# TEA - A C++ Library for the Design of Evolutionary Algorithms

Michael Emmerich Rafael Hosenberg

University of Dortmund
Dept. Computer Science
Systems Analysis Group
44227 Dortmund, Germany
{emmerich,hosenberg}@ls11.cs.uni-dortmund.de

**Abstract** A library for the design of standard and non-standard EAs in C++ is described. The simple object-oriented design of the TEA library allows the fast configuration of new non-standard evolutionary algorithms.
In TEA representation independent algorithms can be combined with non-standard genotypes. Complex genotypes can be build from existing simple genotypes. Furthermore, non-panmictic parallel population structures like neighbourhood and multipopulation EAs are supported. This paper introduces the main concepts of the TEA library. Examples illustrate how to build standard algorithms and how to design new kinds of algorithms and representations with TEA.

## 1 Introduction

Evolutionary Algorithms (EAs) utilize paradigms of biological evolution, like recombination, mutation and selection mainly for the solution of global optimization tasks. As a flexible and robust search technique, they have been successfully applied to various real world problems [1].

Some main classes of EAs are evolution strategies (ESs) [7], genetic algorithms (GAs) [1] and evolutionary programming (EP) [1]. Today, EAs differ mainly in their search operators, the representation of the search space and the type of optimization problem they are designed for, their population model and their generational transition mechanisms. For a broad overview of current EA concepts the reader is referred to [2] and [1].

The TEA C++ library can be used to build completly new EAs or to modify and/or apply the existing (standard) EAs.

The following targets have been adressed in the design of TEA:

- Representation-independent algorithm design

- Support of non-standard representations with mixed chromosome types
- Possible interactive exchange of search operators (without re-compilation)
- Pre-defined standard EAs and representations

Representation-independent algorithms require only minimal information about the individuals, which is typically their fitness value and their feasibility. Implementing algorithms this way, allows the user to apply the same algorithm with different individual types.

With the support of non-standard representations the configuration of complex genotypes is possible. For example in a complex genotype real vectors, bitstrings and integer arrays may be combined. TEA allows to inherit specific operators working on the different parts of the genotype. This allows the user a fast development of algorithms.

Search Operators like mutation, initialisation and recombination are defined as class-objects in TEA that communicate with the data objects (populations, individuals and chromosomes) they modify. Search Operators can be exchanged during running time and their parameters can easily be modified. This gives the user the opportunity to implement complex hybrid algorithms. Moreover, user interactions during the running time of the algorithm are possible.

Often the intention of the user is just to apply a standard EA for a given problem or a slightly modified EA. Therefore, TEA includes pre-defined configurations of the most common EAs like GAs and ESs. They can easily be applied for a given optimization task and as a template for new EA configurations.

TEA is an abbreviation that means 'Toolbox for Evolutionary Algorithms'. The term Toolbox has been chosen to point out that the TEA package contains some more programs than the C++ library, like tools to build a simple GUI. Nevertheless, we will focus in this paper on the features of the C++ library.

In the following the architecture of the TEA library will be explained. An example will be given, that demonstrates how to configure an evolutionary algorithm. Lateron we discuss some technical details, like the directory structure and the installation process. The paper continues with an overview of existing components, i.e. evolutionary algorithms and representations in TEA. We conclude the paper with a brief summary of features and discussion of limitations of the library.

# 2 Components and Structure of the TEA Library

The TEA library works with three main aggregation levels - the chromosome, the individual and the population - as it is depicted in Figure 1. All these objects have their specific operators, which interface is defined in the virtual abstract classes `teaChromosome`, `teaIndividual` and `teaPopulation`.

Genotypes in TEA are build of one or more chromosomes. Typically chromosomes are sequences of one data-type, like real vectors or bitstrings. For each chromosome ($C$) the following operators are specified:

- *mutate* : Mutates the given chromosome
- *recombine* : Recombine a set of new chromosomes from a given set of chromosomes
- *init* : Initialise the chromosome with a start value

An individual contains all chromosomes of the genotype and may contain further (phenotypic) information, that is needed to calculate the fitness function. Fitness and constraint evaluations and the management of a set of chromosomes are typical methods of these objects. For each Individual ($I$) these operators are specified:

- *mutate* : Mutate the chromosomes of the given individual
- *recombine* : Recombine a set of new individuals from given set of individuals
- *getFitness* : Get the individuals fitness object
- *checkConstraints* : Return the severity of constraints violations

A population comprises one or more sets of individuals. Simple populations contain only two sets - the parent population and (temporarily) an offspring population. The operators of the population control the generational transitions for these sets of individuals. For each Population ($P$) these operators are specified:

- *evolve* : Evolve the given Population for a specified number of generations
- *getPartners* : Choose mating partners for recombination from the population
- *recombine* : Generate an offspring population by drawing individuals from parent generation and recombine them
- *mutate* : Mutate all or some selected individuals from the offspring population
- *replace* : Select individuals of parental and offspring population to generate a new parent population
- *terminate* : Checks termination criterion

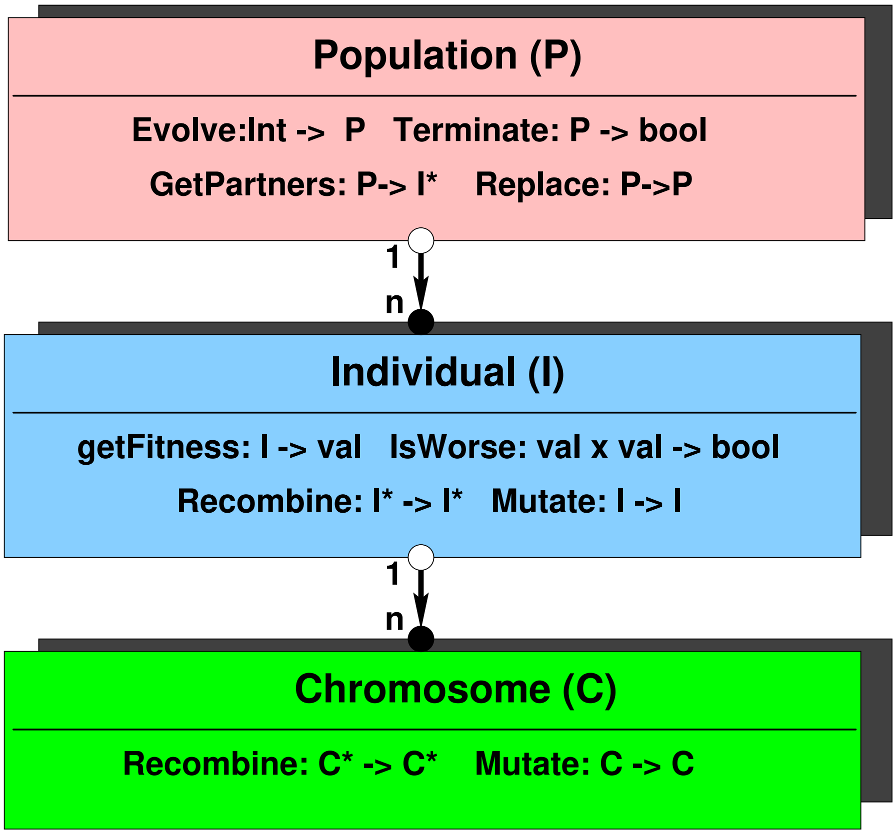


Figure 1: The aggregation scheme of the TEA library and basic operators for the objects chromosome, individual and population

There are many standard population models, individuals and chromosomes that have already been implemented in the TEA library. They all inherit the basic interface of `teaPopulation`, `teaIndiviual` or `teaChromosome`. Figure 2 gives a schematic overview of these objects and their inheritance scheme. We give a more detailed description of these objects in section 4.

## 2.1 Example for the Configuration of an EA

It is quite easy to configure an EA from existing TEA components. The following source code is an exerpt from the source code of `teaPGAExample`.

```
/*  C H R O M O S O M E */
teaCBitVector* cbit = new teaCBitVector();

cbit->resize(10);        /* Set Dimension of BitVector */
```

```
cbit->createOperators();  /* Create Default Operators */

/* Exchange Mutation-Operator */
teaCBitvectorMutate* SimpleMutator=new teaCBitvectorMutate();
delete cbit->getMutate();
cbit->setMutate(SimpleMutator);

/* Initialise the object- and strategy-parameters */
cbit->init();

/*  I N D I V I D U A L   */
teaISimple* myISimple = new teaISimple;
myISimple->createOperators();

myISimple->init(cbit);  /* Set chromosome prototype */

/* Set the Fitness Function */
teaICFCountOnes *fit_fun = new teaICFCountOnes();
delete myISimple->getCalcFitness();
myISimple->setCalcFitness(fit_fun);

/*  P O P U L A T I O N    */
/* Construct a  GA population object*/
teaPGA *myGAPop  = new teaPGA;
myGAPop->createOperators();

/* Set the prototype individual and Population Size (30) */
myGAPop->init(30,myISimple);

/* set Crossover Probability */
teaPGAEvolve* myEvolve = (teaPGAEvolve*)myGAPop->getEvolve();
myEvolve->crossoverProbability=0.5;

/* S T A R T   E V O L V E */
myGAPop->view(3);
myGAPop->evolve(100);
myGAPop->view(3);

/* ... delete objects ... */
```

In Figure 3 a graphical visualization of this procedure is given. We will now follow this routine step-by-step to give a first impression on how EAs are build in TEA.

First, control parameters from the C++ argument vector or a GUI may be received. As a simple GUI for for parameter handling on X/UNIX systems we recommend `tkjoe` which is part of the TEA software package and can be easily integrated into a C++ procedure.

After control parameters are received, a prototype individual has to be designed, which later serves as a template for the individuals in the population. We begin with the construction of chromosomes for this prototype individual. To each chromosome genetic operators (mutation and recombinations) are assigned. These operators are class-objects themselves and need to be initialized. To simplify this procedure the method `createOperators` can be used to set default operators. The specification of chromosomes is completed by initialising them with start values.

Then, the completed chromosomes with their operators is inserted into a new individual. After this, the operators of this individual are constructed. One of these operators is the fitness function object. A fitness function object inherits the interface of the class `teaICalcFitness`, which is specified in `teaOperator.h`. This object serves to calculate the fitness value of an individual and check the constraints. The fitness value is an object with a comparison function, named `teaValue`. It may contain an integer (`teaIntValue`) or a double value , or even a vector of numbers. It is very important to choose a fitness-function which is compatible with the individual and its chromosomes, e.g. there are fitness functions that expect real vectors as chromosomes and others that expect binary strings as chromosomes. These informations should be all contained in their header file. A collection of different fitness function objects can be found in the directory `fitfun`.

The completed individual is now passed to a new population. During the initialization of the Population the prototype may be copied several times, in order to build up the first parent generation. This is done automatically by the `init()` method. The next step is to specify the evolutionary operators for the population, like the selection or replacement operators and the termination function.

The completed population can now be evolved step-by-step applying the `evolve` operator. As it can be seen in the examples it is possible to access and view all data of the objects during the evolution. Furthermore strategy parameters may be modified and even operators may be exchanged during the evolution loop. This can all be controlled in the main procedure.

Last but not least, the best fitness value that occurred in the evolution process is presented and all remaining objects are deleted.

# 3 Installation and Support

The installation of TEA is simple, if the Gnu-C++ compiler is used (e.g. version 2.9.5.2, which is downloadable from the location www.gnu.org/software). The installation process shall be summed up briefly in this section.

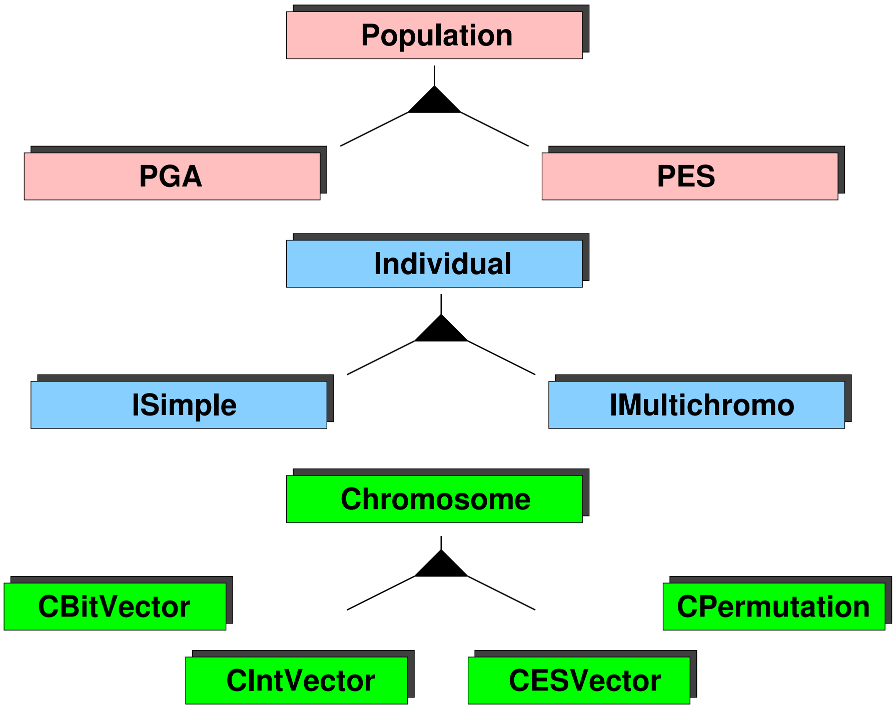


Figure 2: The inheritance scheme and implemented chromosomes, individuals and populations of the TEA library.

## A. System prerequisites

As system prerequisite a C++ compiler (e.g. Gnu-C++) and the tool 'gmake' is required.

## B. Installation

- The configuration file "Makefile.in" must be adapted to the System Environment (e.g. g++ for Sun Solaris and CC for SGI-IRIX Systems). The current settings are optimized for SUN Solaris 5.6.

- Set the TEA_DIR environment variable. It should contain a path (dirname) to the TEA library.

| | |
|---|---|
| Bourne-Shell : | export TEA_DIR=dirname |
| C-Shell : | setenv TEA_DIR dirname |

- cd $TEA_DIR

- make

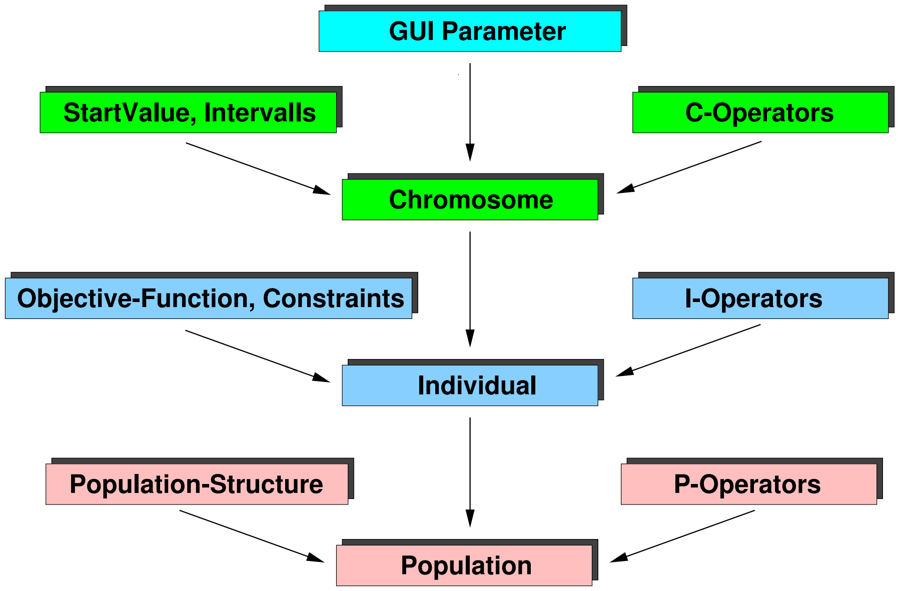


Figure 3: Data-Flow Diagramm illustrating how to build up an EA in a typical main procedure as it is described in section 2.1.

Information about TEA and a website can be found at the homepage of the Chair of Systems Analysis, Computer Science Department, University of Dortmund.

```
website: http://ls11-www.cs.uni-dortmund.de
```

# 4 Description of components in the TEA Package

This section informs the reader about the capabilities of the TEA library and provides important hints on the directory structure.

Experience shows, that it is often much easier to start building a TEA application, by starting with an example implementation. Therefore we begin this section with the introduction to some predefined examples.

## 4.1 Examples

An easy way to learn about building algorithms in TEA is to have a close look at some of the pre-defined examples, included in the software package. These examples consists only of one short main file, specifying the algorithm and its settings.

| | |
|---|---|
| First make Examples with : | make EXAMPLES |
| You can find the examples in: | cd $TEA_DIR/Examples |

### 4.1.1 teaPESExample:

This is an example for a standard $(\mu, \kappa, \lambda)$-ES [7]. Here $\kappa$ denotes the maximal life-span of each individual, e.g. if $\kappa = 1$ we get a $(\mu, \lambda)$-ES and for $\kappa = \infty$ we get a $(\mu + \lambda)$-ES. The ES is applied for the minimisation of the Sphere-Model (Sum of Squares). It works on a real-valued ES-Vector representation with adaptive step-sizes for each parameter.

### 4.1.2 teaPGAExample:

This is an example for a standard GA with fitness-based roulette-wheel selection. The algorithm is applied for the Counting-Ones-Problem and it minimises the number of ones in a bitstring.

### 4.1.3 teaPHypergraphExample:

An example for the application of a structured population [8] model. It illustrates how to define a population structure, i.e. the sets of individuals for recombination and the sub-populations, and how to apply this in a parallel EA.

### 4.1.4 teaIMultiChromoExample:

An example for a non-standard representation with different types of chromosomes contained by one individual. Here the individual contains a bitstring and a real-valued vector. The objective is the minimisation of a mixed-binary quadratic sum, that is optimised with an Evolution Strategy.

### 4.1.5 teaCESVectorExample:

In this example the various features of a real valued chromosome and its specific operators are introduced.

## 4.2 Chromosomes

For Chromosome implementations see `$TEA_DIR/Chromosomes`.

For header files see $TEA_DIR/inc.

The general interface of a chromosome object in TEA is defined in `teaChromosome` and the definition of its operators is found in `teaOperator`. There are some methods that can be unified for almost all vector representations. These are for example the access of values, upper and lower bounds and step sizes. The unified interface is declared and defined in `teaCVector`.

**Continuous vectors** are implemented in the class `teaCESVector`. It is possible to limit the domain of these vectors to an interval, specifying upper and lower bounds for each variable. The mutation operator works with gaussian distribution and a mutative global or local step-length adaptation [7], [1]. For the recombination of variables and step-length the user may choose between discrete and intermediate recombination (global or local) [7].

**Sequences and permutation** representations are implemented are often applied for scheduling problems. In TEA, with `teaCPermutation` a permutation chromosome is pre-defined. The PMX and OX crossover operator as described by Michalewicz [5] are implemented as recombination operators. The mutation operator is a simple shift operation, moving one entry of the sequence to a randomly chosen new position.

**Integer vectors**, that may represent more than two values on each positions, are implemented in the class `teaCIntVector`. For each position of the integer array an upper and lower bound can be defined. As a default, operators are defined, that assume that numbers belong to an ordinal scale. This may be the case, whenever the integer variable decribes the size or position of some entity. As mutation operator the geometric mutation as described by Rudolph [6] serves. It is assumed that small variations of the integer numbers result in small variations of the fitness value.

Tuples of **discrete variables** [3] taken from a finite domain are also supported by TEA. In this case it is assumed that there is no predefined order on the variable's domain. The chromosome `teaCIntVector` implements these discrete tuples. By a parameter switch it is defined if the integer values between the lower and upper bound should be treated as an ordered set or as an discrete variable. In the mode, when integer chromosomes are treated as discrete tuples, mutation is done by choosing with a certain probability randomly a new value in the finite domain defined by interval bounds.

## 4.3 Individuals

For individual implementation see `$TEA_DIR/Individual`.

For header files see `$TEA_DIR/inc`.

For the implementation of fitness objects see `$TEA_DIR/src`.

The general interface of an individual object in TEA is defined in `teaIndividual` and the definition of its operators is found in `teaOperator`. Furthermore, the interface of a fitness object can be found in `teaValue`.

### 4.3.1 Simple individual

An individual that contains only one chromosome is implemented with `teaISimple`. The function of this object is that it manages one fitness object and the chromosome. Its search operators pass the tasks to corresponding search operators of the chromosome, e.g. the mutation operator invokes the mutation operator of the comprised chromosome.

### 4.3.2 Multichromosomal individual

An individual that contains more than one chromosome is implemented with `teaIMultiChromo`. The mutation, initialisation and recombination operators of each chromosome are used, thereby using the general interface of a chromosome class.

## 4.4 Fitness objects and constraints

For Examples of fitness functions, see `cd $TEA_DIR/fitfun`.

For fitness objects, see `$TEA_DIR/inc`.

For implementation of fitness objects, see `$TEA_DIR/src`.

A very important operator in TEA is the fitness calculation. The operator `teaIndividualCalcFitness` is the basic class for all fitness operators and specifies the methods `getConstraints` and `calcFitness`. In a specific fitness operator, e.g. `teaICFSphere`, these functions access chromosomes of the individual to calculate the fitness value. The type of the chromosome has to correspond to the type, the operator exspects.

The general interface of a fitness value can be found in `teaValue`. A fitness value can be compared to other fitness values by the functions `isLess()` and `isEqual()`.

Often a fitness object is given by just one real value. In this case `teaDoubleValue` should be used. For integer values it is recommended to use `teaIntValue`. Both fitness objects offer the method `getScalar`. This method returns the fitness value as an absolute value. This value can be interpreted as the target function value or as a measure for the constraint violation.

For **multicriteria optimization** with pareto vectors [4] TEA offers a vector valued fitness object called `teaVectorValue`. On this object the functions `isLess()` and `isEqual()` define a partial order. A fitness object is smaller than another fitness object only if it is dominated by all positions of the double vector.

### 4.4.1 Multi-Chromosomal Individual

For real-world optimization problems we often need to deal with mixed represention, e.g. some decision variables are of discrete and some are of integer type [3].

An individual that contains just more than one chromosome is implemented with `teaIMultiChromo`. It manages a list of chromosomes and a fitness function object that evaluates this list. The chromosomes in the list can be of different types. This enables the user to build mixed representations, e.g. to combine binary strings with real vectors in one individual. The search operators of the individuals, delegate the tasks to the search operators of the specific chromosomes.

## 4.5 Populations and Algorithms

For population implementations see $TEA_DIR/Population.

For header files see $TEA_DIR/inc.

In TEA a population object aggregates individuals and has the ability to evolve this set of individuals, i.e. it implements the main loop of the evolutionary algorithm including the partner selection, termination and replacement operators.

The general interface of a population object in TEA is defined in `teaPopulation` and the declaration of its operators is found in `teaOperator`.

### 4.5.1 Evolution Strategies

Evolution strategies have been originated by Schwefel and Rechenberg in the early sixties. They have proven to be robust derivative free global optimization algorithm for nonlinear functions. Evolution Strategies usually employ representations like real or integer arrays. They have the feature that they can self-adapt parameters of the mutation-distribution during the evolution.

The class `teaPES` implements the common $(\mu + \lambda) - ES$ and $(\mu, \lambda) - ES$ as described by Schwefel [7]. Here $\mu$ is the number of parent individuals and $\lambda$ the number of offspring individuals in each generation. Standard ES-variants that work with real vectors and integer arrays can be cofigured using the chromosomes `teaCESVector` or `teaCIntVector` and `teaISimple`.

### 4.5.2 Genetic Algorithms

The basic ideas in GAs are quite similar to that in ESs. They mainly differ in the chromosome representation and generational transition. The standard representation for GAs is usually a bitstring. Furthermore a probablilistic roulette-wheel selection of recombination partners takes place. Standard GA variants are configured in TEA using the population object `teaPGA`, the chromosome object `teaCBitString` and the individual object `teaISimple`.

In the generational loop individuals of the population are selected, recombined and mutated. The result is a number of individuals which represent the new population. This population is the basis for the next evolution step. Dependent on the fitness value, individuals are chosen for producing descendant individuals. This is done by the probabilistic roulette wheel selection. The effect is, that individuals with a high fitness value are more often chosen than individuals with a low fitness. Two operators - rank based partner selection and linear fitness scaling [1] are implemented.

### 4.5.3 Structured Populations and Parallel EAs

In structured population models [2, 8] like they are found in neighbourhood and multipopulation EAs, the selection operator works on local subsets of the entire population. By this measure a higher robustness of algorithms on multimodal fitness functions can be achieved and the EAs can aesily be runned in a parallel mode in multiprocessor environments.

TEA allows the realization of a large number of structured population models. As a description, the unified hypergraph model by Sprave [8] is used. The specification of a model can be done in a simple way by defining subpopulations and (time-variable) sets of recombination partners called *demes*.

All these structured population models are defined in the class `teaPHypergraph`. The example `teaPHypergraphExample` illustrates how these objects can be applied. Furthermore, it is posssible to run all these algorithms as well in sequential mode as in distributed mode in multiprocessor environments (using MPI library).

# 5 Summary and Outlook

The TEA library offers a flexible framework for evolutionary algorithm design. It is available on different platforms and coded in a simple C++ style. Its main features are representation-independent algorithms, support of standard algorithms (mainly ES and GA), support of state-of-the-art parallel models and distributed evaluations and last but not least some of the most important representations.

There are also some limitations in the TEA library, so far. As it is not the intention of TEA, to be a large collection of Evolutionary Algorithms. Therefore many algorithms that can be found in literature are not implemented in TEA, but in most cases TEA will provide some of the components that are needed when designing them.

Furthermore, it is exspected that the user of TEA has already some experiences in C++ programming and is familiar with his/her specific graphical visualisation and data-analysis tools. Therefore the effort of the TEA development has been put in the library features rather than into the design of elaborate visualisation tools and graphical user interfaces, so far.

Since statistical benchmark tests may be performed for new EAs with TEA, an emphasize has been put to a fast and efficient implementation. Furthermore, it should also be easy to understand the source code. To achieve this, the design is kept simple and often simple data-types are used instead of complex data-objects.

The development of new components for the TEA library continues. A focus of the current work on the library is the control of the components quality and correctess. e.g. by benchmark comparisons, and the development of parallel evolutionary algorithms and further representations.

## Acknowledgements

This work was supported by the Deutsche Forschungsgemeinschaft (DFG) as part of the *Collaborative Research Center 'Computational Intelligence' (SBF 531).*

## References

[1] Th. Bäck. *Evolutionary Algorithms in Theory and Practice.* Oxford University Press, New York, 1996.

[2] Th. Bäck, D. B. Fogel, and Z. Michalewicz, editors. *Handbook of Evolutionary Computation.* Oxford University Press, New York, and Institute of Physics Publishing, Bristol, UK, 1997.

[3] M. Emmerich, B. Groß, M. Grötzner, and P. Roosen M. Schütz. A mixed integer evolution strategy for chemical plant optimization. In *In: I. Parmee, Evolutionary design and manufacture ACDM, Plymouth UK*, pages 55–67. Springer, NY, 2000.

[4] Frank Kursawe. A Variant of Evolution Strategies for Vector Optimization. In H.-P. Schwefel and R. Männer, editors, *Parallel Problem Solving from Nature — Proc. 1st Workshop PPSN*, volume 496 of *Lecture Notes in Computer Science*, pages 193–197, Springer, Berlin, 1991.

[5] Z. Michalewicz. *Genetic Algorithms + Data Structures = Evolution Programs.* Springer, Berlin, 1996.

[6] G. Rudolph. An evolutionary algorithm for integer programming. In Y. Davidor, H.-P. Schwefel, and R. Männer, editors, *Parallel Problem Solving from Nature - PPSN III*, Lecture Notes in Computer Science, pages 139–148, Springer. Berlin, 1994.

[7] H.-P. Schwefel. *Evolution and Optimum Seeking.* Sixth-Generation Computer Technology Series. Wiley, New York, 1995.

[8] J. Sprave. A unified model of non-panmictic population structures in evolutionary algorithms. Technical Report CI 55 99 SFB 531, University of Dortmund, 1999.