND}(\text{D0}, \text{D1}))$  and  $\text{OR}(\text{OR}(\text{D1}, \text{NOT}(\text{D0})), \text{AND}(\text{NOT}(\text{D0}), \text{NOT}(\text{D1})))$ , their tree representations can be seen in Figure 6a and 6b, respectively. The randomly chosen recombination points are 2 in the first individual and 6 in the second one. The subtrees to be exchanged are framed.

The offspring individuals achieved by replacing the selected subtrees in the parents can be seen in Figure 7, and the new logical expressions are  $\text{OR}(\text{AND}(\text{NOT}(\text{D0}), \text{NOT}(\text{D1})), \text{AND}(\text{D0}, \text{D1}))$  and  $\text{OR}(\text{OR}(\text{D1}, \text{NOT}(\text{D0})), \text{NOT}(\text{D1}))$ .

GP crossover also has its own parameter for determining the maximal depth exchangeable subtrees ( $\text{depth}_{\text{chg-max}}$ ).

### 1.3 Evolution Strategies

Evolution strategies are less popular than genetic algorithms, but they are very interesting because they stand closer to the natural evolution since competition with their descendants is enabled for the parent individuals.

There are two kinds of evolution strategies, the so-called comma and plus strategies:  $(\mu/\rho, \lambda)$ -ES and  $(\mu/\rho + \lambda)$ -ES. Here  $\mu$ ,  $\rho$  and  $\lambda$  denote the population size, the number of parents used in recombination and the size of the selection pool, respectively. The selection pool is a temporary storage for individuals: offspring of the selected parents are put into it and the new generation is formed from the best  $\mu$  individuals of the selection pool. The difference between the comma and plus strategies is that the plus strategy puts the old population (the parents) into the selection pool after generating  $\lambda$  individuals. There are special cases for ES, e.g. when  $\rho$  is set to 0 or 1 (or omitted) then recombination doesn't take place, only mutation is applied. Other special cases are  $(1 + 1)$ -ES (hill climbing) and  $(1, 1)$ -ES (random search).

For ESs, the common representation form of individuals is a fixed length real vector. The genetic operators are developed in accordance with this specific representation form.

#### 1.3.1 ES Mutation

The mutation operator of evolution strategies is very similar to that of genetic algorithms: it changes each element of the real vector (i.e. each gene) with a certain probability ( $p_{mut}$ ). The difference originates from that the genes are real numbers, so they can be either multiplied or increased by a random value (the distribution of the value added to the gene is usually normal). The extent of this random value is controlled by the parameter mutation rate ( $rate_{mut}$ ). Whether multiplication or addition is applied in mutation is controlled by a parameter of the process ( $muttype_{ES}$ ).

#### 1.3.2 ES Recombination

ES recombination takes  $\rho$  individuals as parameters and produces one descendant of them. (Recall that GA's crossover takes two parent individuals and creates two descendants.) ES recombination methods can be classified by two aspects: There are *discrete/intermediate* and *local/global* recombination methods.

**Discrete recombination.** When computing the value of a gene of the descendant, discrete recombination randomly picks one parent individual from the selected ones (cardinality of which is  $\rho$ ) and takes the picked individual's gene value as the descendant's gene value.

**Intermediate recombination.** Intermediate recombination computes the gene values of the descendant by taking the mean values of the parents' genes.

**Global recombination.** Global recombination takes all the  $\rho$  selected parent individuals into account when computing the values of the descendant's genes.

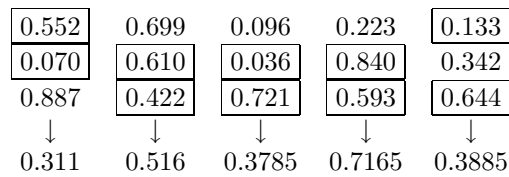


Figure 8: The functioning of the local intermediate ES recombination

**Local recombination.** Local recombination randomly picks  $\rho'$  ( $\rho' \leq \rho$ ) parent individuals each time a gene's value is computed, and considers only these parents in the computation.

To demonstrate the functioning of the ES recombination, suppose that local intermediate recombination is applied,  $\rho = 3$  and  $\rho' = 2$ . The length of the chromosome (the real vector) is 5. Figure 8 shows how the operator is applied on the selected individuals. For each of the five genes, two parent individuals are picked randomly; these selected genes are framed in the figure. Then the average of these two values are computed and that value will be the new individual's gene value. A detailed description of the ES recombination operator with examples can be found in Section 5.3 of [16].

### 1.3.3 Creation of the New Generation

The algorithm for creating the new generation!ES new generation for an ES can be seen on Algorithm 3.

First  $\lambda$  individuals are created in the empty selection pool. To create a new individual,  $\rho$  parent individuals are selected *randomly* from the old population. Then recombination is performed on these individuals to get a descendant. After mutating the descendant, it is put into the selection pool. In the case of the plus strategy, the individuals from the old population are also put into the selection pool. Note that the random selection does not assure the convergence of the process, it is assured by forming the new generation from the best  $\mu$  individuals of the selection pool.

---

**Algorithm 3** The algorithm for creating the new generation for an ES

---

```

procedure esnextgen(oldpop)
begin
  create the empty selection pool
  while (sizeof(selection pool) <  $\lambda$ ) do
    select  $p_1, p_2, \dots, p_\rho$  randomly from oldpop to be the parents
    perform recombination on  $p_1, p_2, \dots, p_\rho$  to get the descendant  $d$ 
    mutate  $d$ 
    put  $d$  into newpop
  endwhile
  if plus strategy is applied then
    copy oldpop into the selection pool
  endif
  select the best  $\mu$  individuals from the selection pool to form newpop
  return newpop
end

```

---

### 1.3.4 Meta-ES

In nature, it can be observed that populations of the same species are sometimes evolving separately, and after some generations, they meet. In the field of evolution strategies, this phenomenon is realized by means of the so-called meta-ES method ([16], Subsection 5.4.5). In meta-ES, several populations of the same type are evolved separately for some generations, and these populations are modified by genetic operators. I.e. the populations are regarded as individuals (vectors of individuals), thus genetic operators can be applied on them. Mutation can be done by randomly replacing some individuals in the population, and recombination can work as crossover works in GAs.

## 1.4 The Theoretical Basis of Probability Adaptation

The individuals during an evolutionary process can be modified by genetic operators such as recombination or mutation. The convergence speed of the process can be increased by adapting the probabilities of the operators according to their effectiveness: when modifying one or more individuals with an operator, the fitness values of the original and the modified individuals are compared. The operator's probability ( $p_\omega$ ) is recomputed in accordance with the result of the comparison. The more effective the operator is, the more likely it is used in the future. Thus, the probabilities of the operators are continually changed during the evolution process [9].

The probability of operator  $\omega \in \Omega$  when creating generation  $t$  is

$$p_\omega(t) = p_\omega(t-1) \cdot \frac{\sum_{i \in I_{\omega,t-1}} f_{\omega,a}(i)}{\sum_{i \in I_{\omega,t-1}} f_{\omega,b}(i)},$$

where

- $\Omega$  is the set of operators used in the evolution process
- $p_\omega(t)$  is the probability of operator  $\omega \in \Omega$  when creating the individuals of generation  $t$
- $I_{\omega,t}$  is the set of individuals from generation  $t$  that were modified by the operator  $\omega$  when generation  $t$  was created
- $f_{\omega,b}(i)$  is the fitness value of the individual  $i$  before operator  $\omega$  was applied on it
- $f_{\omega,a}(i)$  is the fitness value of the individual  $i$  after operator  $\omega$  was applied on it
- for each  $\omega \in \Omega$ ,  $p_\omega(1) = p_\omega$ , i.e. the initial probability of each operator is set from the parameters of the evolution process

It is possible that the probability of an operator becomes greater than 1 (if the operator is very successful in finding better solutions). Since it is mathematically incorrect, probabilities greater than 1 have to be set to 1. It is also possible to protect operators from getting too low probabilities, thus excluded from the evolution process: for this,  $p_{\omega,min}$  values have to be defined for each operator (thus, if  $p_\omega(t) < p_{\omega,min}$  holds for some  $\omega \in \Omega$  and  $t \geq 1$  then  $p_\omega(t)$  can be set to  $p_{\omega,min}$ ). The definition of such minimum values requires much experience from the implementer of the process. Detailed descriptions of similar probability adapting methods also can be found in [9, 16].

## 1.5 Possibilities of Visualization

Visualizing an evolutionary algorithm is useful for *controlling* its run and *understanding* its behavior. Controlling includes the configuration and the interactive execution of the evolution process. The behavior can be analyzed by observing the operation of the selection and the genetic operators, the quality of the solutions found, the individuals' genotypes and phenotypes etc.

To show the internals of the process, basically the following *three techniques* can be applied:

**Plots** are suitable for displaying a smaller amount of numerical data like the values of a feature as a function of one or two other parameters. Depending on the number of the function parameters, two-dimensional or three-dimensional plots can be created.

**Color coding** is an efficient method to display larger amounts of numerical data in a tabular and still easily readable form. Here a two-dimensional table is created, the rows and columns being indexed by the discrete values of the two parameters and the cells representing the respective value by a color. A color is assigned to both the lowest and highest values in the table and intermediary values are represented by tones between these two colors.

**Drawings** can be used to display graphical objects such as the phenotypes of the evolved individuals. This way the changes and differences on the genotype level can be easily recognized as corresponding changes in the individuals' behavior in their evaluating environment.

When talking about visualization possibilities, one has to distinguish between the so-called *course* and *status visualization* methods, that is, between the ones that provide information about the *progress* and the *current status* of the process.

In the case of evolutionary algorithms, course visualization includes plots of particular fitness values, consumed system resources and the diversity of the population (e.g. standard deviation of the fitness values). The plots are usually drawn against generation number or, in the case of a steady-state GA, the number of evaluated individuals but the used CPU time is a very good base for benchmarks, too.

The most useful color-coding progress visualization methods are those which show the best individuals' genotypes and the fitness values of all individuals of each generation. Though the first method is applicable only for fixed-length numerical chromosomes, together with the fitness graphs, it helps identifying the roles and importance of the single genes or gene groups. The latter view of the population shows somewhat more information about the fitness distribution than deviation graphs.

In most cases, displaying information about the current status of an evolution process means showing some characteristics of the complete current generation. This information can be for example the genotypes or phenotypes of all individuals or just the occurring lowest and highest gene values.

Showing the phenotypes of individuals can be very productive when one needs to understand the

connection between the genotypes and the phenotypes. However, being a completely problem-dependent visualization technique, it requires more implementation work from the user than just providing a fitness function.

A very important aspect of graphical data portrayal is the correct determination of the amount of the displayed information: the views should be enough for the user to be able to find the sought relations. On the other hand, they say that one figure is worth a thousand words; but the user should not be overwhelmed by an undigestable pack of knowledge.

## 2 The e\_params Data Structure

This section gives a detailed description of a data structure that was designed to hold parameters of arbitrary objects such as various processes, data elements etc. Its main features are that *relationships* can be defined between the parameters and *conditions* and *restrictions* for the parameter values.

The data structure is designed in a way that the input and output of the functions are stored in easily readable text files, thus they can be modified with a plain text editor or script files.

**e\_params** is implemented in *ANSI C* language for portability and simplicity reasons. It uses some elements of the *GLib*<sup>1</sup> library which is distributed under the *Free Software LGPL* and is available on UNIX, Win32 and OS/2 platforms. The current version of **e\_params** is 0.14.

An extension has been implemented which enables the setting of the parameter values on a graphical user interface (GUI). This extension is written in *ANSI C* as well and it uses the *GIMP ToolKit (GTK)*<sup>1</sup>, which is available on several platforms including Linux and Win32 systems. Of course the **e\_params** data structure can be used without the graphical extension.

### 2.1 Requirements on the Data Structure

The first application of the **e\_params** data structure is related to evolutionary algorithms. The ordinary data types, possible conditions, restrictions and relationships are defined in a way that suits this purpose.

The domain of evolutionary algorithms requests that a list of *main parameters* (the values of which have to be given in every case) should be defined, and some ordinary types can have *dependent parameters* which have to be set only if the value of another parameter satisfies a condition. Moreover, *conditions* can be defined for some data types and certain parameters can *restrict* the possible values of other parameters.

With such a parameter description system, *GEA* and *GraphGEA* can easily be expanded with new individual representations, evolutionary algorithms, selection methods, visualization plug-ins, etc. The following list gives the defined *data types* and their possible conditions, restrictions and dependencies in the **e\_params** data structure.

**String** Holds an arbitrary character string.

**FileName** and **DirName** Hold the name of a file or a directory as a character string. If the file/directory does not exist, the value checker function returns an error value only if the special attribute *AllowNonExistent* is set to *no*.

**Integer** Holds an integer value. One or more *conditions* can be assigned to an integer parameter, all of which must be fulfilled. The parameter value can be bounded by prescribed integer values or other integer-valued parameters. Special conditions can also be defined, which must be implemented in the code of the **e\_params** package.

---

<sup>1</sup>See <http://www.gtk.org> for details

**Real** Holds a single-precision real value. The available *conditions* are similar to those of the integer data type, i.e. one or more conditions can be defined and all of them must be fulfilled by the parameter value. Value boundaries can be predefined real numbers or other real-valued parameters. Special conditions also can be defined.

**Percent** Holds a percentage value as a single-precision real value, that is, a real value between 0.0 and 100.0, inclusive.

**OneOrMore** Holds a positive integer value. If the value of the parameter is greater than 1, then other parameters also can be defined. These parameters are called *dependent parameters*.

**YesNo** Holds a Boolean value. If the value is *true* then dependent parameters can/must be set, too.

**OptionList** Holds one of some predefined values (Options). The predefined values are listed and the user can select one from them. Each of the predefined values can have *dependent parameters* (which must be set only if the parameter is set to this option) and the options can also *restrict* the possible values of other OptionList parameters.

**special types** Special data types can be defined as well. For this, an identifier must be chosen for the new data type and the source code of the `e_params` data structure has to be modified in order to support the new type. At the present state of the package, there is one such special type, called **GeneBounds** which holds the boundaries of each gene of a real vector individual representation in a two-dimensional real-valued array.

At this point, it can be confusing what are conditions and what are restrictions in the system. *Conditions* can be assigned to numerical parameters by setting lower and/or upper boundaries for them. The value of the numerical parameter is valid iff it meets all the conditions assigned. *Restrictions* are a kind of relationship between an Option and a parameter of type OptionList: when an Option is assigned to a parameter as its value, the possible values of other OptionList parameters can be limited. For example, if the representation of the individuals in an evolution strategy is BitString, it doesn't make sense to compute the average of the bits, so the intermediate recombination cannot be selected as the recombination type. A *function* is provided for the data structure that *checks* whether the value of a given parameter *satisfies* its conditions and restrictions or not.

All parameters of a given object can have default values which are defined along with the parameters.

## 2.2 Implementation

This subsection describes what data types and constants are defined in the `e_params` data structure, what their meanings are and how they are used. The implemented functions are also described.



### 2.2.1 Data Types

Five *struct* data types are defined in the system:

**e\_params** This is the main data type, it has two GHashTable fields for the parameters and the Options.

**ep\_paramdata** This structure represents a parameter in the system. It has the following fields:

**char \*name** The name of the parameter; this string is used in the input and output files and in the programs to identify the parameter. This field is used as the key in the parameters hash table of the **e\_params** structure. This value should not contain whitespace and other special characters.

**char \*display** This string is written on the screen when the user is prompted to assign a value to the parameter. It is allowed to contain any printable characters.

**int type** This field holds the type of the parameter. The defined types are exactly as described in Subsection 2.1.

**int optionnum** and **char \*\*options** If the type of the parameter is OptionList, then the keys of the possible values (Options have a **name** field as well, which is used as the key value, see below) are stored in this string array and the integer value tells the size of the array. These two fields are used by the OneOrMore type, too, to store the names of the parameters that must be set if the parameter value is greater than 1 (i.e., the dependent parameters of the OneOrMore parameter).

**int condnum** and **ep\_condition \*\*conditions** These fields hold the conditions that must be fulfilled by a numerical parameter value. The structure **ep\_condition** is described below.

**void \*value** Points to a memory location where the actual value of the parameter is stored. The actual type of the pointer is determined according to the type of the parameter.

**void \*deflt** Points to the default value of the parameter, if there is one. If there is no default value, then this field is NULL.

**int result** This field is only for internal usage, e.g. by the condition checker function.

**int restrnum** and **char \*\*restricteds** For the sake of effectivity, the list of the parameters which can be restricted by this parameter is stored here as well. This data is computed after the whole data structure is loaded.

**char \*pattern** Holds a file name pattern for **FileName**-typed parameter.

**char \*group** The name of the group of parameters this parameter belongs to. The group is used on the GUI to separate parameters of different purposes onto different graphical elements.

**int allowempty** If set to a positive value, then the parameter's value does not have to be defined (can be undefined/NULL).

**int allownonexistent** If set to a positive value, then the value of a parameter of type `FileName` or `DirName` can point to a file that does not exist.

**char \*basedir** Holds a base directory name for a parameter of type file name.

**ep\_option** This type represents an option that is presented to the user when setting the value of a parameter of type `OptionList`. The defined fields are the following:

**char \*name** The name of the option which is used to identify the option internally and this value is written to the input/output files for communication and storing the values.

**char \*display** This string is presented to the user of a system that uses the `e_params` data structure when setting the value of the parameter.

**int paramnum** and **char \*\*params** These fields store the dependent parameters of the option, that is, if the option is selected, then the values of other, additional parameters have to be specified by the user.

**int restrnum** and **ep\_restriction \*\*restrs** specify whether this option restricts the possible values of other `OptionList`-type parameters.

**char \*special** can hold option-specific data.

**ep\_condition** A condition for a numerical parameter can be represented by this type. It has a field that tells the type of the condition (there are constants defined for this purpose) and three fields of different types to hold the value of the parameter (integer, real value or parameter name).

**ep\_restriction** represents a restriction relation between an `Option` and an `OptionList`-type parameter. For this purpose, it stores the name of the restricted parameter and the `Options` that are allowed for the parameter.

### 2.2.2 Functions

The following functions are implemented to help to work with the data structure:

**e\_params \*e\_params\_new\_from\_file(char \*filename)** Creates a new `e_params` data structure by reading the parameters and options from the given file. Returns a pointer to this newly created object, or `NULL` if an error occurs. After the input file is read and the structure is built up, an integrity check is performed to assure that all referenced options and dependent and restricted parameters are defined (for this purpose, the system uses the functions `e_params_check_integrity_of_parameter` and `e_params_check_integrity_of_option`).

**e\_params \*e\_params\_new\_from\_FILE(FILE \*pf, char \*filename)** This function does the same as **e\_params \*e\_params\_new\_from\_file** except for the fact that it reads the data from an already open file. The second argument is to provide information when errors occur.

**void e\_params\_delete(e\_params \*ep)** Frees the memory used by the given **e\_params** object.

**void e\_params\_add\_parameter( ... )** Creates a new parameter with the given name, display, type and default values and inserts it into the parameters hash table of the **e\_params** object passed in the first argument.

**void e\_params\_add\_option( ... )** Creates a new option with the given name, display, dependent parameters and restrictions and inserts it into the option hash table of the **e\_params** object passed in the first argument.

**ep\_paramdata \*e\_params\_get\_parameter(e\_params \*ep, char \*name)** Returns a pointer to the parameter with the given name in the **e\_params** object, or NULL if no parameter with the given name is found.

**ep\_option \*e\_params\_get\_option(e\_params \*ep, char \*name)** Returns a pointer to the option with the given name in the **e\_params** object, or NULL if no option with the given name is found.

**e\_params\_delete\_param** and **e\_params\_delete\_option** These functions are used to delete a parameter or a option from the structure, respectively.

**int e\_params\_check\_value\_of\_parameter(e\_params \*ep, ep\_paramdata \*eppd)** Checks whether the parameter's actual value in the given parameters structure satisfies all the conditions and restrictions that are specified for it. The returned error codes are defined in an include file. The restriction check is performed by the function **e\_params\_check\_restrictions**.

**void e\_params\_set\_defaults(e\_params \*ep)** Sets all parameter values in the given parameters structure to their defaults (in the case when the default value of a parameter is not defined, the value is set to NULL).

**float ep\_paramdata\_get\_numeric\_value(ep\_paramdata \*eppd)** Returns the numerical value of the given parameter, if it is a numerical parameter. This is quite a simple function, but still very useful, because the type of the parameter has to be tested in order to determine the type of the **value** pointer.

**int e\_params\_read\_from\_file( ... )** Reads a list of parameters from a file into a given parameters structure. The dependent parameters are read as well, by calling the parameter reader function recursively.

**int e\_params\_write\_to\_file( ... )** Saves a list of parameters to a given file. The dependent parameters are saved as well, by calling the parameter saver function recursively.

**ep\_parameter\_list \*e\_params\_read\_parameter\_list\_from\_file( ... )** Reads a parameter list file (which has the suffix “.ep1”) into a *parameter list* data structure. The function **e\_params\_free\_parameter\_list** can be used to free the memory allocated by the parameter list.

## 2.3 Input and Output Files

The definition of a parameters data structure (as it can be seen from the constructor of **e\_params**) can be stored in a file with the suffix “.ep”. Actually, this file should be a plain text file; the format of this file is given formally in Subsection 2.3.1 by giving a context-free grammar and context-sensitive restrictions on it.

The files in which the *parameter values* can be stored and which list a *set of parameters* (for example the main parameters) of a system are described in Subsections 2.3.2 and 2.3.3, respectively.

### 2.3.1 Definition of a Parameters Structure

The context-free syntax of an **e\_params** structure input file is given in an *Extended Backus-Naur Form, EBNF* [42]  $\mathcal{E}_{e\_params}$ , where the following additional constructs are used besides the basic Backus-Naur form syntax notation:

- The nonterminals are enclosed between  $\langle$  and  $\rangle$ .
- The terminals are enclosed between “ and ”.
- Subexpressions can be enclosed between round brackets (‘(’ and ‘)’).
- Optional expressions are enclosed between square brackets (‘[’ and ‘]’).
- The Kleene closure (zero or more occurrences of an expression) is denoted by a \* suffix.
- One or more occurrences of an expression is denoted by a + suffix.
- In the list of alternatives, the alternatives are separated by the ‘|’ character.

The *syntactically* correct **e\_params** structure input files can be derived from the starting symbol  $\langle parstruct \rangle$  with the following EBNF rules:

$\langle string \rangle$	$::= \langle printable\_character \rangle^*$	(strings)
$\langle name \rangle$	$::= ( \text{“A”} \mid \dots \mid \text{“Z”} \mid \text{“a”} \mid \dots \mid \text{“z”} \mid$ $\quad )$ $\quad ( \text{“A”} \mid \dots \mid \text{“Z”} \mid \text{“a”} \mid \dots \mid \text{“z”} \mid$ $\quad \mid$ $\quad \text{“0”} \mid \dots \mid \text{“9”} \mid )^*$	(names)
$\langle namelist \rangle$	$::= \langle name \rangle ( \text{“,”} \mid \langle name \rangle )^*$	(name lists)
$\langle digit \rangle$	$::= \text{“0”} \mid \dots \mid \text{“9”}$	(digits)
$\langle intvalue \rangle$	$::= [ \text{“+”} \mid \text{“-”} ] \langle digit \rangle^+$	(integer values)
$\langle exp \rangle$	$::= ( \text{“e”} \mid \text{“E”} ) \langle intvalue \rangle$	(exponents)

$\langle \text{realvalue} \rangle$	::= [ "+"   "-" ] ( ( $\langle \text{digit} \rangle^+ [ "." \langle \text{digit} \rangle^* ]$ )   ( $\langle \text{digit} \rangle^* "." \langle \text{digit} \rangle^+$ ) ) [ $\langle \text{exp} \rangle$ ]	(real values)
$\langle \text{parname} \rangle$	::= "Parameter" ":" $\langle \text{name} \rangle$	(parameter name specifications)
$\langle \text{optname} \rangle$	::= "Option" ":" $\langle \text{name} \rangle$	(option names)
$\langle \text{display} \rangle$	::= "Display" ":" $\langle \text{string} \rangle$	(display string specifications)
$\langle \text{intcondspec} \rangle$	::= ( ">"   ">="   "<"   "<=" ) ( $\langle \text{intvalue} \rangle$   $\langle \text{name} \rangle$ )	(integer condition specifications)
$\langle \text{intcondlist} \rangle$	::= $\langle \text{intcondspec} \rangle$ ( "," $\langle \text{intcondspec} \rangle^*$ )	(integer condition lists)
$\langle \text{inttype} \rangle$	::= "Integer" [ "(" $\langle \text{intcondlist} \rangle$ ")" ]	(integer types)
$\langle \text{realcondspec} \rangle$	::= ( "<"   "<="   ">"   ">=" ) ( $\langle \text{realvalue} \rangle$   $\langle \text{name} \rangle$ )	(real condition specifications)
$\langle \text{realcondlist} \rangle$	::= $\langle \text{realcondspec} \rangle$ ( "," $\langle \text{realcondspec} \rangle^*$ )	(real condition lists)
$\langle \text{realtype} \rangle$	::= "Real" [ "(" $\langle \text{realcondlist} \rangle$ ")" ]	(real types)
$\langle \text{oneormoretype} \rangle$	::= "OneOrMore" [ "(" $\langle \text{namelist} \rangle$ ")" ]	(oneormore types)
$\langle \text{opttype} \rangle$	::= "OptionList" "(" $\langle \text{namelist} \rangle$ ")"	(option types)
$\langle \text{yesnotype} \rangle$	::= "Yesno" [ "(" $\langle \text{namelist} \rangle$ ")" ]	(yesno types)
$\langle \text{typespec} \rangle$	::= "Type" ":" ( "String"   "FileName"   $\langle \text{inttype} \rangle$   $\langle \text{realtype} \rangle$   "Percentage"   $\langle \text{oneormoretype} \rangle$   $\langle \text{yesnotype} \rangle$   $\langle \text{opttype} \rangle$   $\langle \text{name} \rangle$   "DirName"   ) )	(type specifications)
$\langle \text{default} \rangle$	::= "Default" ":" ( $\langle \text{intvalue} \rangle$   $\langle \text{realvalue} \rangle$   $\langle \text{name} \rangle$   $\langle \text{string} \rangle$   ( "yes"   "no" ) ) )	(default values)
$\langle \text{patternspect} \rangle$	::= "Pattern" ":" $\langle \text{string} \rangle$	(pattern specifications)
$\langle \text{groupspec} \rangle$	::= "Group" ":" $\langle \text{string} \rangle$	(group specifications)
$\langle \text{aespect} \rangle$	::= "AllowEmpty" ":" ( "yes"   "no" )	(allow empty value specifications)
$\langle \text{anespect} \rangle$	::= "AllowNonExistent" ":" ( "yes"   "no" )	(allow nonexistent files / directories specifications)
$\langle \text{basedirspec} \rangle$	::= "BaseDir" ":" $\langle \text{name} \rangle$	(base directory specifications)

$\langle parattr \rangle$	$::= \langle parname \rangle  $ $\langle display \rangle  $ $\langle typespec \rangle  $ $\langle default \rangle  $ $\langle patternspec \rangle  $ $\langle groupspec \rangle  $ $\langle allowemptyspec \rangle  $ $\langle allownonexistentspec \rangle  $ $\langle basedirspec \rangle$	(parameter attributes)
$\langle param \rangle$	$::= ( \langle parattr \rangle \text{ “;” } )^*$	(parameters)
$\langle optparams \rangle$	$::= \text{“Parameters” “:” } \langle namelist \rangle$	(option parameters)
$\langle restrspec \rangle$	$::= \langle name \rangle \text{ “(” } \langle namelist \rangle \text{ “)”}$	(restriction specifications)
$\langle restrlist \rangle$	$::= \langle restrspec \rangle ( \text{ “ ” } \langle restrspec \rangle )^*$	(restriction specification lists)
$\langle optrestrs \rangle$	$::= \text{“Restrictions” “:” } \langle restrlist \rangle$	(option restrictions)
$\langle optspecial \rangle$	$::= \text{“Special” “:” } \langle string \rangle$	(option-specific data)
$\langle optattr \rangle$	$::= \langle optname \rangle  $ $\langle display \rangle  $ $\langle optparams \rangle  $ $\langle optrestrs \rangle  $ $\langle optspecial \rangle$	(option attributes)
$\langle option \rangle$	$::= ( \langle optattr \rangle \text{ “;” } )^*$	(options)
$\langle block \rangle$	$::= \{$ $( \langle param \rangle   \langle option \rangle )$ $\}$	(definition blocks)
$\langle parstruct \rangle$	$::= \langle block \rangle^+$	(parameters structure definitions)

The nonterminal  $\langle printable\_character \rangle$  means any character that is printable in the user’s operating system and working environment (it is quite hard, and neither it makes sense, to give a formal definition for this).

Let  $\langle n \rangle$  be a nonterminal. Then the set of terminal strings derivable from  $\langle n \rangle$  is denoted by  $W(\langle n \rangle)$ . Let  $ep \in W(\langle parstruct \rangle)$ . Then  $ep$  is a *correct* parameters structure definition iff it satisfies the following additional context-sensitive restrictions (note that in some cases – where it does not lead to malfunctions – no error or warning messages are produced by the functions):

- None of the *name lists* in  $ep$  contains any *name* more than once.
- All parameters defined in  $ep$  have different names.
- All options defined in  $ep$  have different names.
- If an *integer condition specification* in  $ep$  contains a *name*, then this is the name of a parameter of type **Integer** or **OneOrMore**.
- If a *real condition specification* in  $ep$  contains a *name*, then this is the name of a parameter of type **Real** or **Percentage**.
- The *name list* of a *oneormore type* in  $ep$  contains only names of defined parameters.
- The *name list* of a *option type* in  $ep$  contains only names of defined options.

- The *names* that occur in *type specifications* of *ep* are special type identifiers that are implemented in the `e_params` code.
- Any *default value* occurring in the *parameters* is compatible with the *type specification* of the *parameter* in which it occurs.
- All *parameters* have exactly one *parameter name* and *type specification* and at most one *display*, *default value*, *pattern specification*, *group specification*, *allowempty specification*, *allownonexistent specification* and *base directory specification*.
- Only parameters of type `FileName` and `DirName` can have *allownonexistent specifications* and only parameters of type `FileName` can have *pattern specifications* and *base directory specifications*.
- The *names* appearing in *base directory specifications* must be *names* of a defined parameters of type `DirName`.
- The *name lists* that occur in *option parameters* in *ep* contain only names of defined parameters.
- In all *restriction specifications*, the *name* is the name of a defined parameter of type `Option-List`, and the *name list* in the *restriction specification* contains only names of possible values of this parameter.
- All elements of any *restriction list* in *ep* restrict different parameters.
- All *options* have exactly one *option name* and *display* and at most one *option parameter*, *option restriction* and *option-specific data* field.
- If option *S* of parameter  $P_1$  restricts the possible values of parameter  $P_2$ , then  $(P_2 \text{ set}) \rightarrow (P_1 \text{ set})$  must hold.
- The parameter dependency graph is acyclic (cycles in the parameter dependency graph may cause infinite recursive calls in some functions).

The whitespace (space, tabulator and newline characters) is truncated from the front and the end of the display strings.

Whitespace characters may occur between any subexpressions of every nonterminal, except the nonterminals  $\langle \text{name} \rangle$ ,  $\langle \text{intvalue} \rangle$ ,  $\langle \text{exp} \rangle$  and  $\langle \text{realvalue} \rangle$ .

If the parser of the input file encounters a hash mark (`#`) at a point, then the remaining content of the actual line is regarded as a comment.

### 2.3.2 Storing the Parameter Values

The files that store parameter values for an **e\_params** parameter structure usually have the suffix “.epv”. In such a file, the parameter values are stored in lines of the form “*parameter\_name = parameter\_value*”. Special forms may be defined for special-type parameters such as arrays. The functions *e\_param\_read\_from\_file* and *e\_param\_write\_to\_file* read and write files that are in this form. The lines beginning with a hash mark are regarded as comments.

### 2.3.3 Listing a Set of Parameters

A file which lists a set of parameters for a system is very simple: it just contains the names of the parameters, one name in one line; lines beginning with hash mark are interpreted by the functions of the **e\_params** library as comments. Of course empty lines are allowed.

## 2.4 The Graphical Extension

The graphical extension was implemented in order to provide an easy-to-use interface for setting the parameter values. It is realized using the *GIMP ToolKit*<sup>1</sup> because it is available in different platforms (e.g. Linux and Win32). From version 0.12, GTK version 1.1.4 is required.

An important feature of the extension is that it *checks* the parameter values whether they satisfy the defined conditions and restrictions. The *dependent parameters* can be set up easily as well.

The **ep\_gtk.h** file defines only one function: **ep\_gtk\_set\_params** which has a whole bunch of parameters, namely the following:

**e\_params \*ep** This is the **e\_params** data structure that contains the parameters, the values of which should be set.

**ep\_parameter\_list \*pl** The list of the parameters the values of which should be set. It should include only the *main parameters*; the *dependent parameters* are offered to the user automatically (i.e. only if they must be set). For the difference between the *main parameters* and the *dependent parameters* see Section 2.1. In fact, the type **ep\_parameter\_list** is only a cast to the GLib type **GList**. It should contain character arrays as the data part of the list elements.

**char \*windowtitle** Simply the string that should appear on the title bar of the parameter setting window.

**int ismain** Indicates if the parameters offered for setting are *main parameters* or not. This value is passed on to the *result function* (see below) so that it knows when to perform operations. The automatically displayed windows for *dependent parameters* have this value as 0.

**GtkWidget \*parentwindow** This argument specifies a **GtkWindow** on the top of which the parameter setting window should appear. NULL value for this parameter is allowed. If specified, then the appearing window will be modal to it and will appear justified to the center of it.



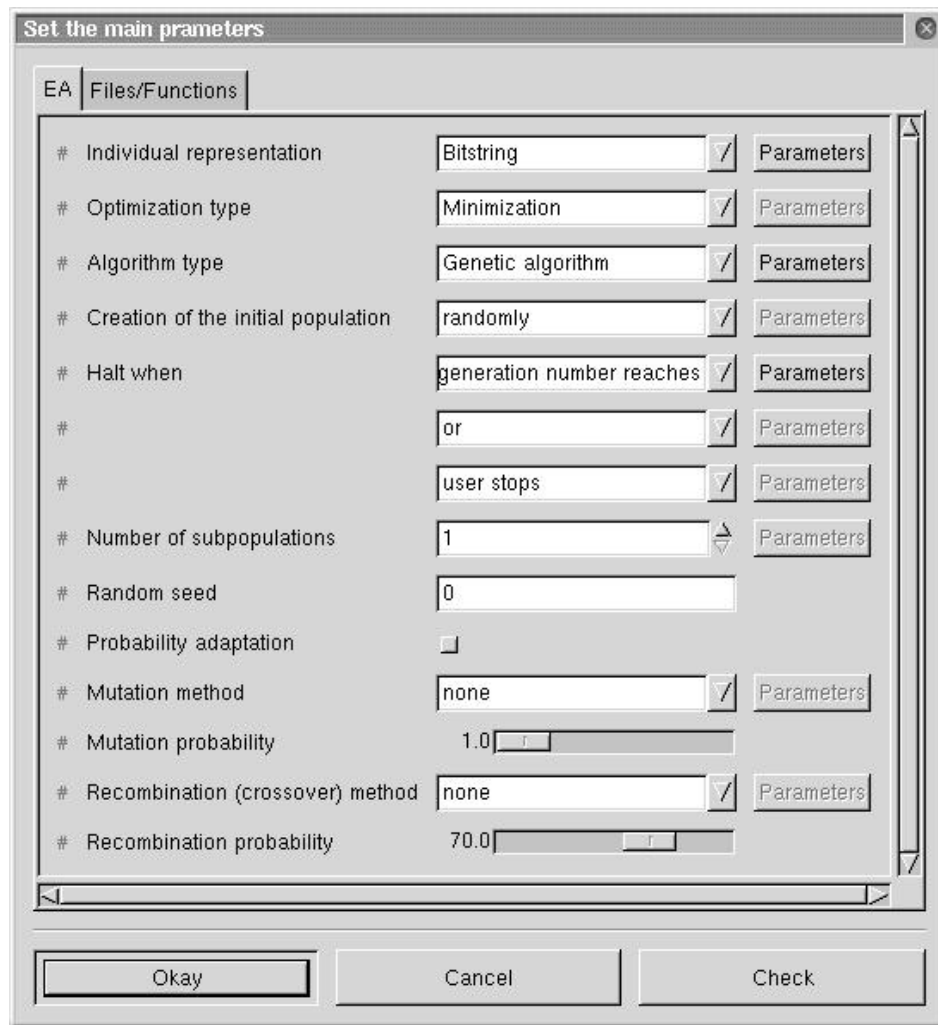


Figure 9: Setting the main parameters

**ep\_gtk\_result\_function rf** When the user presses one of the *Ok* or *Cancel* buttons in the parameter setting dialog box, then this callback function is called so that the main program may know that the parameters are set (or not, in case the *Cancel* button is pressed). The **ep\_gtk\_result\_function** has two integer arguments: The first one tells the type of the button pressed (**EP\_GTK\_RESULT\_OK** or **EP\_GTK\_RESULT\_CANCEL**) and the second contains the value of the **ismain** argument. This is useful for example when the programmer wants to save the modified parameters automatically. Then the parameters should be saved if the *Ok* button has been pressed and the main parameters has been set. If this argument is **NULL**, then no function is called when the buttons are pressed.

**eu\_message\_function mf** The message function is introduced in order to have a common interface for messages towards the user of the programs. A message function has two arguments: The first is a character array containing the message itself, and the second is an integer value determining the type of the message (**EU\_MSG\_MESSAGE**, **EU\_MSG\_WARNING** or

EU\_MSG\_ERROR). A message function can be for example a function that writes the message into a text field. If not specified, the messages are written to the standard error output.

The form of the parameter setting dialog box can be seen in Figure 9. From version 0.11 the parameters are put onto a tabbed pane and the parameters belonging to one group are placed onto the same tab. This makes the GUI more transparent.

Each row shown in the table corresponds to one parameter in the passed parameter list. The second column of the table shows the parameter's display value, and the value itself can be set using the widget placed into the third column. The type of this widget is determined according to the type of the parameter (for example, the value of a parameter of type **Option** can be set with a combo box).

If dependent parameters can be set to a parameter, then the *Parameters* button is enabled in the last column. E.g. if a parameter of type **Option** has a value that has dependent parameters or the value of a **OneOrMore** parameter is set to a value greater than one, then this button can be pressed. When the button is pressed, then a new window appears offering modifications to the dependent parameters (Figure 10).

The first column of each row contains a hash mark which indicates the *correctness* of the parameter in that row. When the hash mark is *yellow*, then the parameter value had been changed since the last check was performed and the new value is not checked yet. A *green* hash mark indicates a correct value of the parameter and a *red* one indicates that the value is incorrect. Note that a red hash mark does not necessarily indicate that the value of the parameter in that row is incorrect, but it is also possible that one or more of its dependent parameters have incorrect values. The *Check* button can be pressed to perform a test of the values of the parameters. This changes the color of each hash mark to either green or red according to the result of the check. A check is performed when the window is first displayed and when the *Okay* button is pressed. Parameters with incorrect values cannot be saved, that means, if a parameter has an incorrect value, then the *Okay* button does not close the window. The user can escape only by entering valid values or pressing the *Cancel* button.

If the user presses the *Cancel* button then all modifications to the parameters are discarded (except the modifications of the dependent parameters if the *Okay* button was pressed in the newly appearing dialog box). Additionally, the *result function* is called to tell the main program that the user has discarded the changes.

By pressing the *Okay* button the user can tell the interface that the changes should be saved. In this case a check is performed and if every displayed parameter has correct values, then the changes are saved and the result function is called to inform the main program that the parameters have been changed. If one or more parameters have incorrect values then the save is not performed, only error messages are given with the specified *message function*.

When the value of a parameter of type **Option** is changed to an option that has defined restrictions to other **Option** parameters, then the combo boxes of the displayed restricted parameters are

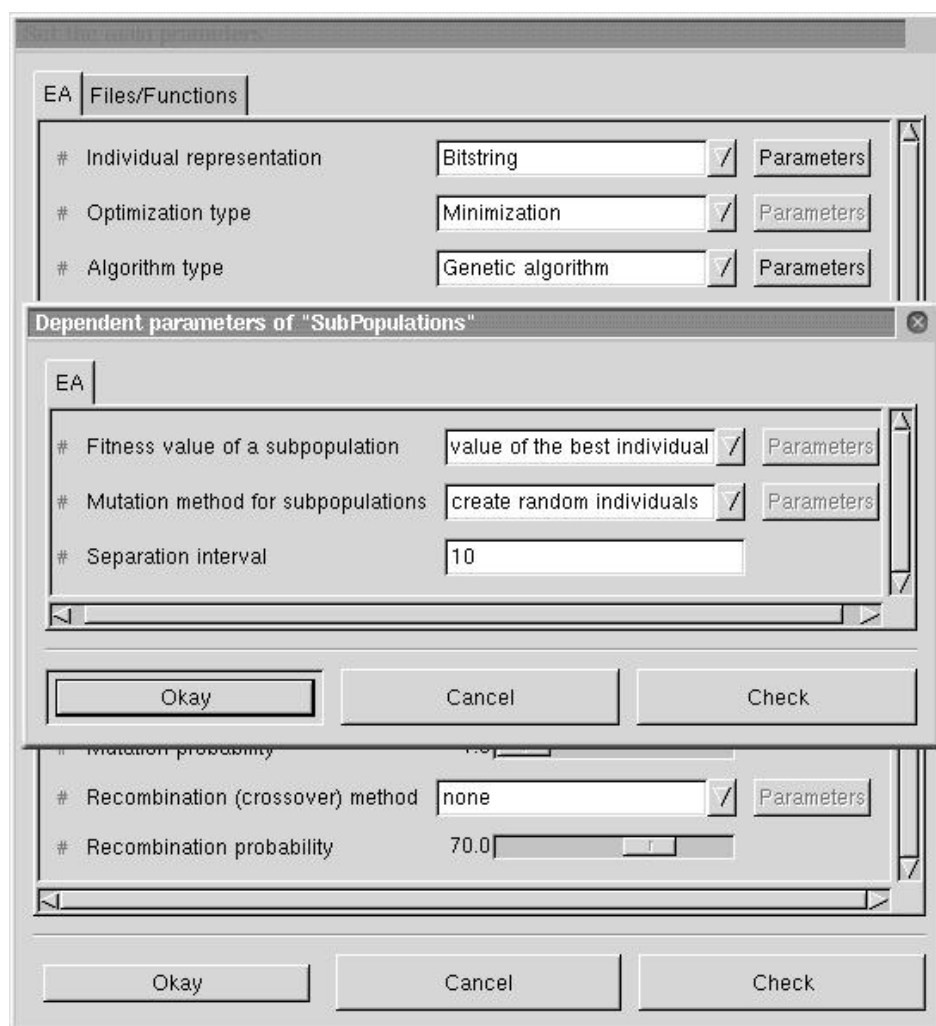


Figure 10: Setting the dependent parameters

updated so that they will contain those options which are enabled by their restrictors.

By pressing the *Browse ...* button right to a file or directory name input field a standard file/directory selection dialog box appears in which the user can select a file/directory easily. The files shown in the file selection's list are filtered with the pattern given in the **FileName** parameter's *Pattern* tag. If a **FileName** parameter has a specified *base directory* and the value of the parameter begins with the value of its *base directory*'s value, then the base directory is chopped from the beginning of the file name.

## 2.5 Final Notes

A part of *GEA*'s parameter structure description file and its graph representation with explanations can be found in Appendix A.

This document presents the **e\_params** data structure in its state as of May 2002. Its applications are always under development, thus the data structure itself may change in the future if the applications

require this. However, these changes will be well documented and backward compatibility will be kept in mind in future developments.

### 3 The GEA System

This section describes the *GEA* (*Generic Evolutionary Algorithms*) system in detail. First, a short history of *GEA* is given: how the idea of creating such a programming library came up and how the class hierarchy has changed until it achieved its present form. Then the class hierarchy of the latest version is presented. This means that the implemented representation forms, their genetic operators and the evolutionary algorithms are also described here.

Kókai, Ványi and Tóth has been involved in evolving fractal images since 1997. The first attempt was to reproduce and improve Koza's results with *Lindenmayer systems* [19, 23]. This project was written in *Java* and did not use any general genetic programming libraries. Then it was realized that *Lindenmayer systems* are capable of describing plants and these plants can be evolved by interactive evolution [20, 38, 39]. At the same time an ophthalmologist came up with the idea of describing the blood vessels of the eye with *L-systems*. This idea led to the *GREDEA* system [21, 40]. These two projects required the evolution of the rewriting rules of the *L-systems* as well as the parameters of them. The most suitable algorithm for the evolution of the rewriting rules is genetic programming, while that for the parameter vectors are evolution strategies. Since the *ANSI C++* programming language was used to implement *TEvol* and *GREDEA* and a programming library which dealt with GPs and ESs at the same time could not be found at that time (in 1998), the design and implementation of a suitable system had begun. This system later was given the name *GEA* (**G**eneric **E**volutionary **A**lgorithms Programming Library).

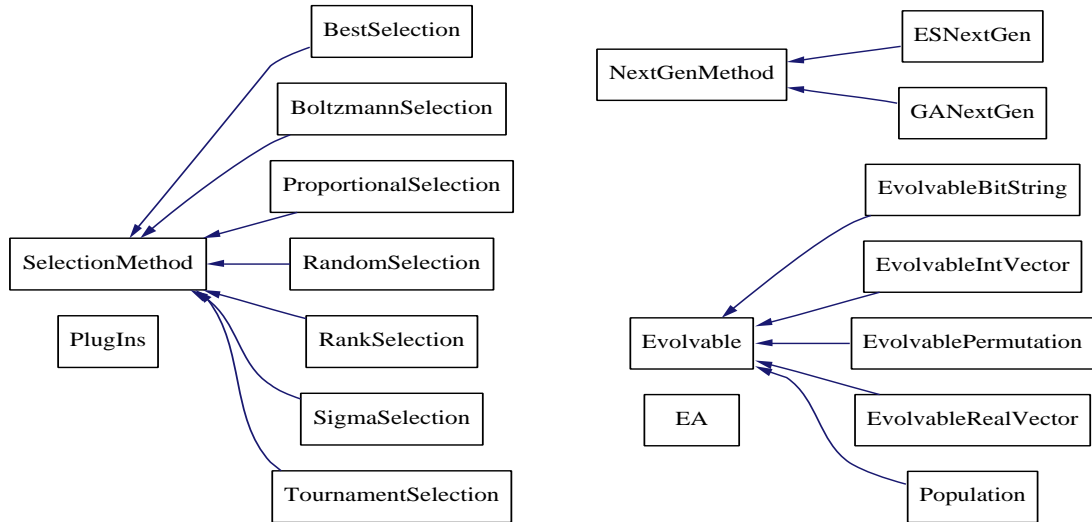


Figure 11: The class hierarchy of the *GEA* system

The class hierarchy of the current version of *GEA* can be seen in Figure 11. Already the first version contained the `Evolvable` abstract class which is the superclass of all evolvable objects, but at that time the integration of new selection methods and evolutionary algorithms was not easy to carry out. The latest version contains the abstract classes `SelectionMethod` and `NextGenMethod`

as well, which define interfaces for selection methods and evolutionary algorithms. These enable the user of the system to easily expand it.

The latest version of *GEA* uses the so-called plug-in technology for the integration of newly implemented classes. The subclasses of `SelectionMethod`, `NextGenMethod` and `Evolvable` have to be compiled and linked as shared libraries (*.so* files on Unix systems and *DLLs* under Windows). When the new plug-ins are registered in the parameter data structure of *GEA* (see Section 2), it will find and load them in case of necessity. As an example, the implementation of a new individual representation form is presented at the end of this section.

Another novelty in the new version is that the population is implemented in separate class, and is also a subclass of `Evolvable` with all the required functions implemented, thus *meta-ES* can be performed more easily.

The application of the `e_params` data structure is also new in *GEA*. This data structure makes the extension of the system easier and provides a hierarchical structure of the parameters. Just as a sidenote, the system has currently 94 parameters (not all of which have to be set of course), which makes having a transparent interface to them reasonable.

For more information on the *GEA* system, see [36] and [37].

### 3.1 The Structure of GEA

In the following, subsections are devoted to describe the functionality of the various classes.

#### 3.1.1 Class Evolvable and the Representation Forms

Class `Evolvable` is the abstract superclass of all evolvable classes: it declares all the methods a class has to implement in order to become an evolvable class and implements a few basic functions. An `Evolvable` object represents one individual in the evolution process.

The *GEA* system uses three genetic operators which must be implemented in all evolvable classes: *Mutate* mutates the individual according to the given genetic parameters. *Crossover* performs crossover with a given individual; here, the other individual is also changed. *Recombine* recombines the given parent individuals and thus creates a descendant. The latter two functions are used by GAs and ESs, respectively.

The functions *PrintOn* and *DrawOn* are provided to print the individual's genotype and phenotype on a given output stream, respectively. This way the individuals can be displayed as graphical objects or saved to files. All evolvable classes have a static function that reads the saved information from a given input stream and recreates the individual.

The functions *clone* and *random* create copies of the object. The difference between them is that by cloning an exact copy is made with similar gene values, while the latter method gives a new individual which can be recombined with the original one. That means, it must have the same type and length, but the chromosome values can be different.

For the sake of efficiency, the system keeps account of the individuals that had been changed with

the genetic operators and recomputes their fitness values as necessary. That means, if an individual had been copied from the current generation to the new one or, due to lower operation probability values, it hasn't been changed, the previously determined fitness value is used.

*GEA* has currently four built-in representation forms, all of which are linked together in the file *libGEArepres.so* (or *GEArepres.dll* on Win32 systems). Remember that the individual representation forms are loaded at run-time by *GEA*. Each of the four data structures is capable of representing chromosomes of a given length, the main difference between them lies in the way the genetic operators are implemented and the genotypes and phenotypes are printed.

Class **EvolvableBiString** encodes individuals as series of bits (0-1 series). Mutation can either flip the selected bit or generate a new random value. Intermediate recombination (for ESs) is not implemented.

The other vital representation form of individuals is real vectors, this is implemented in class **EvolvableRealVector**. Here mutation can add a Gaussian variable to the selected gene or multiply it with a randomly generated value. The implemented recombination/crossover methods are those described in Subsection 1.3.2. In order to keep the genetic operators as effective as possible, the genes are represented internally as real values from the interval  $[0.0; 1.0]$ . Besides, each gene has a lower and upper boundary, which determine the domain of the gene. When the fitness value of the individual is computed, the gene values are scaled from the interval  $[0.0; 1.0]$  to their respective domains.

Fixed-length integer vectors can be represented with the class **EvolvableIntVector**. Here the gene values are nonnegative integer numbers with a higher boundary for each gene.

The genes of class **EvolvablePermutation** are integer values from the interval  $[0; l - 1]$  (where  $l$  is the length of the chromosome): the  $i$ th gene encodes the  $i$ th element of the permutation. The genetic operators are implemented in a way that the modified individual remain permutations: mutation exchanges two randomly selected elements of the vector, while the applied recombination operator is the so-called binary order crossover: it defines two crossover points along the chromosome and copies the segment between them from one of the parents into the first descendant. The remaining genes are copied into the descendant in the order in which they occur in the other parent. The second child is produced by switching the roles of the parents. An example on how this operator works can be seen in Figure 12.

### 3.1.2 Representation of a Population

Class **Population** represents a population (a community of individuals) in the *GEA* system. It stores pointer to its individuals in an array; the elements of this array are sorted by decreasing fitness values.

The class has a few special functions which are needed by some selection methods (for example, some of them use modified fitness values) and some functions which compute statistical characteristics of the population.

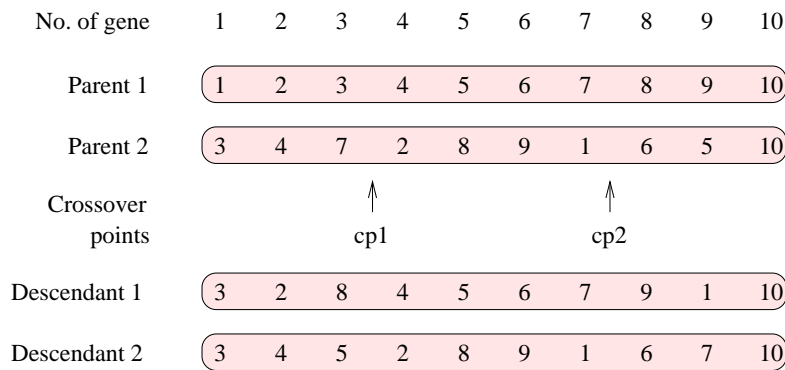


Figure 12: The functioning of the order crossover

Being a subclass of `Evolvable`, populations of populations can be created, thus populations can be evolved, too. This makes *meta-ES* available in *GEA*. The implementation of the genetic operators is very similar to that of bit-strings. The fitness value of a population can be either its best individual's fitness value or the mean of all individuals' fitness values.

The *PrintOn* and *DrawOn* functions of this class first print some data about the represented population and then invoke the individuals' respective functions to print or draw them on the given output stream.

### 3.1.3 Selection Methods

The abstract class `SelectionMethod` is the superclass of all implemented selection methods in *GEA*. Currently, these are exactly the ones described in Subsection 1.1.4. Their code is linked into the shared object *libGEAselmeths.so* (*GEAselmeths.dll*) which is loaded at the creation of the evolution process. When an instance of the requested selection method class is created.

The selection of an individual can be performed by the *Select* method of these classes which returns an integer value: the index of the selected individual. If an error occurs, the return value is  $-1$  and a data member of the class is set to identify the type of the error.

### 3.1.4 Evolutionary Algorithms

As it is explained in Section 1, evolutionary algorithms mostly differ in the way the individuals are represented and new generations are created. Different representation modes are available via the `Evolvable` abstract class (Subsection 3.1.1) and its subclasses. Different methods for creating a new generation are available in *GEA* through the `NextGenMethod` abstract class and its subclasses. Just like in the case of the selection methods, evolutionary algorithms are implemented as plugins and the required class is loaded at running time. Class `NextGenMethod` declares the function *NextGen*, which has to be implemented by the derived classes. An unsuccessful function call returns `NULL` and sets a data member of the object to indicate the type of the error occurred.

Currently, two subclasses of class `NextGenMethod` are available in the *GEA* system: `GANextGen` and `ESNextgen` implement genetic algorithms and evolution strategies as they are described in



Subsections 1.1 and 1.3, respectively. Their compiled code is linked into the shared object file *libGEAeas.so* (*GEAeas.dll*).

### 3.1.5 Class EA

Class **EA** represents an evolution process in the *GEA* system. It has all methods at its disposal that are necessary to handle a population and create new generations. The constructor of the class receives an **e\_params** data structure and according to the settings, it loads the necessary plug-ins and creates the initial population.

The most important member function of the **EA** class is called *NextGen* and it performs the given number of generation steps in the evolution. Since the evolutionary algorithms are implemented in independent classes, this simply means that the *NextGen* function of a subclass of **NextGen** is called and of course the possible errors are handled.

### 3.1.6 Plug-ins

Class **PlugIns** provides convenience functions for handling plug-ins. It is a common plug-in handling interface to all classes which use shared libraries: individual representation forms, selection methods, evolutionary algorithms, fitness functions etc. A static data member is used to keep account of the loaded shared libraries and a function is provided to look up a given symbol in a given shared library; the function loads the object file if needed.

### 3.1.7 Problem-dependent Functions

The most important problem-dependent function in all evolutionary processes is the fitness function. In the *GEA* system, fitness functions are implemented as callbacks and are loaded from plug-ins, like every customizable part of the program code. The callback receives a pointer to an evolvable object as its parameter and should return the result of the evaluation as a real number. Whether this value should be maximized or minimized is determined by the parameters of the evolutionary process.

For some of the optimization problems, it is necessary to perform certain preparatory tasks before the start of the evolution process (e.g. the training and test data sets have to be loaded and preprocessed for a machine learning application). The data structures created by the preparator function and used for fitness calculation have to be properly destroyed after the optimization process had finished and sometimes the task requires maintaining operations between the generations. These tasks can be performed in *GEA* by so-called *preliminary*, *intermediary* and *posterior* functions. Since they usually use common data structures with the fitness function, in most cases their code is linked into the same shared library.

### 3.1.8 The GEA Executable

After the problem-specific implementations (fitness function, in some cases special functions and/or individual representation) are ready, the optimization process can be started by typing

GEA <parameter value file> <path to parameter structure> [shmid]

into the command line. The command-line parameters are the following:

**parameter value file** Contains the values of the parameters of the evolution process: the type of the representation of the individuals, the required evolutionary algorithm, the name of the fitness function etc. A complete example of this file can be found in the next subsection.

**path to parameter structure** The name of the directory that contains the description of the parameter data structure of GEA. A snippet of the current version of this file can be found in Appendix A.

**shmid** This optional argument is a so-called *shared memory identifier*. This is an integer number used to identify shared memory locations in the *Unix System V Interprocess Communication* system. When *GEA* is being run by *GraphGEA*, the calling graphical user interface allocates this shared memory and the two programs communicate through it. This feature is available only on Unix platforms.

When *GEA* is started, it performs the following tasks:

- loads the parameter structure file
- loads the parameter values
- processes the termination parameters of the EA
- processes the logging parameters of the EA, initializes logging facilities
- if an *shmid* is provided in the command line, initializes the communication with *GraphGEA*
- creates the evolution process
- runs the evolution process according to the termination parameters; during the run, manages logging and listens to the messages of the controlling graphical user interface
- after the evolution process had finished, properly frees the used resources and closes the log files

Since the input file of *GEA* is an easily readable and editable text file, the automation of performing several runs of the evolutionary algorithm with different parameters is very simple to carry out. Previous runs can be reconstructed by directly specifying the random seed of the process. Knowing the structure of the log files, the results of the run(s) can be extracted and converted to the desired format with standard text-processing tools. *GEA* is capable to dump the genotypes and the phenotypes of all individuals to given files in certain generation intervals or after the evolution process had finished. All the dumps can be written to one file or the file names can contain the generation number. The genotype dumps can be used to start an evolution process later with a given starting population.

### 3.2 Sample Application: the Traveling Salesman

The well known *traveling salesman problem (TSP)* [18] is used to show, how to apply the *GEA* on a particular problem. In this example, the traveling salesman has to visit a number of towns. He starts from one of the towns and at the end of the route, he has to return to this initial location. The goal is to minimize the length of the salesman's journey. The order in which he visits the towns is a potential solution, and this potential solution can be represented as a *permutation* of the numbers  $1 \dots n$ , where  $n$  is the number of towns to visit.

The number of towns to visit and their coordinates are described in a text file of the following form: the first line of the file specifies the number of towns in the form 'towns  $n$ '. The subsequent lines of the form 'town  $i$   $x$   $y$ ' contain the coordinates of the towns. Here  $i$  is the number of the town ( $0 \leq i \leq n - 1$ ) and  $x$  and  $y$  are the coordinates.

Note that the chromosome length (the number of the towns) is determined in the file described above. This means, that the value specified in the parameter file will be neglected. The TSP task requires a preparatory function that loads the town description file, stores the coordinates of the towns in a data structure that can be accessed by the fitness function and sets the chromosome length in the data structure used by *GEA*, so that when it comes to creating individuals, they will be of the correct size. The code of this function can be studied in Appendix B.

Although the evolution of permutations in the *GEA* system is already implemented through the *EvolvablePermutation* class, the *DrawOn* function has to be modified, so that *GraphGEA* can draw a map of the route according to the given drawing commands (which are described in Subsection 4.3.1) during the evolution process (remember that the *EvolvablePermutation* class doesn't know that the individuals represent routes, so its *DrawOn* function only prints the permutation as a series of integer numbers). For this purpose, a subclass of *EvolvablePermutation* had to be created. It is called *EvolvableTSP* and besides *DrawOn*, it redefines the *clone* and *random* functions and its constructor calls the constructor of its superclass.

After the above described implementation steps, the optimization process can be started by typing

```
GEA evtsp.epv $HOME/GEA
```

into the command line. The parameter settings (*evtsp.epv*) and a snippet from the resulting *GEA* log file can be found in Appendix B.



## 4 GraphGEA

This section describes the *GraphGEA* program in detail. The system uses the graphical object set of the *GIMP ToolKit* (*GTK*<sup>1</sup>) which is written in the C programming language, thus the same language was used to implement *GraphGEA*, too. The functionality is presented beginning with the parameter settings, through the execution of the evolution process and closed with the visualization possibilities.

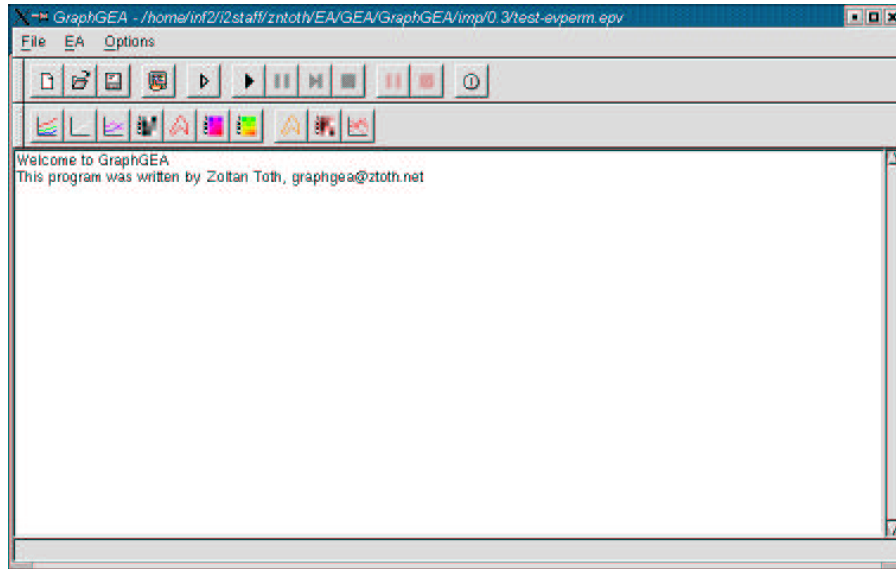



Figure 13: The main window of *GraphGEA*

The main window of *GraphGEA* can be seen on Figure 13. Below the menu bar, it contains two rows of buttons (so-called toolbars), the upper row for managing parameter settings and controlling the evolution process, the lower one for showing and hiding the various visualization windows. The middle of the window is occupied by a large text field where the messages of the application are written. Among others, these messages provide information about the actions between the graphical user interface and the underlying *GEA* process. Error reports are also printed here if one or more of the parameters have invalid values. A menu item of the Options menu serves for clearing the text field. A hint bar can be found at the bottom of the window. If the user moves the mouse over a button or a menu item, a short description of the associated function appears in this area.







### 4.1 Managing the Parameters

When the *GraphGEA* program starts, it loads the parameter structure definition and main parameter list of *GEA*. These files are introduced in Subsection 2.3. The first three buttons of the first toolbar realize the *New-Load-Save* functions known from many applications. The program keeps track of the changes of the parameter values and sends confirmation messages if non-saved information might be lost or used.

The  button brings up a dialog box of the `e_params` data structure (introduced in Subsection 2.4) with the main parameters of the evolution process. The main parameters are divided into three groups: the representation form of individuals, halting condition, applied genetic operators etc. belong to the first group. The second group contains the program-specific parameters such as the fitness function and plug-in file names, while the third group is for specifying logging options and log files.

## 4.2 Running the Evolution Process

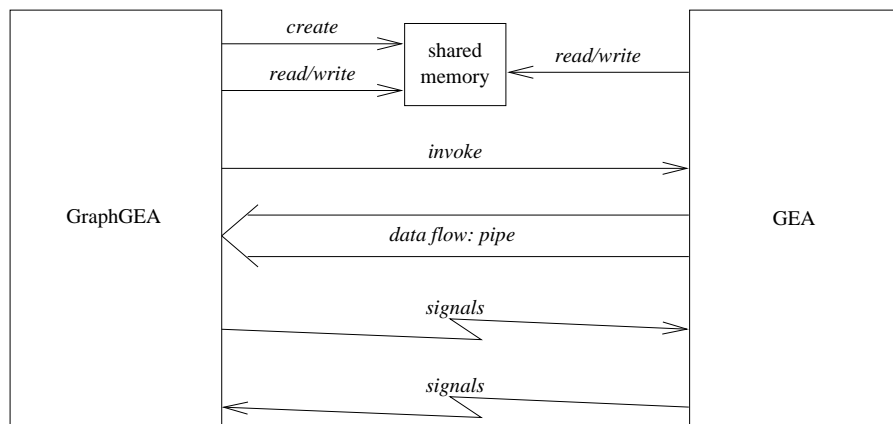
After the parameters are set, the evolution process can be started and controlled with the buttons that resemble to those of CD or cassette players. Their functions are the following:

-  Starts the evolution process. The mechanism of the interaction between *GraphGEA* and *GEA* is described below.
-  If the evolution process is running, this button can be used to suspend it after the current generation had been processed. The suspended process can be resumed by pressing the button again.
-  If the evolution process is suspended, it can be executed generation by generation with this button, that is, this button proceeds one generation in the process. This enables the user to conveniently analyze the progress of the EA.
-  Causes the evolution process to stop after the current generation. After it is clicked, *GraphGEA* sends an appropriate signal to *GEA* and waits until it exits before enabling other actions for the user.
-  and  The two red buttons of the second group of controls can be used to pause/resume and stop the running evolution process immediately, i.e. without finishing the current generation and writing the results to the log files.


As it can be expected from every worthy application, the above listed buttons have their counterparts in the menu system of the program and they are enabled only if they are meaningful in actual state of the evolution process.


When the evolution is started, the graphical user interface invokes the *GEA* program as its child process and communicates with it during the run. The relationship and interaction between the two programs are depicted on Figure 14.

At the start of the evolution run, *GraphGEA* first checks whether the current parameter settings are correct and saved. If there are incorrect parameter values then it lists the error messages of the `e_params` data structure in its text area. In the case when the parameter settings had not been stored since the last changes, it asks the user if they should be written to disk before starting *GEA* or not.

Figure 14: The interaction between *GraphGEA* and *GEA*

Starting the *GEA* program takes the following steps: first, a shared memory area is requested from the operating system, the messages between of the two programs are stored here before they are processed. Then *GraphGEA* creates a child process with the *fork* system call and the child invokes *GEA* with the necessary command-line arguments (see Subsection 3.1.8) using the *execvp* function. The parent process opens a pipe to the child and registers an event handler to manage its output (remember that by default, *GEA* writes its log onto its standard output channel). Signal handlers are also registered by both programs, for they use the SIGUSR1 signal to let each other know about messages waiting on the shared memory area for processing.

Once the evolution process is started, the graphical user interface can send messages to it with system signals. Suspending and stopping *GEA* after the current generation and resuming a suspended run is done by placing the appropriate message identifier into the shared memory and sending a SIGUSR1 signal to the child process. *GEA* also sends a message to *GraphGEA* with the same mechanism each time a generation is ready, this can be used for example at the step-by-step execution to enable/disable control buttons at the right time. Immediate suspend/resume and stop of the evolution process is done by sending SIGSTOP/SIGCONT and SIGTERM signals, respectively. *GraphGEA* is watching the SIGCHLD signal, too, so that it knows when *GEA* exits. The off-line visualization of an already finished evolution run can be initiated with the  button. For this, the parameter settings and the log file of the run are needed. When these two files are given, *GraphGEA* invokes a simple program (called *GEmul*) that echoes the log file to its standard output and communicates with the graphical user interface the same way as *GEA* does. In short, *GEmul* emulates the behavior of *GEA*, thus the suspension, step-by-step execution of the EA, etc. are all possible. When a complete reconstruction of an evolution run is needed (a reason for this can be for example that the user wants to have more detailed logs), the original parameter settings are needed and the evolution process should be started with the random seed which was used in the original run (the used random seed is always printed into the log file).

After the work with *GraphGEA* is finished, the user can leave the program with the  button or by pressing *Ctrl-Q* on the keyboard.

### 4.3 The Visualization Plug-ins

The visualization options of the *GraphGEA* system are implemented as so-called plug-in modules (plug-ins for short). Plug-ins are compiled code segments, modules, which are not loaded by the operating system when the application is started, but the application itself can load them if it needs their functionalities. The most important advantages of plug-ins against traditional objects linked to the application are the following:

- Since they are stored in separate files (DLLs – dynamically loaded libraries – under Windows systems and `.so` – shared object – files in Unices), several applications can use the same files without the need of having the same compiled code stored several times on the hard disks.
- If an application does not need a certain module during a particular run, the code of that module don't have to be loaded and initialized, thus the start-up speed can be increased and the program can economize on system resources.
- Due to the standard interface of loading and using shared libraries, a part of a program can be improved by updating the plug-in file, thus avoiding the complete reinstallation.
- The standard interface also enables the easy and fast extension of applications, it is usually done by just copying the compiled code into a predefined directory and in some cases modifying configuration files.

Besides the advantages listed above, the generation of shared objects and their usage require only a very little implementation work from the program developer: in the compilation and linking, one only has to use a few additional command line arguments of the linker, and loading and using the plug-ins in the main program necessitate the call of only three simple library functions.

The main reason of using plug-ins in the *GraphGEA* program is extendibility: new visualization plug-ins can be added with minimal modification of the existing program code. Each plug-in has a corresponding button in the second toolbar which shows and hides its visualization window. These buttons are enabled according to the successfulness of the loading and initialization of the plug-ins at start-up.

*GraphGEA* looks up the following functions in each loaded plug-in:

**create** is invoked at the start-up of the program, its main role is to set up the graphical elements of the plug-in's window

**init** is called just before the evolution process is launched when all the parameters of the optimization task are known; its purpose is cleaning up its data structure after the previous run and carrying out parameter-dependent initialization steps (e.g. some visualizations need to



know the representation form and chromosome length of the individuals which can vary from run to run)

**new\_data** is used to pass the output of *GEA* to the plug-ins; each time a line of text arrives from the pipe, the GUI hands the line on without any modifications

**done** should free the allocated memory and destroy the graphical elements, for this function is called before the *GraphGEA* program exits

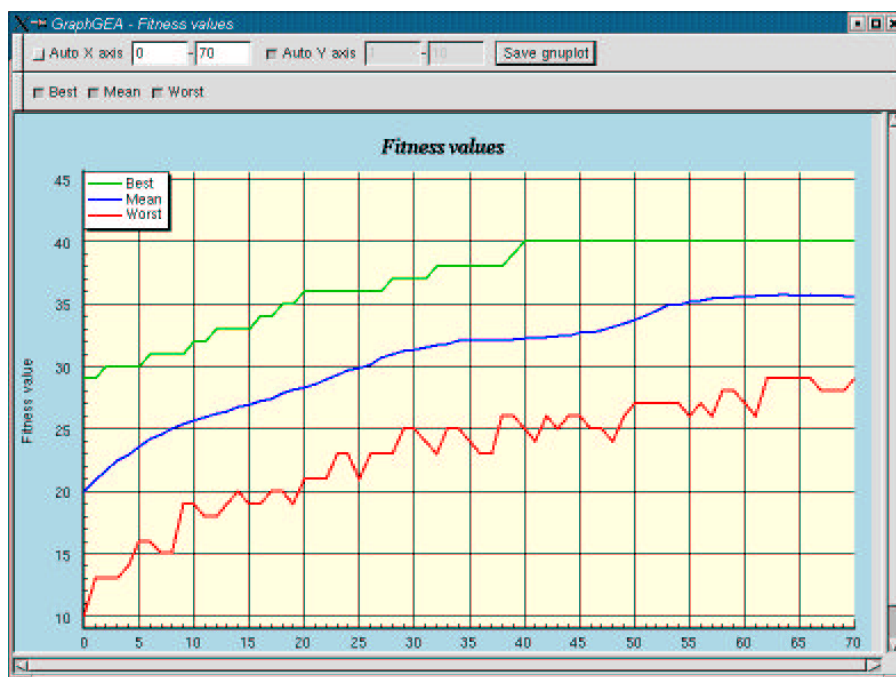
All of the above four functions have to be implemented by each plug-in but this means mostly the calling of provided method-dependent helper functions with appropriate arguments. The method-dependent details are discussed in Subsection 4.3.1.

The evolution process (that is, the *GEA* program) runs as a child process of the graphical user interface and *GraphGEA* is reading its output from a pipe. Each time when the input handler function of the GUI gets a line from the pipe, it invokes the appropriate standard input handler function of each loaded plug-in. Each visualization method can decide whether the received information is relevant for its purposes or not and carry out the necessary actions (updating its database, executing certain drawing commands, etc.); for this reason it is very important to set the logging parameters of the evolution process correctly. If *GEA* does not print an information into the log (and its standard output) then obviously this information will not be passed on to the plug-ins which might need them. On the other hand, if the user finds the output of one or more plug-ins irrelevant to his/her work, then turning off the corresponding logging options can be reasonable because it can increase the performance of the GUI. If the information turns out to be important in a later phase of the research, the evolution run can be reconstructed given the evolution parameters and the random seed are still available. The visualization windows and their necessary parameters are listed in Table 2.

#### 4.3.1 Methods of Visualization

As it is described in Subsection 1.5, there are basically three different visualization methods discussed in this thesis: plots, color coding, and drawings. Each of these three methods use the common plug-in interface but of course their behavior and look are different, so the implementation of the provided *create* and *new\_data* functions differs, too. Next, the look of the three plug-in windows, their functionality and the helper functions are described.

A plot window of *GraphGEA* is depicted on Figure 15. It is capable of showing several diagrams in one coordinate system, each of which can be shown and hidden individually with the checkboxes of the second toolbar. In the example, the best, mean and worst fitness values are shown with different colors and the legend is shown in the top-left corner of the plot. By default, the lower and upper boundaries of the X and Y axes are computed automatically according to the ranges of the shown values. This computation considers only the shown diagrams. The boundaries can be set manually in the first toolbar by unchecking the appropriate checkboxes and entering the values

Figure 15: A plot window of *GraphGEA*

into the input lines next to them. The plot is updated as soon as the input fields lose the focus (i.e. when the user clicks outside the entry field). The actual view of the diagrams can be saved in gnuplot format with the 'Save gnuplot' button. The *gnuplot* program can convert its input into various well-know graphical formats, e.g. the encapsulated postscript file created from the plot shown on Figure 15 is displayed on Figure 16.

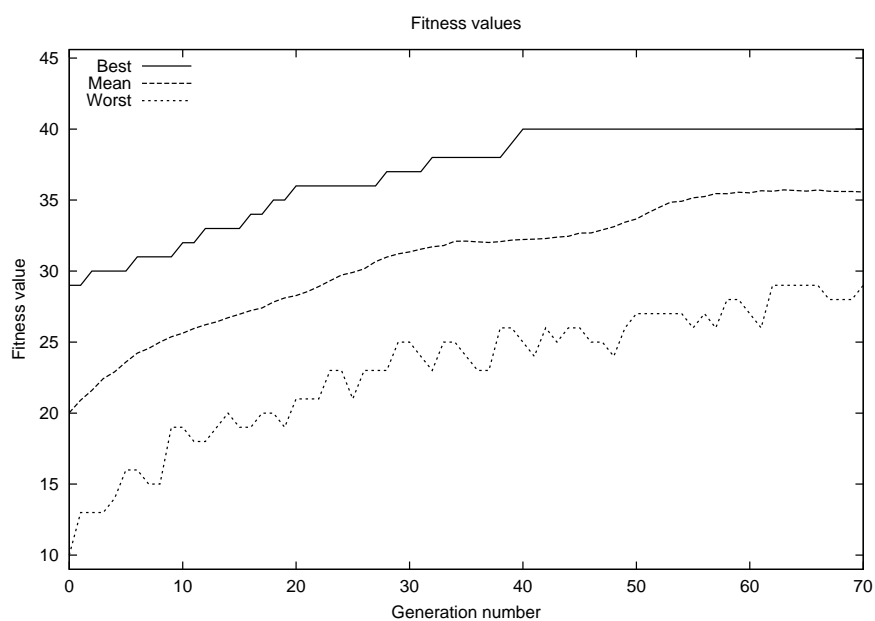


Figure 16: The gnuplot output of the plot window

When implementing a new plot plug-in, the *create* helper function has to be provided with the following information:

- The title of the plot window;
- The labels of the X and Y axes;
- The number of diagrams in the plot;
- The names of each diagram (these will appear in the second toolbar next to the on/off checkboxes);
- The colors of the diagrams given with their RGB components.

The helper function creates the window and handles all events occurring inside it. The only function that requires a little implementation work is *new\_data*. It has to process the output of *GEA* and filter out the necessary information. New points can be added to the plot with the *new\_plot\_data* helper function. Its two arguments are the X coordinate of the new points and an array containing the Y coordinates. The number of the elements of the array has to be equal to the number of the diagrams in the plot. For example, the *new\_data* function of the ‘Fitness values’ plug-in gets the generation number and the best, mean and worst fitness values from its input and calls the *new\_plot\_data* function with the generation number as the X coordinate and the three fitness values as the Y coordinates of the new points.

With the color coding technique, it is possible to depict a large amount of values in a transparent way: they are displayed with different colors, not with numbers. A color coding visualization window can be seen on Figure 17. Arbitrary numerical values can be shown in the form of a two-dimensional array with additional explanatory columns on the left and the right side of the color matrix.

Similarly to the plot-type plug-ins, helper functions are provided for the color coding method, too. The most important are again *create* and *new\_data* but the *init* function may also have plug-in dependent special tasks. To create a window, the following attributes have to be defined:

- The title of the window;
- The red, green and blue components of the colors representing the lowest and highest displayed value;
- Whether there are fixed lowest and highest displayed values in the table or these values should be computed automatically as new data are added;
- In the case of fixed low and high boundaries, these need to be specified;
- If the boundaries are computed automatically, this can be done either separately for each column or all columns can share the same limits;

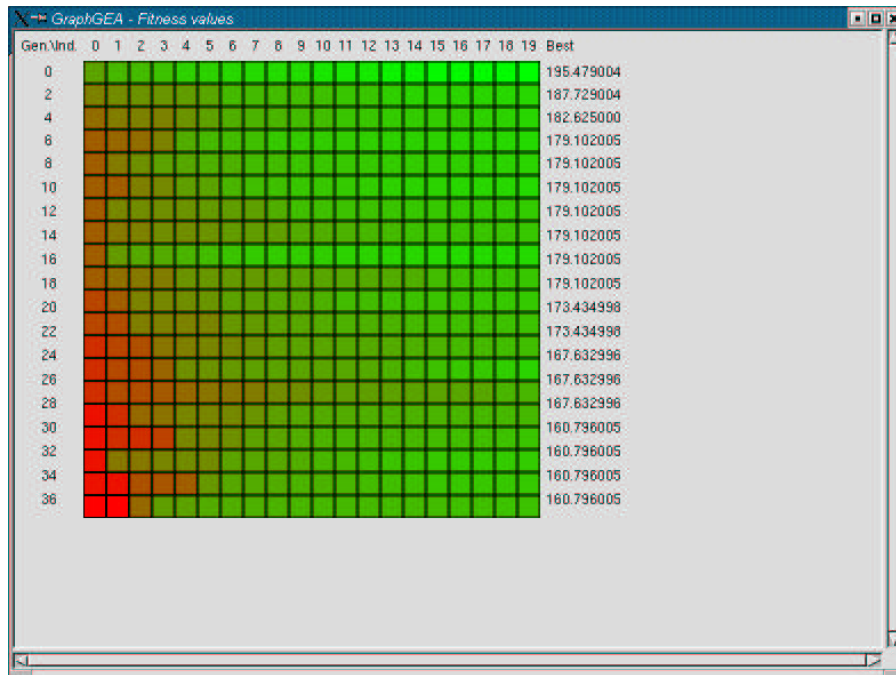


Figure 17: A color coding window of *GraphGEA*

- The titles of the first and last columns of the table.

The total number of columns is not specified at the creation of the window, since this can be a problem-dependent parameter. For this reason, the *init* function has to set this value. The *new\_cc\_data* function can be used to add a new row to the table: its first two arguments are the values to be displayed in the first and last columns, and the third argument is an array containing the elements to be visualized with colors.

There are two possibilities of displaying the individuals' phenotypes in the *GraphGEA* system: by printing the phenotypes as a series of strings into a text field or by using the drawing commands of the program. The phenotype visualization plug-ins choose between these two methods according to the representation form of the individuals. A window with a solution of the TSP problem can be seen on Figure 18. One individual is displayed at a time and the user can use the scrollbar at the top of the window to select from the available individuals.

The *create* helper function has only two arguments: the title of the window and the string that is shown as the name of the scrollbar. It is the task of the *init* function to create and display the appropriate drawing widget (the graphical elements of GTK are called *window gadgets*): if the representation is known as a drawable one (that is, its *DrawOn* function produces a series of drawing commands) then a drawing area is created, otherwise a text field will appear. The list of available individuals can be extended by the *new\_draw\_data* helper function; its two arguments are the string that appears between the scrollbar and a NULL-terminated array of strings describing the new phenotype. If the phenotypes are displayed in a simple text field then these strings will appear there; if the individual is displayed by vector graphics then each of the strings should

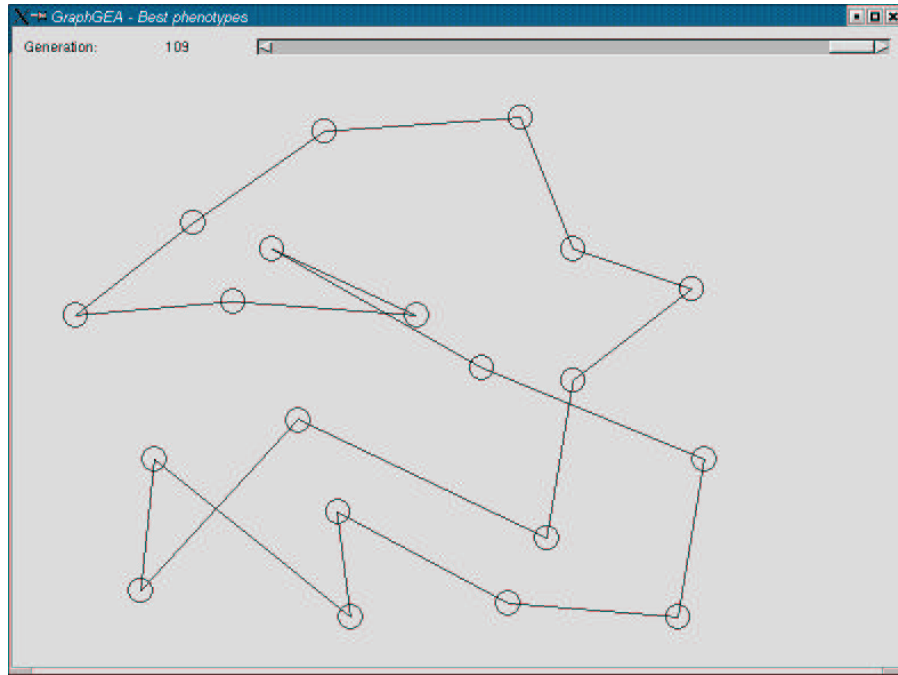


Figure 18: The phenotype of an individual drawn by *GraphGEA*

contain one of the following drawing commands:

**B  $x_1$   $y_1$   $x_2$   $y_2$**  This command determines the boundaries of the drawing area. The individual is drawn in a way that the primitives within the boundaries are always visible in the plug-in window.

**P  $x$   $y$**  Puts a point with coordinates  $(x, y)$ .

**L  $x_1$   $y_1$   $x_2$   $y_2$**  Draws a line from  $(x_1, y_1)$  to  $(x_2, y_2)$ .

**R  $x$   $y$   $w$   $h$   $f$**  Draws a rectangle with the upper-left corner being in  $(x, y)$ , width  $w$  and height  $h$ . If  $f$  is equal to  $T$  then the rectangle will be filled.

**A  $x$   $y$   $w$   $h$   $a_1$   $a_2$   $f$**  Draws an arc. The upper-left corner will be at  $(x, y)$ , the width and height of the arc will be  $w$  and  $h$ , respectively. The starting angle of the arc is determined by  $a_1$ , the length by  $a_2$  (that is,  $a_2$  is the ending angle relative to  $a_1$ ). The values of the angles should be between 0 and 360, 0 being at 12 O'clock, the positive direction is counter-clockwise. The last argument ( $f$ ) determines the filling:  $T = yes$ ,  $F = no$ .

**Y  $n$   $f$   $x_1$   $y_1$   $x_2$   $y_2$  ...  $x_n$   $y_n$**  Draws a polygon. First the number of vertices ( $n$ ) is given, then the filling parameter, at the end follow the coordinates of the vertices.

**S  $x$   $y$   $s$**  Puts the string  $s$  at the coordinates  $(x, y)$ ;  $x$  and  $y$  are the left edge and the baseline of the string, respectively.

### 4.3.2 The Implemented Plug-ins

Table 2 shows the list of the currently available visualization plug-ins of the *GraphGEA* system. Besides the name, icon of the show/hide button and type of the visualization tools, a short description and a list of pertinent logging parameters are listed. If necessary, the roles of the parameters are also explained.











Name	Type	Description	Parameters
Fitness values 	plot, course	Diagrams of the best, mean and worst fitness values plotted against the generation number.	PrintBMWFitness
CPU times 	plot, course	A diagram showing the used CPU time of the evolution process plotted against the generation number.	PrintCPUTimes
Fitness variance 	plot, course	A diagram showing the variance of the fitness values in the population against the generation number.	PrintFitnessVariance
Best genotypes 	color coding, course	A table containing the color coded gene values of the best individuals of the generations. The lowest gene values correspond to black cells, the highest gene values to white cells. The first and the last columns show the generation number and the fitness value of the depicted individual, respectively. This plug-in is applicable only in the case of fix chromosome length and numerical gene values.	PrintGenotypes WhenGeno GenoPrintInterval
Best phenotypes 	draw, course	The phenotypes of the best individuals of the generations. The individual can be selected by the generation number (this value appears next to the scrollbar).	PrintPhenotypes WhenPheno PhenoPrintInterval
Gene variances 	color coding, course	Shows the variances the of values of each gene in the population. Blue corresponds to low variance, red corresponds to high variance. The first and the last column contain the generation number and the fitness value of the best individual in the generation, respectively.	PrintGeneVariances PrintBMWFitness
All fitness values 	color coding, course	The fitness values of all individuals are shown in one table. High fitness values with green, low values with red. The first and last columns of the matrix show the generation number and the fitness value of the best individual, respectively.	PrintFitnesses PrintFitnessesInterval
All phenotypes 	draw, status	Offers all phenotypes of the current generation for viewing. The individual can be selected by its position in the population. It is useful to set the <i>UseAllPhenosPlugin</i> parameter to false if this plug-in will not be used because displaying several individuals in every generation is a very resource-consuming task for <i>GraphGEA</i> .	PrintPhenotypes WhenPheno PhenoPrintInterval UseAllPhenosPlugin
All genotypes 	color coding, status	Displays all genotypes of the current generation. The low and high gene values are represented by white and brown colors, respectively. The first and last columns show the number of the individual and its fitness value. It is useful to set the <i>UseAllGenosPlugin</i> parameter to false if this plug-in will not be used because displaying several individuals in every generation is a very resource-consuming task for <i>GraphGEA</i> .	PrintGenotypes WhenGeno GenoPrintInterval UseAllGenosPlugin
Current gene values 	plot, course	The lowest, average and highest values of each gene are plotted against the gene number.	PrintGenotypes WhenGeno GenoPrintInterval UseLHGenesPlugin

Table 2: The currently available visualization plug-ins of the *GraphGEA* system





## 5 Related Work

In this section some other EA visualizing/controlling tools and the differences between them and *GraphGEA* are discussed. It must be emphasized that the main purpose of *GEA* is solving real world optimization problems and *GraphGEA* is a graphical user interface that supports analysis of the evolution process's behavior and education. *GraphGEA* does not affect the efficiency of the underlying evolution process.

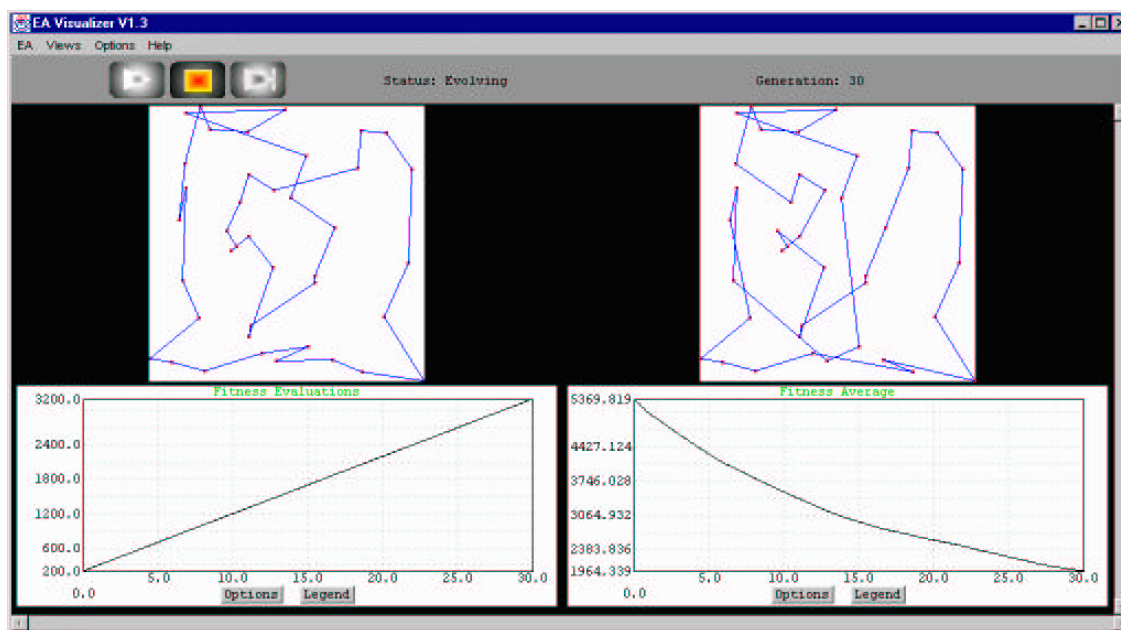


Figure 19: The *EA Visualizer* working on a TSP

The *EA Visualizer* [4] is a platform independent tool for running and visualizing evolutionary algorithms written in the Java programming language. It has a wide variety of convergence graphs and a special tool called *GraphDrawer* is provided to create various plots. Some of its disadvantages is that chromosomes can be depicted only in the case of binary representations and the phenotypes of the individuals can be drawn for some determined problems only, e.g. the traveling salesman problem (Figure 19). Since all individual representation forms in *GEA* have functions to output the genotypes of the individuals, these can be shown in every case. The internal drawing language of *GraphGEA* and the phenotype output of *GEA* enable depicting the phenotypes of solutions of any problem (see Figure 18). On the other hand, the *EA Visualizer* is able to handle multiple runs with different parameter settings. The evolutionary algorithms are implemented in Java and assembled from ‘building blocks’ and this makes the system easily extendable, although genetic programming is not supported.

*EvolVision* [12] is a client-server based tool to visualize the output of *Mathematica notebooks* which use the *Evolvica* system [17]. The client-server architecture is very useful to make the EA process independent from the visualization tool, but *EvolVision* cannot control the run of the evolution process. It is able to perform off-line and on-line visualization as well and can depict any genomes

and a range of various graphs. A plug-in interface is used for possible extensions. A disadvantage of the system is that it does only the visualization of the results and has no real connection with the running evolution process. The graphical components of the Java language (*Swing*) are slow and require much memory and time for visualizing larger data sets.

*GIGA* [7] is what its name stands for: a **G**raphical user **I**nterface for **G**enetic **A**lgorithms. That is, only GAs are implemented and the evolution process can be controlled via the GUI to some extent: some parameters of the GA can be set in the control windows. It is able to do off-line and on-line visualization of some graphs and the algorithm's internals, but the latter figures are hard to read because the user has to find the crossover points and mutated genes himself, as these are not shown directly (see Figure 20). The phenotypic representation of the individuals is also available, but being completely problem dependent, this visualization has to be implemented by the user. The system is written in the C programming language using the Unix/X11 environment and the OSF/Motif GUI library, thus its portability is strongly bounded. It is possible to implement new algorithms for *GIGA*, but these must meet the quite strict restrictions of a given prototyping interface.

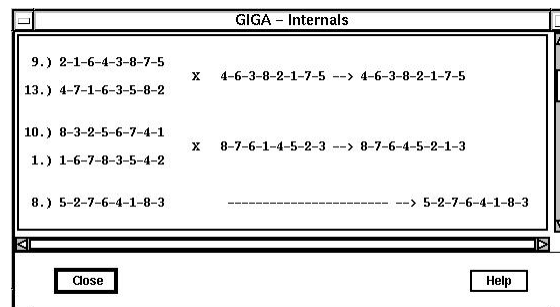


Figure 20: The 'internals' window of *GIGA*

*GeatBox* [31] is another very promising visualization tool with various plots and graphs for depicting the *course* and the *state* of the running EA, but its disadvantage is that it is written in the *Matlab* computer algebra system, thus the user must know *Matlab* to use *GeatBox* efficiently. It is able to do off-line and on-line visualization as well. GAs and ESs are implemented in the system but it is not able to visualize GPs and because of the lack of extendibility, the option to make experiments with these latter algorithms is completely missing. Only the genotypic representation of the individuals can be depicted, the phenotypes cannot be visualized with this tool.

*Gonzo* [6] is a tool for visualizing genetic algorithms written in LISP. The number of users of this system is strongly bounded because of the choice of the programming language, since LISP is not so widespread as C/C++ or Java. *Gonzo* is designed to explain the search behavior of the algorithm, so the search space and its representation stand in the center of this program. It can depict some graphs and plot how the gene values develop during the EA process (note that this technique is not applicable for genetic programming). Besides *GraphGEA*, this is the only system with total control over the running evolutionary algorithm: the user can start, stop, pause, resume the GA or execute it by generation steps.

## 6 Summary

In this document the *GraphGEA* system, a visualization extension of the *Generic Evolutionary Algorithms* programming library is presented. The first section covers the theoretical fundamentals of *evolutionary algorithms*. An evolution process has many, sometimes intricately interrelating parameters. A data structure for handling and extending this parameter structure is presented in Section 2. The *GEA* system is described in Section 3, while Section 4 deals with the *GraphGEA* system itself. Finally, a view on related work is given in Section 5.

*GraphGEA* has two main objectives: first, it helps the researchers to analyze and understand the search behavior of evolutionary algorithms, and second, it is a very good tool for students to get acquainted with these optimization methods. Since *GEA*, the underlying EA implementation, is an efficient and easy-to-use optimization utility, the graphical user interface can be used just to set all the parameters of an optimization correctly, thus the GUI can be useful in solving real industrial optimization problems.

The graphical user interface can be disposed into three main parts. Solving an optimization problem with an evolutionary algorithm always begins with the selection of the representation form of the individuals, the most suitable evolutionary algorithm and other parameters. *GraphGEA* offers very handy dialog boxes for setting all the parameters and it also assures that the values are correct. If one wants to analyze the optimization process, looking at the log files after the run is not always the best and most convenient way. The implemented software offers the possibility of the interactive execution of the evolution run, this way the user can suspend the process at any time and look at its course and status. The huge amount of numerical data describing an evolution run can be displayed by various visualization plug-ins in the *GraphGEA* system. The visualization windows provide a run-time look at the evolution process: the user can observe how the individuals change during the optimization, how much system resource is consumed, what is the diversity of the population, etc. Since the visualization methods are implemented as plug-ins and they have a common programming interface, it is very easy to expand the GUI with new methods.

Looking at the work done in the field of the visualization of evolutionary algorithms, the most important conclusion is that most of the available tools are very specific in terms of the implementation language and the range of suitable problems. Throughout the design of the *GEA* and *GraphGEA* systems, the two most important objectives were efficiency and applicability. This is the reason of the selection of the C and C++ programming languages and the application of the plug-in technology. Together with the used parameter structure, these make the programs able to solve and visualize a wide range of optimization problems.



## Acknowledgments

Hereby I say thanks to all the people who provided their help in the development of the *GraphGEA* system. This list is created in no particular order, and cannot be complete as well, since there are many people in the research field who had ideas regarding the visualization of evolutionary algorithms.

The first thanks must go to *Dr-Ing. Gabriella Kókai* at the Department of Computer Science II, Programming Languages of the Friedrich-Alexander University of Erlangen-Nürnberg, who had introduced me into this very interesting area of science and have been my supervisor for the last couple of years. Her ideas had improved my work in a very efficient way and she always promoted my work at conferences and at the university.

I owe thanks to *Prof. Dr. Hans Jürgen Schneider*, head of the Department of Computer Science II, Programming Languages of the University of Erlangen-Nürnberg, who provided me with a very peaceful and comfortable working place at his department to finish this document. I say thanks to the members of the staff of the department, I have really felt that I was a part of their community during the completion of my thesis.

I am grateful to *Dr-Ing. Heide Wichmann* at the Department of Computer Science X, System Simulation of the University of Erlangen-Nürnberg, the student advisor of the Computational Engineering program, who solved some of my bureaucratic problems.

My stay and study in Germany would have not been possible without the financial support of *Siemens* and the *German Academic Exchange Service (DAAD)*.

Last, but not least, I say thanks to *Dr-Ing. Jörg Nilson* at 3SOFT GmbH, Erlangen, for his useful pieces of advise in the C++ programming language and for the corrections of parts of my thesis.



# Appendices

## A A Detail of GEA's Parameter Structure

A part of the graph representing the parameter structure of the *GEA* system can be seen on Figure 21. The graph has been generated from the parameter structure description file. The selected detail depicts the individual representation form and the applicable genetic operators, which are described in the parameter description file as follows:

```
{
  Parameter: Representation;
  Display: Individual representation;
  Type: OptionList(EvolvableBitString, EvolvableRealVector,
    EvolvableLogicRuleSet, EvolvableIntVector, EvolvablePermutation,
    EvolvableTSP);
  Default: EvolvableBitString;
  Group: EA;
}

{
  Parameter: ChromosomeLength;
  Display: Chromosome length;
  Type: Integer(>0);
  Default: 1;
  Group: EA;
}

{
  Parameter: MutationProbability;
  Display: Mutation probability;
  Type: Percentage;
  Default: 1;
  Group: EA;
}

{
  Parameter: MutationType;
  Display: Mutation method;
  Type: OptionList(mutnone, flip, mutrandom, add, multiply, mutexchange);
  Default: mutnone;
  Group: EA;
}

{
  Option: EvolvableBitString;
  Display: Bitstring;
  Parameters: ChromosomeLength;
  Restrictions:
    MutationType(mutnone, flip, mutrandom) |
    RecombinationType(recline, point, uniform, discrete);
  Special: GEArepres;
  # representation is implemented in libGEArepres.so/GEArepres.dll
}

{
  Option: EvolvableLogicRuleSet;
  Display: Logic rule set;
  Parameters: MaxRuleNumber, TrainigSetFile, TestSetFile, GoalClasses;
  Special: evlrs;
  # representation is implemented in libevlrs.so/evlrs.dll
}
```

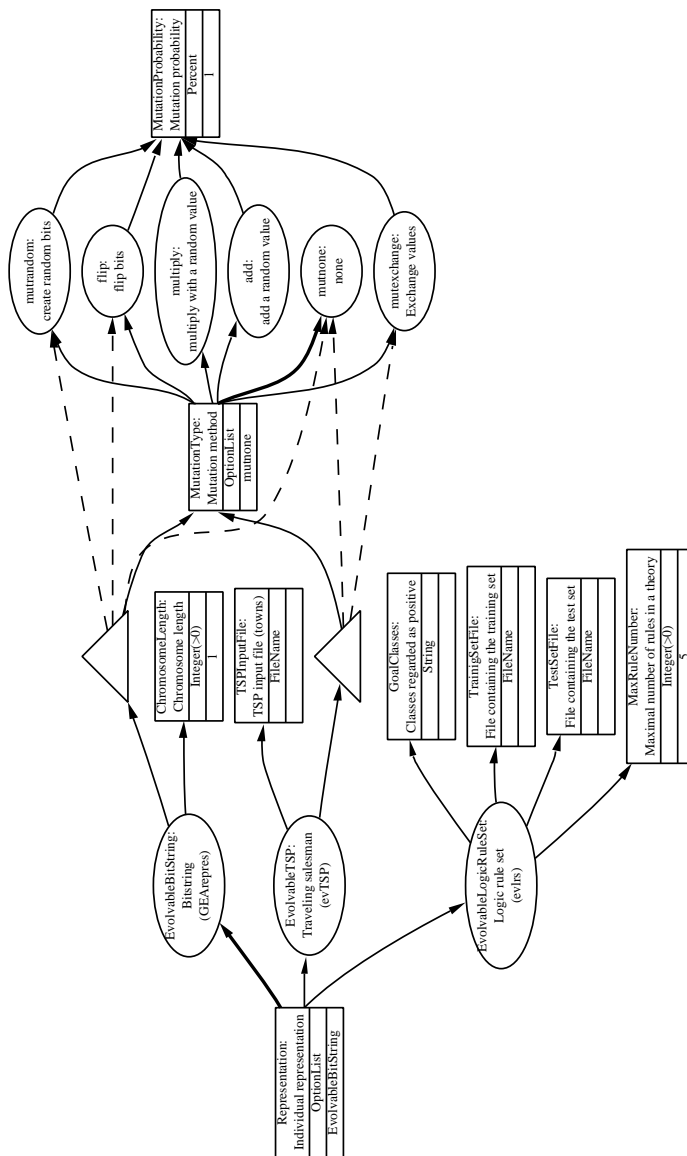


Figure 21: A detail of the GEA parameter structure



```

}

{
  Option: EvolvableTSP;
  Display: Traveling salesman;
  Parameters: TSPInputFile;
  Restrictions: MutationType(mutnone, mutexchange) |
    RecombinationType(recnone, recorder);
  Special: evTSP;
    # representation is implemented in libevTSP.so/evTSP.dll
}

{
  Option: mutnone;
  Display: none;
}

{
  Option: flip;
  Display: flip bits;
  Parameters: MutationProbability;
}

{
  Option: mutrandom;
  Display: create random bits;
  Parameters: MutationProbability;
}

{
  Option: add;
  Display: add a random value;
  Parameters: MutationRate, MutationProbability;
}

{
  Option: multiply;
  Display: multiply with a random value;
  Parameters: MutationRate, MutationProbability;
}

{
  Option: mutexchange;
  Display: Exchange values;
  Parameters: MutationProbability, MutationRate;
}

{
  Parameter: MaxRuleNumber;
  Display: Maximal number of rules in a theory;
  Type: Integer(>0);
  Group: ML;
  Default: 5;
}

{
  Parameter: TrainigSetFile;
  Display: File containing the training set;
  Type: FileName;
  Group: ML;
}

```

```

{
  Parameter: TestSetFile;
  Display: File containing the test set;
  Type: FileName;
  Group: ML;
}

{
  Parameter: GoalClasses;
  Display: Classes regarded as positive;
  Type: String;
  Group: ML;
}

{
  Parameter: TSPInputFile;
  Display: TSP input file (towns);
  Type: FileName;
  AllowEmpty: no;
  AllowNonExistent: no;
  Group: TSP;
}

```

Please refer to Subsection 2.3.1 for an explanation of the syntax of the *input file*. On the *graph*, boxes, ellipses and triangles represent parameters, possible parameter values (of parameters of type `OptionList`) and restrictions on the values of `OptionList` parameters, respectively. The upper part of a box contains the name and display name of the respective *parameter*. The middle part shows the type of the parameter (with the conditions of the correct parameter value, if there are any), while the bottom part shows the default value. As for the *options*, the ellipses are labeled with the names and the display names of the options, and the file name of the shared object file can be read in parentheses in the case of the individual representation forms.

For a parameter of type `OptionList`, arrows starting at the box of the parameter going to ellipses show its possible values. A thicker arrow indicates the default value. Arrows are going from an ellipse to the boxes of the *dependent parameters* of the option. Triangles are interpreted in the following way: the solid arrow going to the triangle starts from the restricting parameter value, while the one starting at the ellipse points to the restricted parameter. A dashed arrow going from a triangle to an ellipse indicates that the restricted parameter may have as value the option shown in the ellipse if the restricting parameter is set to the restricting parameter value. An example from the shown graph: if the representation of the individuals is chosen to be `EvolvableTSP` then the *type of the mutation* can be either *none* or *mutexchange*, the other possible values of the parameter `MutationType` are not allowed.

## B The TSP Problem

### The Special Functions

Besides the fitness function, two additional methods belong to the implementation of the TSP problem: an initializer and a terminator function. The initializer reads the description of the task

(i.e. the data of the towns) into a data structure for the fitness function, while the terminator frees the allocated memory area. Here follows a draft of this code.

```
int TownNo = 0;
float *XCoords = NULL;
float *YCoords = NULL;
float **Distances = NULL;

void evtsp_init_cpp(e_params *ep) // initializer
{
    const ep_paramdata *eppd;

    // determine the input file name
    eppd = e_params_get_parameter(ep, GEA_TSP_INPUTFILE);
    if (!eppd || !eppd->value) { /* Error message, return */ }
    const char *ifname = (char *)eppd->value;

    // load the town's coordinates
    // ...

    // compute distances
    // ...
}

void evtsp_done_cpp(e_params *ep) // terminator
{
    for (int tc = 0; tc < TownNo; tc++) delete[] Distances[tc];
    delete[] Distances;
    delete[] XCoords;
    delete[] YCoords;
}

double evtsp_fit_cpp(Evolvable *ev, int &result) // fitness
{
    double d = 0;
    EvolvableTSP *evtsp;

    // type correct?
    if (typeid(*ev) != typeid(EvolvableTSP)) {
        result = 1; return 0;
    }

    evtsp = (EvolvableTSP *)ev;

    // length correct?
    if (evtsp->GetLength() != TownNo) {
        result = 2; return 0;
    }

    for (int tc = 0; tc < TownNo-1; tc++)
        d += Distances[evtsp->GetValueAt(tc)][evtsp->GetValueAt(tc+1)];
    d += Distances[0][TownNo-1];

    return d;
}
```

## The Parameters of the Evolution

The parameter settings of the evolution process are stored in the file `evtsp.epv`, the contents of which are listed here (note that the names of the functions have been altered here, since the symbols generated by the C++ compiler have to be given as parameter values):

```
# GraphGEA project file, v0.3

TSPInputFile = sample.tsp
Representation = EvolvableTSP
OptimizationType = minimize
FitnessReduction = 70
SelectionMethod = ProportionalSelection
ElitismRate = 1
EA = GA
PopulationSize = 1000
InitialPopulation = random
ChangeUnderGeneration = 300
ChangeUnderPercent = 0.3
HaltingCondition1 = changeunder
HaltingCondRelation = or
HaltingCondition2 = break
SubPopulations = 1
RandomSeed = 0
ProbabilityAdaptation = no
MutationProbability = 15.2
MutationRate = 7
MutationType = mutexchange
RecombinationProbability = 86.2
RecombinationType = recorder
FitnessFunctionName = evtsp_fit_cpp__FP9EvolvableRi
FitnessPlugInName = libTSPfit.so
IsPreFunction = yes
PrePlugInName = libTSPfit.so
PreFunctionName = evtsp_init_cpp__FP8e_params
IsInFunction = no
IsPostFunction = yes
PostPlugInName = libTSPfit.so
PostFunctionName = evtsp_done_cpp__FP8e_params
EAPlugInDir = /home/inf2/i2staff/zntoth/lib/
SelectPlugInDir = /home/inf2/i2staff/zntoth/lib/
RepresPlugInDir = /home/inf2/i2staff/zntoth/lib/
PrintCPUtime = yes
PrintBMWFitness = yes
PrintFitnessVariance = yes
PrintGeneVariances = yes
DumpPopulationDuring = no
DumpPopulationAfter = no
PrintParams = no
PrintFitnesses = fitnever
WhenGeno = genochange
GenoBestN = 10
WhichGeno = genobestn
UseAllGenosPlugin = no
UseLHGenesPlugin = no
PrintGenotypes = genomainlog
WhenPheno = phenochange
PhenoBestN = 1
WhichPheno = phenobestn
UseAllPhenosPlugin = no
PrintPhenotypes = phenomainlog
```

```
MainLogFileName = evperm.ggl
MainLog = logboth
```

### The Log File

Finally, the beginning of the log file (evperm.ggl) of the run with the above parameters is inserted. Note that the phenotypes are not printed after the first generation, because the respective parameter is set to print phenotypes only if the fitness value of the best individual had changed since the previous generation (WhenPheno = phenochange). The phenotype of a solution for the traveling salesman problem is a series of drawing commands displaying the towns and the route. The drawing commands are described in Subsection 4.3.1.

```
GEA v1.2-2 started: Fri May  3 19:41:39 2002
```

```
Random seed: 1020447699
```

```
Generation: 0
```

```
Time: 0.08
```

```
Best fitness: 183.958
```

```
Mean fitness: 257.244
```

```
Worst fitness: 315.046
```

```
Fitness variance: 405.656
```

```
Gene variances: 32.5714 34.3055 32.9902 34.3339 31.3495 33.4094 32.7305 31.7346 33.7063 33.0637
```

```
Phenotypes:
```

```
PHEN00000 B -1.4 -0.9 27.4 21.9
A 5.025 4.525 0.95 0.95 0 360 F
A 10.025 1.025 0.95 0.95 0 360 F
A 13.525 8.025 0.95 0.95 0 360 F
A 19.525 5.525 0.95 0.95 0 360 F
A 17.525 0.525 0.95 0.95 0 360 F
A 24.025 7.025 0.95 0.95 0 360 F
A 18.525 16.525 0.95 0.95 0 360 F
A 9.025 12.025 0.95 0.95 0 360 F
A 16.025 10.025 0.95 0.95 0 360 F
A 6.525 7.525 0.95 0.95 0 360 F
A 3.525 13.525 0.95 0.95 0 360 F
A 10.525 15.525 0.95 0.95 0 360 F
A 11.025 19.525 0.95 0.95 0 360 F
A 17.025 19.025 0.95 0.95 0 360 F
A 3.025 18.525 0.95 0.95 0 360 F
A 8.025 5.525 0.95 0.95 0 360 F
A 0.525 8.025 0.95 0.95 0 360 F
A 19.525 10.525 0.95 0.95 0 360 F
A 23.525 19.525 0.95 0.95 0 360 F
A 24.525 13.525 0.95 0.95 0 360 F
L 7 8 1 8.5
L 1 8.5 14 8.5
L 14 8.5 3.5 19
L 3.5 19 4 14
L 4 14 25 14
L 25 14 19 17
L 19 17 11.5 20
L 11.5 20 17.5 19.5
L 17.5 19.5 11 16
L 11 16 9.5 12.5
L 9.5 12.5 20 11
L 20 11 24 20
```

L 24 20 16.5 10.5  
 L 16.5 10.5 5.5 5  
 L 5.5 5 8.5 6  
 L 8.5 6 10.5 1.5  
 L 10.5 1.5 18 1  
 L 18 1 20 6  
 L 20 6 24.5 7.5  
 L 24.5 7.5 7 8

EndPhenotypes

Genotypes:

0	9	16	2	14	10	19	6	12	13	11	7	17	18	8	0	15	1	4	3	5	183.958
1	12	14	0	11	8	2	15	9	1	5	17	19	3	7	4	6	18	13	10	16	194.954
2	14	13	12	6	18	19	8	9	0	15	1	2	7	10	3	16	11	5	4	17	196.398
3	4	8	17	11	18	6	0	9	15	3	2	12	5	19	13	14	10	7	16	1	197.827
4	5	17	6	12	1	4	18	19	7	11	9	8	3	13	2	15	10	14	16	0	199.251
5	19	6	13	1	3	5	8	2	7	9	10	15	17	18	12	11	4	14	16	0	199.551
6	4	14	12	13	3	5	17	7	11	10	15	2	8	18	6	1	16	9	0	19	201.228
7	16	15	12	13	9	1	4	19	5	7	8	3	17	6	18	2	11	14	0	10	201.844
8	18	13	8	17	2	5	7	10	16	1	0	4	3	14	12	6	19	15	11	9	203.268
9	18	11	13	19	5	16	12	10	14	17	4	8	3	1	0	15	9	2	6	7	205.526

EndGenotypes

Generation: 1

Time: 0.22

Best fitness: 183.958

Mean fitness: 253.835

Worst fitness: 319.749

Fitness variance: 398.254

Gene variances: 33.006 33.1043 31.762 33.3335 30.4134 34.2017 33.7558 32.589 34.1774 33.8579 35.6898

# Index

- adaptation of operator probability, 5
- building blocks, 53
- callback, 37
- child process, 42, 43, 45
- chromosome, 5
- client-server, 53
- comma strategy, 6, 14
- context-free, 24
- context-sensitive, 26
- convergence graphs, 53
- CPU time, 17
- crossover
  - multi-point, 7
  - parametrized uniform, 7
  - single-point, 7
- crossover points, 7
- dependent parameters, 20
- dump, 38
- e\_params, 34, 42
  - conditions, 19, 20, 62
  - correctness of the parameter values, 30
  - data types, 19
  - default value, 20, 62
  - definition of a parameters structure, 24
  - dependent parameters, 28, 30, 62
  - graph, 59
  - graphical extension, 28
  - parameter list, 28
  - parameter value list, 28
  - restrictions, 20, 62
- e\_params data structure, 19
- EA Visualizer, 53
  - GraphDrawer, 53
- EBNF, 24
- elitism, 8
- encapsulated postscript, 46
- evaluation, 5
- event handler, 43
- evolution process, 37
  - reconstruction, 38, 43, 45
- evolution strategies, 6, 14, 33
  - comma, 6, 14
  - plus, 6, 14
- evolutionary algorithms, 5
- Evolvica, 53
- EvolVision, 53
- fitness, 5
  - modified, 10
- fitness function, 37, 62
- GEA, 33
  - class hierarchy, 33
    - EA, 37
    - ESNextGen, 36
    - Evolvable, 33
    - EvolvableBiString, 35
    - EvolvableIntVector, 35
    - EvolvablePermutation, 35, 39
    - EvolvableRealVector, 35
    - GANextGen, 36
    - NextGenMethod, 33, 36
    - PlugIns, 37
    - Population, 35
    - SelectionMethod, 33, 36
- GeatBox, 54
- GEmul, 43
- gene, 5
- generation, 5
- genetic algorithms, 6
- genetic operators, 5
  - crossover, 7, 13
    - order crossover, 35
  - mutation, 5, 7, 12, 14
    - exchange mutation, 35
  - probability, 5
    - adaptation, 5, 16
  - recombination, 5, 7, 14
    - discrete, 14
    - global, 14
    - intermediate, 14
    - local, 15
  - reproduction, 5
- genetic programming, 6, 12, 33
- genotype, 5, 34
- GIGA, 54
- GIMP ToolKit, 19, 28, 41
- GLib, 19
- gnuplot, 46
- Gonzo, 54
- GraphGEA, 41
- GREDEA, 33
- helper functions, 45
- hill climbing, 14
- individual, 5
- initial population, 5, 38
- interactive evolution, 11
- interactive execution, 17, 42, 54
- Lindenmayer systems, 33
- log file, 38, 43, 65
- logging parameters, 45, 50
- Mathematica notebooks, 53
- Matlab, 54
- meta-ES, 16, 34, 36
- multiple runs, 53
- mutation, 7
- mutation rate, 14
- new generation
  - ES, 15

- GA, 7
- OSF/Motif, 54
- parameter description file, 59
- parameter settings, 64
- permutation, 39
- phenotype, 5, 34
- pipe, 43, 45
- plug-in, 34, 36, 37, 44, 54
- plus strategy, 6, 14
- population, 5, 35
  - initial, 5
- population mean, 10
- population standard deviation, 10
- probability, 5
  - adaptation, 5
- programming language, environment
  - C, 19, 41
  - C++, 33
  - Java, 53, 54
  - LISP, 54
  - Matlab, 54
- random search, 14
- random seed, 43, 45
- recombination, 7
- recombination points, 13
- representation forms, 5, 35
  - bit-string, 5, 6, 35, 53
  - branching structures, 6
  - integer vector, 35
  - permutation, 35
  - real vector, 5, 14, 35
  - tree-like, 12
- roulette wheel, 9–11
- selection, 5, 8
- selection methods, 8, 36
  - best, 10
  - Boltzmann, 11
  - fitness proportional, 9
    - fitness reduction, 9
  - interactive, 11
  - random, 10
  - rank based, 9
  - Sigma scaling, 10
  - tournament, 10
- selection pool, 14, 15
- selection pressure, 8, 10, 11
- shared memory, 38, 43
- signal handler, 43
- special functions
  - intermediary, 37
  - posterior, 37, 62
  - preliminary, 37, 39, 62
- steady-state GA, 17
- temperature, 11
- termination, 6
- traveling salesman problem, 39, 53, 62
- Unix System V Interprocess Communication, 38
- visualization, 17, 44
  - color coding, 17, 47
  - course, 17, 54
  - drawing commands, 49, 65
  - drawings, 17, 48
  - fitness values, 17
  - genotypes, 17
  - internals, 54
  - lowest and highest gene values, 17
  - off-line, 43, 53, 54
  - phenotypes, 17, 48
  - plots, 17, 45
  - plug-in, 50
  - search behavior, 54
  - status, 17, 54



## References

- [1] M. Abramson and L. Hunter. Classification using cultural co-evolution and genetic programming. In J. R. Koza, D. E. Goldberg, D. B. Fogel, and R. L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 249–254, Stanford University, CA, USA, 28–31 July 1996. MIT Press.
- [2] K. Aoki, H. Takagi, and N. Fujimura. Interactive ga-based design support system for lighting design in computer graphics. In *International Conference on Soft Computing (IIZUKA '96)*, pages 533–536, Iizuka, Fukuoka, Japan, 1996. World Scientific.
- [3] 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*, Orlando, Florida, USA, 13-17 July 1999. Morgan Kaufmann.
- [4] P. A. N. Bosman. EA visualizer.  
<http://www.cs.ruu.nl/people/peterb/computer/ea/eavisualizer/EAVisualizer.htm>.
- [5] C. Caldwell and V. S. Johnston. Tracking a criminal suspect through face-space with a genetic algorithm. In *ICGA91*, pages 416–421, 1991.
- [6] T. D. Collins. *The Application of Software Visualization Technology to Evolutionary Computation: A Case Study in Genetic Algorithms*. Ph.d research, Knowledge Media Institute, The Open University, Milton Keynes, UK, 1998.
- [7] T. Dabs. Eine Entwicklungsumgebung zum Monitoring Genetischer Algorithmen. Master's thesis, University of Würzburg, 1994.
- [8] C. Darwin. *On the Origin of Species*. Murray, London, 1859.
- [9] L. Davis. Adapting operator probabilities in genetic algorithms. In *Proceedings of the Third ICGA*, pages 61–67. Morgan Kaufmann, 1989.
- [10] K. A. De Jong. An analysis of the behavior of a class of genetic adaptive systems. Ph.d thesis, University of Michigan, 1975.
- [11] S. Droste. Efficient genetic programming for finding good generalizing boolean functions. In J. R. Koza, K. Deb, M. Dorigo, D. B. Fogel, M. Garzon, H. Iba, and R. L. Riolo, editors, *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pages 82–87, Stanford University, CA, USA, 13-16 July 1997. Morgan Kaufmann.
- [12] T. Fühner and C. Jacob. Evolvision - an evolvisca visualization tool. In L. Spector, E. D. Goodman, A. Wu, W. B. Langdon, Hans Michael Voigt, M. Gen, S. Sen, M. Dorigo, S. Pezeshk, M. H. Garzon, and E. Burke, editors, *Proceedings of the Genetic and Evolutionary Computation Conference*, page 176, San Francisco, California, USA, 7-11 July 2001. Morgan Kaufmann.

- [13] D. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley Reading, MA, 1989.
- [14] J. Graph and W. Banzhaf. Interactive evolution of images. In *International Conference on Evolutionary Programming*, 1995.
- [15] J. H. Holland. *Adaption of Natural and Artificial Systems*. University of Michigan Press, Ann Arbor, Michigan, 1975.
- [16] C. Jacob. *Principia Evolvica – Simulierte Evolution mit Mathematica*. Dpunkt Verlag, 1997.
- [17] C. Jacob. *Principia Evolvica – Simulierte Evolution mit Mathematica*, page 443. In [16], 1997.
- [18] Johnson and McGeoch. The traveling salesman problem: A case study. In E. Aarts and J. K. Lenstra, editors, *Local Search in Combinatorial Optimization*, Wiley, 1997. 1997.
- [19] G. Kókai, Z. Tóth, and R. Ványi. Application of genetic algorithms with more populations for Lindenmayer systems. In *Proceedings of the International Symposium on Engineering of Intelligent Systems, EIS'98*, pages 324–331. ICSC Academic Press, 1998.
- [20] G. Kókai, Z. Tóth, and R. Ványi. Evolving artificial trees described by parametric L-systems. In *Proceedings of the First Canadian Workshop on Soft Computing*, pages 1722–1728, Edmonton, Alberta, Canada, 9 # may 1999.
- [21] G. Kókai, R. Ványi, and Z. Tóth. Parametric L-system description of the retina with combined evolutionary operators. In Banzhaf et al. [3], pages 1588–1595.
- [22] J. R. Koza. Evolving a computer program to generate random numbers using the genetic programming paradigm. In R. K. Belew and L. B. Booker, editors, *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 37–44, University of California - San Diego, La Jolla, CA, USA, 13-16 July 1991. Morgan Kaufmann.
- [23] J. R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, Massachusetts, 1992.
- [24] J. R. Koza. Automated discovery of detectors and iteration-performing calculations to recognize patterns in protein sequences using genetic programming. In *Proceedings of the Conference on Computer Vision and Pattern Recognition*, pages 684–689. IEEE Computer Society Press, 1994.
- [25] D. Levine, M. Facello, and P. Hallstrom. Stalk: An interactive system for virtual molecular docking. *IEEE Science and Engineering*, 2/97:55–67, 1997.
- [26] Z. Michalewicz. *Genetic Algorithms + Data Structures = Evolution Programs*. Artificial Intelligence. Springer-Verlag, 1992.

- [27] J. F. Miller. An empirical study of the efficiency of learning boolean functions using a cartesian genetic programming approach. In Banzhaf et al. [3], pages 1135–1142.
- [28] M. Mitchell. *An Introduction to Genetic Algorithms*. MIT Press, Cambridge Massachusetts, 1996.
- [29] U.-M. O'Reilly and G. Ramachandran. A preliminary investigation of evolution as a form design strategy. In C. Adami, R. K. Belew, H. Kitano, and C. E. Taylor, editors, *Proceedings of the Sixth International Conference on Artificial Life*, University of California, Los Angeles, 26-29 June 1998. MIT Press. Forthcoming.
- [30] V. P. Plagianakos and M. N. Vrahatis. Training neural networks with 3-bit integer weights. In Banzhaf et al. [3], pages 910–915.
- [31] H. Pohlheim. Visualization of evolutionary algorithms – set of standard techniques and multidimensional visualization. In Banzhaf et al. [3], pages 533–540.
- [32] I. Rechenberg. *Evolutionsstrategien: Optimierung Technischer Systeme nach Prinzipien der Biologischen Evolution*. Fromman-Holzboog, Stuttgart, 1973.
- [33] H.-P. Schwefel. Numerische optimierung von computer-modellen mittels der evolutionsstrategie. *Interdisciplinary Systems research (26)*, Birkh.user, Basel, 1977.
- [34] J. Sherrah. *Automatic Feature Extraction for Pattern Recognition*. PhD thesis, University of Adelaide, South Australia, July 1998.
- [35] J. R. Smith. Designing biomorphs with an interactive genetic algorithm. In *ICGA91*, pages 535–538, 1991.
- [36] Z. Tóth. *The Generic Evolutionary Algorithms Programming Library*. Master's thesis, University of Szeged, Szeged, Hungary, 2000.
- [37] Z. Tóth and G. Kókai. An evolutionary optimum searching tool. In *The Proceedings of the Fourteenth International Conference on Industrial & Engineering Applications of Artificial Intelligence & Expert Systems (IEA/AIE-2001)*, volume 2070 of *LNAI*, pages 19–24, Budapest, Hungary, June 4–7 2001. Springer-Verlag.
- [38] Z. Tóth, G. Kókai, and R. Ványi. Interactive visual tree evolution. In *EIS2000 Second International ICSC Symposium on Engineering of Intelligent Systems, June 27 - 30, 2000 at the University of Paisley, Scotland, U.K.*, pages 384–390, 2000.
- [39] R. Ványi. *Modelling the Evolution of the Flora*. Bachelor's thesis (in Hungarian), József Attila University, Szeged, Hungary, 1998.

- 
- [40] R. Ványi, G. Kókai, Z. Tóth, and T. Pető. Grammatical retina description with enhanced methods. In R. Poli, W. Banzhaf, W. B. Langdon, J. F. Miller, P. Nordin, and T. C. Fogarty, editors, *Genetic Programming, Proceedings of EuroGP'2000*, volume 1802 of *LNCS*, pages 193–208, Edinburgh, Apr. 15–16 2000. Springer-Verlag.
  - [41] G. Venturini, M. Slimane, F. Morin, and J. P. A. de Beauville. On using interactive genetic algorithms for knowledge discovery in databases. In *ICGA97*, pages 696–703, 1997.
  - [42] N. Wirth. What can we do about the unnecessary diversity of notation for syntactic definitions. *Communications of the ACM*, 20(11):822–823, Nov. 1977.