Acta Cybernetica 16 (2003) 337-365.

A Graphical User Interface for Evolutionary Algorithms*

Zoltán Tóth †

Abstract

The purpose of *Generic Evolutionary Algorithms Programming Library (GEA[1])* system is to provide researchers with an easy-to-use, widely applicable and extendable programming library which solves real-world optimization problems by means of evolutionary algorithms. It contains algorithms for various evolutionary methods, implemented genetic operators for the most common representation forms for individuals, various selection methods, and examples on how to use and expand the library. All these functions assure that *GEA* can be effectively applied on many problems. *GraphGEA* is a graphical user interface to *GEA* written with the GTK API. The numerous parameters of the evolutionary algorithm can be set in appropriate dialog boxes. The program also checks the correctness of the parameters and saving/restoring of parameter sets is also possible. The selected evolutionary algorithm can be executed interactively on the specified optimization problem through the graphical user interface of *GraphGEA*, and the results and behavior of the EA can be observed on several selected graphs and drawings. While the main purpose of *GEA* is solving optimization problems, that of *GraphGEA* is education and analysis. It can be of great help for students understanding the characteristics of evolutionary algorithms and researchers of the area can use it to analyze an EA's behavior on particular problems.

1 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

*This work was supported by the grants of the German Academic Exchange Service (DAAD)
†Institute of Informatics, University of Szeged, Árpád tér 2, H-6720 Szeged, Hungary. Now visiting Department of Computer Science 2: Programming Systems, Friedrich-Alexander University of Erlangen-Nuremberg, Martensstr. 3, D-91058 Erlangen, Germany.
e-mail: `zntoth@inf.u-szeged.hu`
[1]The project's home page can be found at `http://gea.ztoth.net`

the genotypes by *genetic operators* observed in nature: generally *mutation* and *recombination* [15, 22, 32].
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 in 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 paper is to present a tool for visualizing evolutionary algorithms: the *GraphGEA* program. *GraphGEA* is a graphical user interface (GUI) to the *Generic Evolutionary Algorithms Programming Library* (*GEA*) [36]. *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 2 offers a short overview of evolutionary algorithms. The presented systems use a special data structure to hold the parameters of the evolution process, this data structure is presented briefly in Section 3. Section 4 deals with the *GEA* system: the class hierarchy, the functioning of the various evolutionary algorithms, the selection methods and the genetic operators are described. Section 5 presents the *GraphGEA* program with a detailed description of the user

interface and the visualization tools. In Section 6, some references to and comparisons with the related work can be found. Finally in Section 7 a summary of the work is given.

2 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) 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 on, 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 the *fitness value*, a specified extremum of which has to be found by the evolutionary algorithm. This fitness value is considered when performing *selection*.
The creation of new individuals is implemented 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].
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 one can be used with random modifications. Then the individuals are evaluated and new generations are created until a termination condition

is satisfied, which, in the simplest cases, is reaching a certain generation number or the stagnation of the best individual's fitness value. The generation of the new population is absolutely algorithm-dependent, so these methods will be discussed at the specification of the algorithms.

Several kinds of evolutionary algorithms are known, the most important ones of which are *genetic algorithms* (GAs) [10, 15] and *evolution strategies* (ESs) [31]. 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: it is bit-string for GAs and real vector for ESs. The two kinds of EAs also differ in the way genetic operators are applied.

There is a special kind of genetic algorithms, namely *genetic programming* (GP), introduced by John R. Koza [22]. 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.

In the following, the characteristics of genetic algorithms, genetic programming and evolution strategies are presented in brief. At the end of the section, the possibilities of the visualization are discussed.

2.1 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 multidimensional spaces in order to solve, or solve approximately, a wide variety of problems [13, 25]. Here follows the description of the two most important genetic operators for GAs: *mutation* and *crossover*.

Mutation randomly changes each bit of an individual with a certain probability. 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 independently whether to change it or not.

In the case of GAs, the *recombination* operator always takes two parents and creates two descendants, thus it is usually called *crossover*. The main kinds of crossover are *point-based crossover* and *parametrized uniform crossover*. For *point-based crossover*, the crossover points (whose number is given) are chosen at random. The case when there is only one crossover point is called *single-point crossover*. After choosing the crossover points, the parts of the individuals between these points are exchanged. *Parametrized uniform crossover* exchanges each bit of the parent individuals with a given probability to create the descendants.

The process of *creating the new generation* for a GA is quite simple: first, a new empty population is created. Then, to ensure the monotonity of the process, a number of best individuals in the previous generation is copied into the new pop-

ulation 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.

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. Since there are a wide range of functions that can be optimized with genetic algorithms and these functions behave very differently, various selection methods have been developed to deal with them [27]. 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. 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 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 selection probability of an individual is usually proportional to its fitness value or rank in the population. Other constructs use only a subset of the population when selecting or apply more complicated transformation functions to the fitness values. Interactive selection is usually used when it is impossible to formalize an effective fitness function (e.g. in some design and shape recognition applications [2, 5, 14, 24, 34, 40]); here, the individuals are presented to the user and he/she can decide which of them are the most suitable solutions.

2.2 Genetic Programming

It is difficult and restrictive to represent hierarchies of dynamically varying size and shape with fixed length vectors. *Genetic programming* (GP) 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 [22].

Genetic programming has been successfully applied to problems such as classification [1] and pattern recognition [23, 33], generation of maximal entropy sequences of random numbers [21], Boolean function learning [11, 26], simultaneous architectural design [28] and training of neural networks [29].

GP's *genetic operators* work with sub-trees of the individuals. *Mutation* chooses a node of the tree and replaces the corresponding sub-tree with a new, randomly generated one, while *crossover* creates the offspring by exchanging randomly selected sub-trees of the parent individuals.

2.3 Evolution Strategies

Evolution strategies (ESs) are less popular than genetic algorithms, although 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 μ, ρ and λ 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 μ 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 λ individuals. Obviously, $\mu \leq \lambda$ must hold if the comma strategy is applied. There are special cases for ES, e.g. when ρ 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.

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. 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 controled by the mutation rate parameter.

ES recombination takes ρ individuals as parents 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 exist *discrete/intermediate* and *local/global* recombination methods; their detailed description with examples can be found in Section 5.3 of [16].

The algorithm for *creating the new generation* for an ES is the following: First λ individuals are created in the empty selection pool. To create a new individual, ρ 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 μ individuals of the selection pool.

In nature, it can be observed that populations of the same species are sometimes evolving separarately, 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 carried

out by randomly replacing some individuals in the population, and recombination can work as crossover works in GAs. The similar approach in genetic algorithms is called *island model.*

2.4 Possibilities of the 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 state* 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 the 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.

3 The e_params Data Structure

This section gives a short 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*[2] *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*[2]*)*, 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.
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 iff 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.
The possible *data types* of the parameters include strings, file names, integer and real numbers and boolean values. A type called `OptionList` has been introduced for parameters which can have their values from a predefined finite set (`Options`).

[2] http://www.gtk.org

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 can also be defined for more sophisticated functionality.
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 the 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.
The definition of a parameters data structure is stored in a plain text file with the suffix "`.ep`" the format of which is given formally by a grammar in *Extended Backus-Naur Form* (EBNF, [41]) and context-sensitive restrictions. 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. Lines beginning with a hash mark are regarded as comments.
A *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*[2] 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 form of the parameter setting dialog box can be seen in Figure 1. Each row shown in the table corresponds to one parameter. The second column of the table shows the parameter's "display name" (which can be different from the name used for the internal representation), 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 `OptionList` 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. When it is pressed, a new window appears offering modifications to the dependent parameters.
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 and the new value has not been checked yet. A *green* hash mark indicates a correct value of the parameter and a *red* one

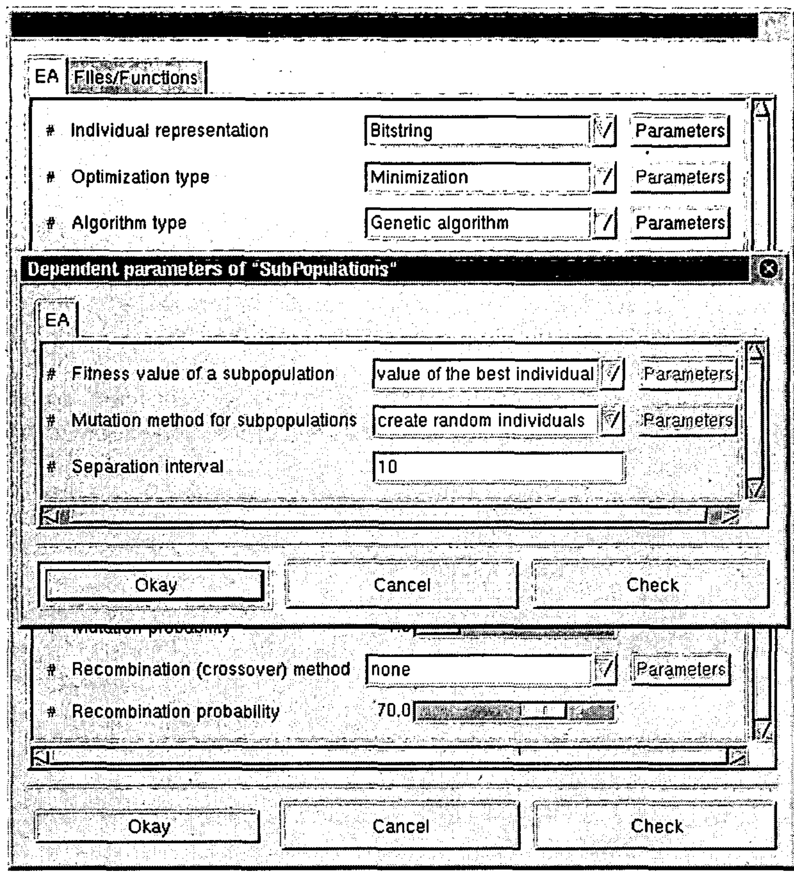

Figure 1: The parameter settings dialog boxes of the e_params data structure. The figure shows the setting of the dependent parameters

indicates that the value is incorrect. The *Check* button can be pressed to perform a test of the values of the parameters. A check is performed automatically when the window is first displayed and when the *Okay* button is pressed. Parameters with incorrect values cannot be saved.

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

4 The GEA System

Kókai, Ványi and Tóth have been involved in evolving fractal images since 1997. The first attempt was to reproduce and improve Koza's results with *Lindenmayer systems* (*L-systems*) [18, 22]. This project was written in *Java* and did not use any general genetic programming libraries. Then it was realized that *L-systems* are capable of describing plants and these plants can be evolved by interactive evolution (the *TEvol* program, [19, 37, 38]). 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 [20, 39]. These two projects required the evolution of the rewriting rules of the *L-systems* as well as their parameters. The most suitable algorithm for the evolution of the rewriting rules is genetic programming, while the one 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 both GPs and ESs could not be found at that time (in 1998), the design and implementation of a suitable system had begun. This system was later named *GEA* (Generic Evolutionary Algorithms Programming Library).

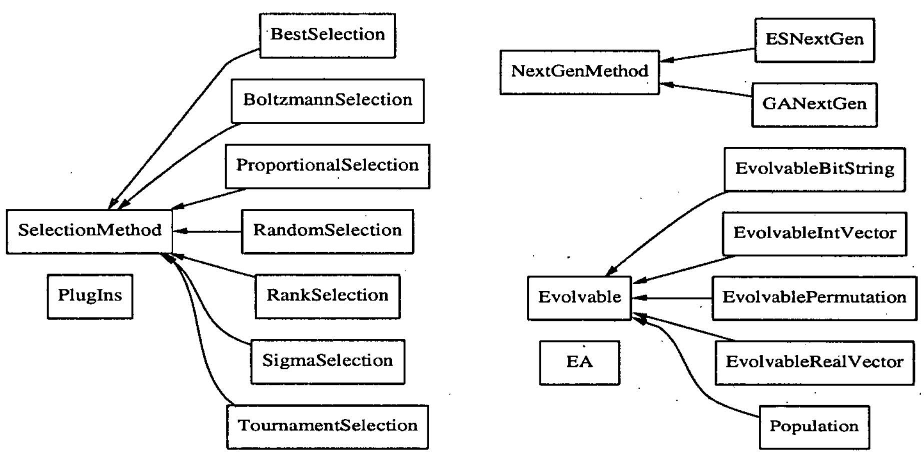

Figure 2: The class hierarchy of the *GEA* system

The class hierarchy of the current version of *GEA* can be seen in Figure 2. 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 3), it will find and load them if necessary.
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 at the same time), which makes having a transparent interface to them reasonable.
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*, *Crosswith* and *Recombine*. Input/output and factory functions provide an interface for the transportation of the evolvable objects.
GEA has currently four *built-in representation forms*, namely for bit-strings, real vectors, integer vectors and permutations. The class that represents a *population* is also an evolvable class (that is, genetic operators can be applied on it), this makes experiments with meta-ES in *GEA* possible.
The abstract class SelectionMethod is the superclass of all implemented *selection methods* in *GEA*.
As it is explained in Section 2, evolutionary algorithms mostly differ in the way the individuals are represented and new generations are created. Various representation forms are available via the Evolvable abstract class 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 plug-ins and the required class is loaded at running time.
Currently, two *evolutionary algorithm frameworks* are available in the *GEA* system: GANextGen and ESNextgen implement genetic algorithms and evolution strategies as they are described in Subsections 2.1 and 2.3, respectively.
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.
A common *plug-in handling interface* is provided to all classes which use shared libraries by class PlugIns. A static data member is used to keep account of the loaded shared libraries and a function can be used to look up a given symbol in a given shared library; the function loads the object file if needed.
The most important *problem-dependent function* in all evolution 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 evolution 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 has 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.

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.

path to parameter structure The name of the directory that contains the description of the parameter data structure of GEA.

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. The genotype dumps can be used as *milestones* to start an evolution process later with a given initial population.
For more information on the *GEA* system, see [35] and [36]. Usage examples can be found on the GEA home page.

5 GraphGEA

This section describes the *GraphGEA* program in detail. The system uses the graphical object set of the *GIMP ToolKit* (GTK[2]) which is written in the C programming language, thus this same language was used to implement *GraphGEA*. The functionality is presented beginning with the parameter settings, through the execution of the evolution process and closed with the visualization possibilities.

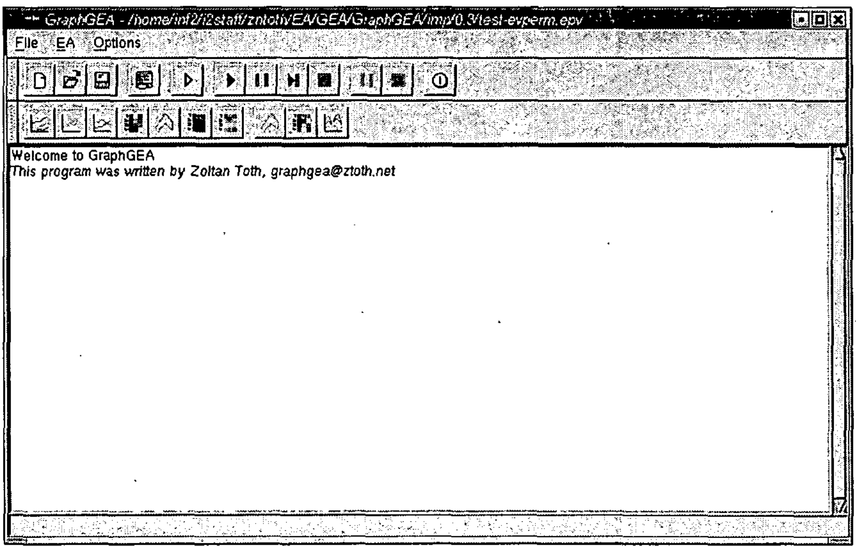

Figure 3: The main window of *GraphGEA*

The central window of the software is depicted in Figure 3. Below the menu bar, the main window of *GraphGEA* 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 parameter values are invalid. 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.

5.1 Managing the Parameters

When the *GraphGEA* program starts, it loads the parameter structure definition and main parameter list of *GEA*. 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 Section 3) 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.

5.2 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 has 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 in Figure 4.

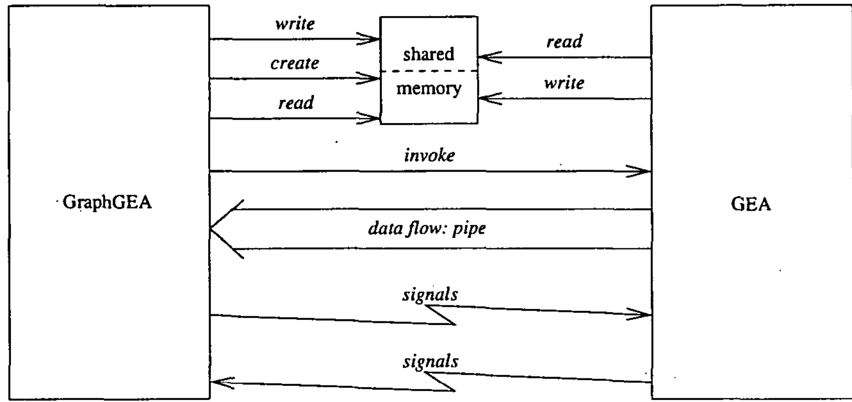

Figure 4: The interaction between *GraphGEA* and *GEA*

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 have not been stored since the last changes, it asks the user if they should be written to disk before starting *GEA* or not.
Starting the *GEA* program takes the following steps: first, a shared memory area is requested from the operating system, the messages between 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 Section 4) using the *execvp* function. The parent process opens a pipe to the child and registers an event handler to manage its output (by default, *GEA* writes its log onto the 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 has 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 is 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.

5.3 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 doesn'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 make the call of only three simple library functions necessary.
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 four functions (*create*, *init*, *new_data*, and *done*) in each loaded plug-in for the communication.
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 to 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 with short descriptions are listed in Table 1.

5.3.1 Methods of Visualization

As it is described in Subsection 2.4, there are basically three different visualization methods discussed in this paper: 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 some functions differs, too. Next, the look of the three plug-in window types and their functionality are discussed.
A *plot window* of *GraphGEA* is depicted in Figure 5. 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 depicted with different colors and the legend is displayed 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 visible diagrams. The boundaries can be set manually in the first toolbar by unchecking the appropriate checkboxes and entering the values into the input lines next to them. 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 in Figure 5 is

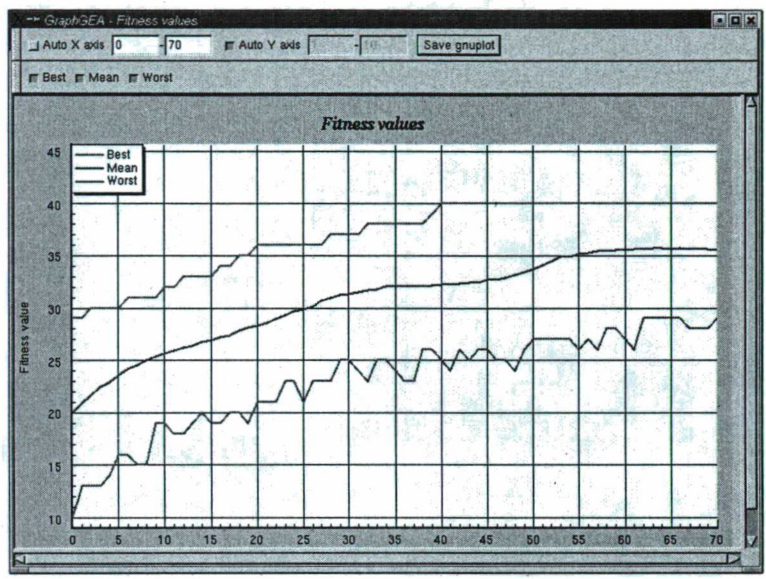

Figure 5: A plot window of *GraphGEA*

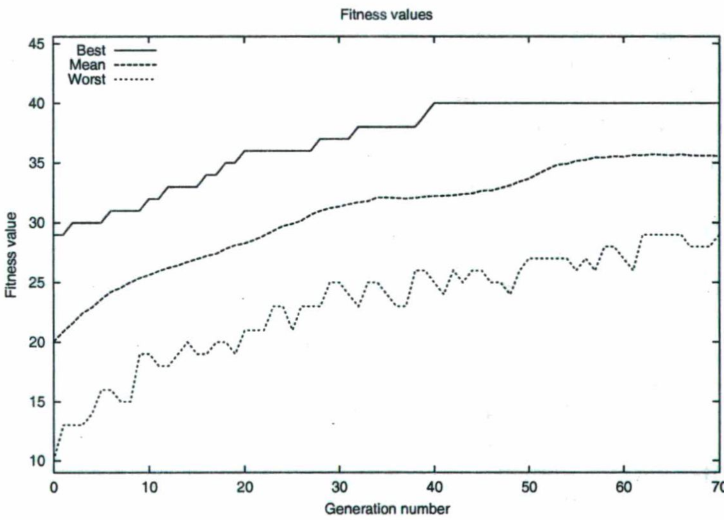

Figure 6: The gnuplot output of the plot window

displayed in Figure 6.

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 7. Arbitrary numerical values can be shown in the form of a two-dimensional array with additional explanatory

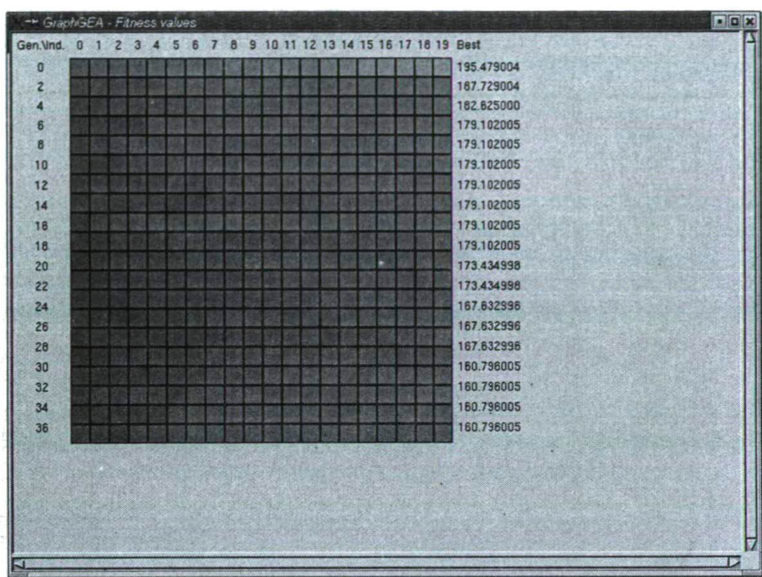

Figure 7: A color coding window of *GraphGEA*

columns on the left and the right hand side of the color matrix. In the current implementation, the lowest and highest displayed values can be specified directly or the plug-ins can compute them automatically. The specification or the automatic computation can be done either separately for each column or all columns can share the same limits.

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 8. 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 initializer function creates and displays the appropriate drawing object by looking at the representation form of the individuals: if the representation is known as a drawable one (that is, its phenotype is printed as a series of drawing commands) then a drawing area is created, otherwise a text field will appear.

The set of *drawing commands* of *GraphGEA* is the following:

B x1 y1 x2 y2 This command determines the boundaries of the drawing area. The individual is drawn in a way that the graphical primitives within the boundaries are always visible in the plug-in window.

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

L x1 y1 x2 y2 Draws a line from $(x1, y1)$ to $(x2, y2)$.

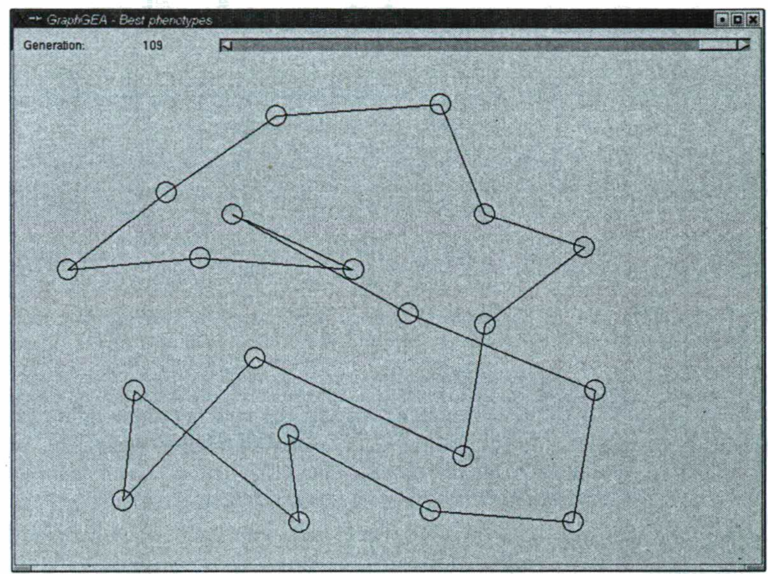

Figure 8: The phenotype of an individual drawn by *GraphGEA*

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 a1 a2 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 $a1$, the length by $a2$ (that is, $a2$ is the ending angle relative to $a1$). 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 x1 y1 x2 y2 ... xn yn 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.

5.3.2 The Implemented Plug-ins

Table 1 shows the list of the currently available visualization plug-ins of *GraphGEA*. Besides the name, icon of the show/hide button and type of the visualization tools, a short description is also given.

Name	Type	Description
Fitness values	plot, course	Diagrams of the best, mean and worst fitness values plotted against the generation number.
CPU times	plot, course	A diagram showing the used CPU time of the evolution process plotted against the generation number.
Fitness variance	plot, course	A diagram showing the variance of the fitness values in the population against the generation number.
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.
Best phenotypes	draw, course	The phenotypes of the best individuals of the generations. The individual can be selected by the generation number.
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.
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.
All phenotypes	draw, status	Offers all phenotypes of the current generation for viewing. The individual can be selected by its position in the population.
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.
Current gene values	plot, course	The lowest, average and highest values of each gene are plotted against the gene number.

Table 1: The currently available visualization plug-ins of the *GraphGEA* system

6 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 primary 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.

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 are 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 9). 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 8). 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 modules; 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 only realizes 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 Graphical user Interface for Genetic Algorithms. 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 10). 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.

GeatBx [30] is another very promising visualization tool with various plots and

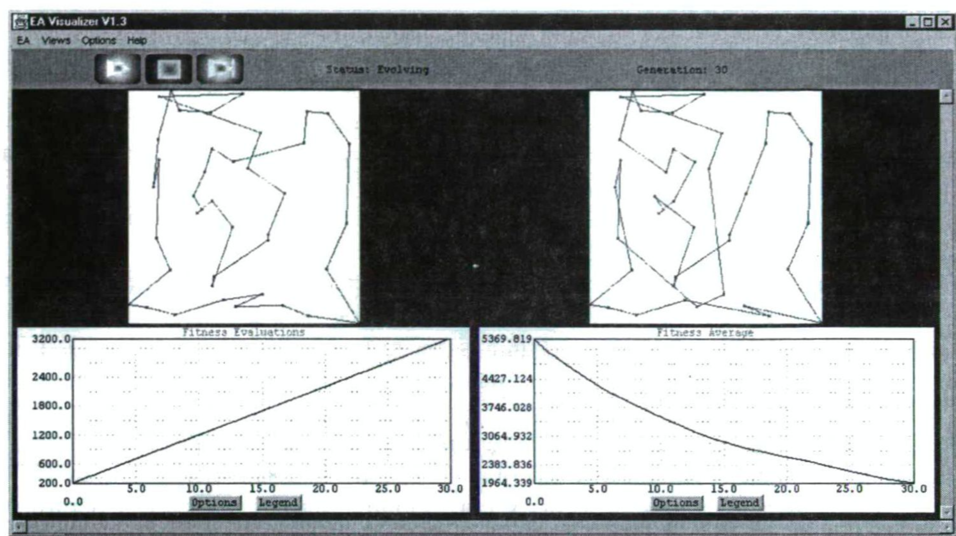

Figure 9: The *EA Visualizer* working on a TSP

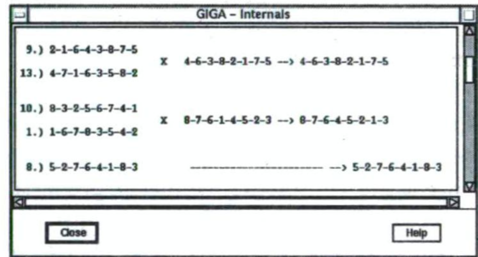

Figure 10: The 'internals' window of *GIGA*

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 *GeatBx* efficiently. Besides, *Matlab* is a commercial software. The tool 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

Name of the system	Supported algorithms	Language of implementation	Type of visualizations	Extension possible?	Interactive?	Off-line visualization?
GraphGEA	EAs	C/C++	Genotype, phenotype	yes	yes	yes
EA Visualizer	no GP	Java	Genotype, phenotype with restrictions	yes	yes	no
EvolVision	EAs	Mathematica, Java	Genotype, phenotype	yes	no	yes
GIGA	GAs	C, OSF/Motif	Genotype, phenotype	yes	partly	yes
GeatBx	no GP	MatLab	Genotype	no	yes	yes
Gonzo	GAs	LISP	Genotype	yes	yes	yes

Table 2: Comparison of the various visualization tools

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.
The advantages and disadvantages of the described systems are summarized in Table 2.

7 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 3. The *GEA* system is described in Section 4, while Section 5 deals with the *GraphGEA* system itself. Finally, a view on related work is given in Section 6.
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 divided 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.

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 thesis, 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 evolvica 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] 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.

[19] 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.

[20] 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.

[21] 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.

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

[23] 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.

[24] D. Levine, M. Facelllo, and P. Hallstrom. Stalk: An interactive system for virtual molecular docking. *IEEE Science and Engineering*, 2/97:55–67, 1997.

[25] Z. Michalewicz. *Genetic Algorithms + Data Structures = Evolution Programs.* Artificial Intelligence. Springer-Verlag, 1992.

[26] 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.

[27] M. Mitchell. *An Introduction to Genetic Algorithms.* MIT Press, Cambridge Massachusetts, 1996.

[28] 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, Cambridge.

[29] V. P. Plagianakos and M. N. Vrahatis. Training neural networks with 3-bit integer weights. In Banzhaf et al. [3], pages 910–915.

[30] H. Pohlheim. Visualization of evolutionary algorithms – set of standard techniques and multidimensional visualization. In Banzhaf et al. [3], pages 533–540.

[31] I. Rechenberg. *Evolutionsstrategien: Optimierung Technischer Systeme nach Prinzipien der Biologischen Evolution.* Fromman-Holzboog, Stuttgart, 1973.

[32] H.-P. Schwefel. Numerische Optimierung von Computer-Modellen mittels der Evolutonsstrategie. *Interdisciplinary Systems research (26), Birkh.user, Basel,* 1977.

[33] J. Sherrah. *Automatic Feature Extraction for Pattern Recognition.* PhD thesis, University of Adelaide, South Australia, July 1998.

[34] J. R. Smith. Designing biomorphs with an interactive genetic algorithm. In *ICGA91*, pages 535–538, 1991.

[35] Z. Tóth. *The Generic Evolutionary Algorithms Programming Library.* Master's thesis, University of Szeged, Szeged, Hungary, 2000.

[36] 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.

[37] 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.

[38] R. Ványi. *Modelling the Evolution of the Flora.* Bachelor's thesis (in Hungarian), József Attila University, Szeged, Hungary, 1998.

[39] 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.

[40] 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.

[41] N. Wirth. What can we do about the unnecessary diversity of notation for syntatic definitions. *Communications of the ACM*, 20(11):822–823, Nov. 1977.