xegaX: A Family of R-Packages for Genetic and Evolutionary Algorithms with Multiple Genome Representations

Geyer-Schulz, Andreas

Abstract `xegaX` is a family of R-packages for genetic and evolutionary algorithms with multiple gene representations. At the moment, the following gene representations are supported: Binary genes, integer genes, real genes, and derivation tree genes. The package provides a common framework for genetic algorithms with binary genes (sga), genetic differential evolution algorithms (sgde), genetic algorithms with integer permutations (sgPerm), grammar-based genetic programming algorithms (sgp), and grammatical evolution algorithms (sge). The packages have a layered architecture with 4 layers: The (top-level) main program layer, the population layer which is independent of the gene representation, the gene layer which is split into gene representation dependent (initialization, crossover, mutation, and decoding) and gene representation independent (selection, evaluation) components. In addition, several innovations have been integrated into the package with the aim to improve several architectural goals simultaneously: Increased flexibility, configurability, and extensibility combined with performance improvements and scalability. For example, extensive support for parallel and distributed processing has been added: Multicore processing on notebooks (Linux only), distributed processing on clusters of servers on a local area network (for security reasons), parallel processing on high-performance processor clusters based on rmpi. This paper will give

---

Geyer-Schulz, Andreas
Karlsruhe Institute of Technology (KIT), Kaiserstraße 12, D-76131 Karlsruhe, Germany.
✉ `Andreas.Geyer-Schulz@kit.edu`

ARCHIVES OF DATA SCIENCE, SERIES A
(ONLINE FIRST)
KIT SCIENTIFIC PUBLISHING
Vol. 10, No. 1, 2025

DOI 10.5445/IR/1000187255
ISSN 2363-9881

an architectural overview of the packages as well as a description of selected innovations and their impact on the architectural goals.

1 Introduction

In the development of the architecture of the `xegaX` family of R-packages for extended and evolutionary algorithms, the main architectural goal has been to combine

1. flexibility, extensibility, and reconfigurability of algorithms
2. with high performance and scalability on multi-core machines, local area networks and on high performance computing clusters.

The first set of goals is motivated by more than two decades of work on parameter tuning, automatic algorithm selection, and self-adaptivity in evolutionary and genetic algorithms (e.g. Aleti and Moser (2016)) as well as a proliferation of algorithm variants with different gene representations (e.g. Rothlauf (2006)). The vision of the grid as a new computing infrastructure (Foster and Kesselmann, 1999) and its realization in the last two decades is the rationale for the second set of goals. In the last few years, we witnessed an increasing availability of parallel and distributed processing from multi-core CPUs in notebooks, clusters of networked machines in departmental LANs as well as large high-performance computing (HPC) clusters in research universities. In combination with the availability of a common operating system (Linux), communication middleware (TCP/IP, mpi), and the R software environment for statistical computing with middleware packages for all three types of systems this makes a seamless migration of extended evolutionary and genetic algorithms from the notebook to the HPC cluster possible.

This article provides a high-level description of the architecture of the `xegaX` family of R-packages and the architectural design decisions made. We distinguish between *users* of `xega` who want to solve an optimization or machine learning problem with an evolutionary or genetic algorithm and *developers* of evolutionary and genetic algorithms. We split the description of the `xegaX` architecture into three parts:

1. The analysis pipeline for the user of evolutionary and genetic algorithms is described in section 2.

2. The layered architecture of the evolutionary and genetic algorithm of the R-package `xega` is presented in section 3.
3. The architectural design of the `xegaX` family of R-packages, their design principles, and their rationale are explained in section 4.

Section 5 summarizes the main features of the `xegaX` architecture and gives an outlook on future extensions.

2 For the User: The Analysis Pipeline

A problem environment (see Fig. 1) provides a function factory which generates a problem environment (a named list of functions) for a genetic algorithm. E.g. the package `xegaEnvReal` will provide a set of benchmark functions for nonlinear function optimization. The user of the package `xega` is expected to formulate his problem as a function factory which produces a problem environment (Geyer-Schulz, 2024a). In addition, packages for constraint optimization, L1 and L2 regression, knapsacks, and traveling salesman problems will be provided.

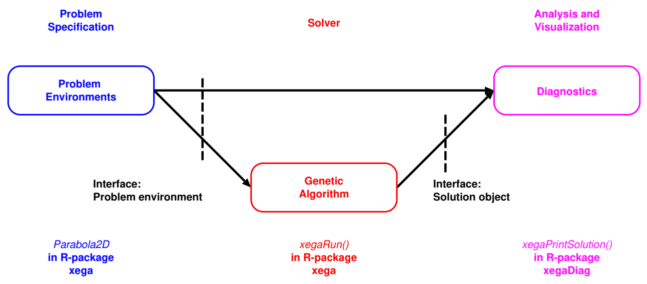
Problem Specification
Solver
Analysis and Visualization
Problem Environments
Diagnostics
Genetic Algorithm
Interface: Problem environment
Interface: Solution object
Parabola2D in R-package xega
xegaRun() in R-package xega
xegaPrintSolution() in R-package xegaDiag

Fig. 1 The Analysis Pipeline.

The genetic algorithm (see Fig. 1) is the solver component. The solver package `xega` (with all variants of genetic algorithms) provides the function `xegaRun` as a command line user interface. With `xegaRun` the user chooses the algorithm variant by specifying the algorithm parameter:

1. `algorithm="sga"`: Binary coded genetic algorithm. See Goldberg (1989).

2. `algorithm="sgde"`: Real coded genetic algorithm. Currently only differential evolution (Price et al, 2005) is provided.
3. `algorithm="sgPerm"`: Permutation coded genetic algorithm. See e.g. Syswerda (1991) and Lin and Kernighan (1973).
4. `algorithm="sgp"`: Derivation tree coded genetic algorithm for context-free grammars. See Geyer-Schulz (1997).
5. `algorithm="sge"`: Grammar evolution with binary genes. See O'Neil and Ryan (2003) and Ryan et al (2018).

All diagnostic functions (see Fig. 1) currently available are collected in the package `xegaDiag` (in preparation). These provide printouts of a solution, plots of the algorithm's population statistics and of the problem environment as well as profiling and profile comparison functions to document where in the algorithm how much time is spent.

```
> library(xega); library(xegaDiag)
Loading required package: parallelly
> a<-xegaRun(penv=Parabola2D, algorithm="sga", max=FALSE, verbose=
    0)
> xegaPrintSolution(a)
Min   Parabola2D !
Fitness: -0.0008741791 f(Parameters): -0.0008741791
Parameters: 0.02612259 -0.01384879
Time used: 0.1407065 seconds.
```

Fig. 2 An End User Session: Find the minimum of a 2-D Parabola!

Fig. 2 shows a sample user session. In line 1, the R-packages `xega` and `xegaDiag` are loaded (set up of the R-environment). In line 3, `xegaRun` (the Solver) minimizes (`max=FALSE`) the (predefined) problem environment `Parabola2D` in package `xega` (the Problem Specification) with a binary-coded genetic algorithm (option `algorithm="sga"`) without console output (option `verbose=0`). For all other parameters, the default settings are used. For convenience, the function `xegaPrintSolution(a)` (line 4) from the R-package `xegaDiag` prints the fitness, the minimal function value, the parameters, and the time used to the console (Analysis and Visualization).

The interface between problem environment and solver is specified as a list of functions which is provided by the problem environment and required by the solver. For a genetic algorithm, the user **must specify** a problem environment. A problem environment `penv` is a list of functions written by the user which are required by the solver. A binary coded genetic algorithm as a solver requires a function list with the following functions as elements:

- `$name()`: Returns the name of the problem environment.
- `$bitlength()`: Vector of bit lengths of parameters.
- `$genelength()`: Length of gene in bits.
- `$lb()`: Vector of lower bounds.
- `$ub()`: Vector of upper bounds.
- `$f(p, gene=0, lF=0)`: The fitness function. `p` is the phenotype: For a binary genetic algorithm, `p` may represent the parameter vector of a real-valued function. `gene` and `lF` are the gene and the local function list which are passed to the fitness function. These parameters may be used for influencing the evaluation of the fitness by information in the gene or about the internal configuration and state of `xega` and the problem environment.

Additional elements may be provided, e.g. the element `$pname()` which returns a list of parameter names.

This specification of a problem environment is also sufficient for real-coded evolutionary algorithms: E.g., differential evolution. All fields which are only required for binary genetic algorithms (e.g. `$genelength()`) are ignored. The user specifies `algorithm="sgde"` instead of `algorithm="sga"`.

The interface between the solver and the analysis and diagnostic routines is a solution object which is a list with the following named elements:

- $popStat: Matrix of population statistics. 1 row per generation.
- $solution: Named list with genotype and phenotype information.
- $evalFail: Integer. 0 means no failed evaluations of fitness functions.
- $GAconfig: Solver call with all parameters. For reproducibility.
- $GAenv: Environment with variable bindings.
- $timer: Time spent in the main program and in the calls to the population level operations in the main loop.

A user of `xega` is expected

1. to know how to program a problem environment and, for grammar-based genetic programming and grammatical evolution, how to specify the Backus-Naur form of a context-free grammar as well as how to implement the language specified by the grammar so that a program in this language can be excuted in R and its fitness assessed. Examples for all algorithms available can be studied by typing `example(xegaRun)` on the R-command line.

2. to learn about the parameters and their interrelationships of the solver `xegaRun`. They are described in the documentation of `xegaRun`. On the R-command line, type:

```
> ?xegaRun
```

3 For the Developer: The Layered Architecture of `xega`

The `xegaX` family of R-packages uses a relaxed version of the *Layers* architectural pattern of Buschmann et al (1996a). Fig. 3 shows the layers of `xega`. Only the top-level R-package `xega` (boxed in red) is visible for the end-user (Geyer-Schulz, 2024a). For the developer, the R-package `xega` provides

1. the R-functions `xegaRun` (the solver) and `xegaReRun` (for repetition and reproducibility) which form the R-command line user interface (with documentation) and
2. a set of function factories whose names start with `sgX` for configuring gene initialization, gene maps, decoders, replication, mutation, and crossover operations in an algorithm specific way.

The blue box in Fig. 3 identifies the internal packages used by the R-package `xega`. The two main criteria for structuring the internal packages are the *scope of operations* and the *independence from the representation of a gene* (short: representation independence). The criterion *scope of operations* distinguishes between gene level and population level operations, whereas the criterion *representation independence* distinguishes between representation independent and representation dependent genetic operations. Both criteria are not orthogonal: At the gene level, representation independent operations (e.g. scaling and selection) coexist with representation dependent operations (e.g. mutation and crossover). Therefore, the internal packages are organized into two layers, namely the population layer (representation independent) and the gene layer which is split into a representation independent and representation dependent part. For grammar-based genetic programming and grammatical evolution, all grammar and derivation-tree related functionality is encapsulated in the packages `xegaBNF` and `xegaDerivationTrees`.

The dependency graphs of the R-packages for binary-coded genetic algorithms (`algorithm="sga"`) and grammar-based genetic programming (`algorithm="sgp"`) shown in Fig. 4 reveal a *relaxed layered system* ar-

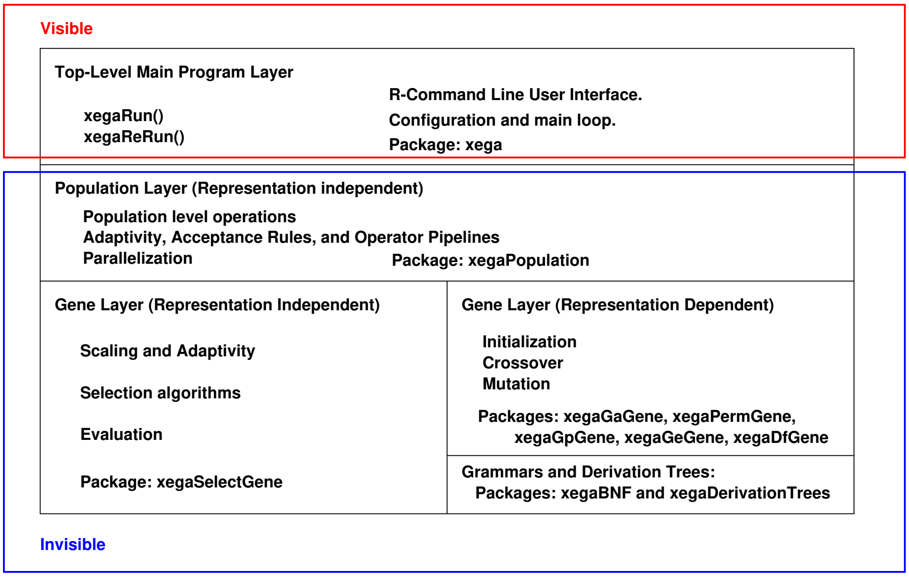
Visible
Top-Level Main Program Layer
xegaRun()
xegaReRun()
R-Command Line User Interface.
Configuration and main loop.
Package: xega
Population Layer (Representation independent)
Population level operations
Adaptivity, Acceptance Rules, and Operator Pipelines
Parallelization
Package: xegaPopulation
Gene Layer (Representation Independent)
Scaling and Adaptivity
Selection algorithms
Evaluation
Package: xegaSelectGene
Gene Layer (Representation Dependent)
Initialization
Crossover
Mutation
Packages: xegaGaGene, xegaPermGene, xegaGpGene, xegaGeGene, xegaDfGene
Grammars and Derivation Trees:
Packages: xegaBNF and xegaDerivationTrees
Invisible

Fig. 3 The Layers of `xega` Geyer-Schulz (2021)

chitecture (Buschmann et al, 1996b, pp. 45-46) where each layer may use the services of all layers below it. All R-packages of the `xega` family provide executable examples of all functions documented. The dependency graph for binary-coded genetic algorithms shows all dependencies necessary for providing working examples with broken arrows. The names of gene representation dependent packages are typed in blue, their support packages in magenta.

In the following, we give a bottom-up survey of the functionality provided by the internal R-packages of the `xegaX` family of R-packages.

The representation independent part of the gene layer is implemented by the package `xegaSelectGene` (Geyer-Schulz, 2024j). `xegaSelectGene` consists of four groups of functions:

1. Functions for selecting a gene in a population of genes according to its fitness value and for adaptive scaling of fitness values as well as functionality for performance optimization of selection functions. The most important selection methods discussed in the literature are provided: Selection proportional to fitness and fitness differences, stochastic universal sampling, tournament selection and stochastic tournament selection, rank selection, and, last but not least, uniform selection.

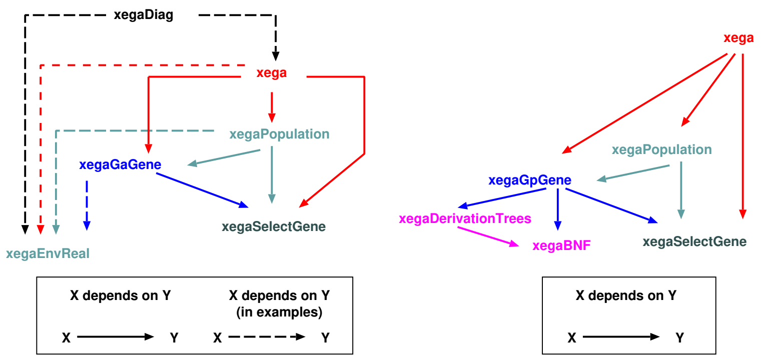

Fig. 4 Dependencies of `xega` for binary-coded genetic algorithms (left) and for grammar-based genetic programming (right)

2. Specialized evaluation functions for deterministic and stochastic problem environments and for gene error-correcting decoders.
3. Timing and counting functions for profiling of the main phases of a genetic algorithm for bottle-neck detection.
4. A small set of problem environments (one for each type of algorithm supported).

The population layer (independent of the representation of genes) is implemented by the package `xegaPopulation` (Geyer-Schulz, 2024i). The package consists of the following group of functions:

1. Functions for initializing, logging, observing, and evaluating a population of genes and of functions for computing the next population of genes. Sequences of gene operations are partially implemented as configurable pipelines of genetic operators.
2. Acceptance rules and cooling schedules for simulated annealing and for greedy randomized approximate search procedures.
3. Parallelization is restricted to the evaluation of a population of genes. Four different execution models (`Sequential`, `MultiCore`, `FutureApply` and `Cluster`) are provided. They are implemented by overloading the `lapply()` function of the function for evaluating a population of genes. In addition, example code for using these execution models and user-defined apply functions in several hard- and software environments is provided in the `example` directory of the author's GitHub directory for xega

https://github.com/ageyerschulz/xega. This includes support for `mpi` under the `SLURM` workload manager on HPC clusters.
`MultiCore`, `FutureApply` and `Cluster` assume that the cost of evaluating the fitness function is constant.
4. Support for the adaptivity of scaling, crossover, and mutation rates based on population statistics is provided.

The representation dependent part of the gene layer is implemented by five packages:

1. `xegaGaGene` for binary coded genes (Geyer-Schulz, 2024e). The package provides gene maps for decoding binary strings into k real parameter values (binary to decimal and gray codes), into an integer permutation for e.g. traveling salesman problems as well as the identity map e.g. for Knapsack problems.
   A standard mutation operator for bitwise mutation operations and an individually adaptive mutation operator are implemented. One-point crossover, uniform crossover and parametrized uniform crossover operations, as well as gene replication functions for one and two kids complete the functionality of the package.
2. `xegaDfGene` for real coded genes (Geyer-Schulz, 2024d). At the moment, the package supports a slightly extended implementation of the differential evolution algorithm of Price et al (2005). The extensions consist of random scaling and parametrized uniform crossover.
3. `xegaPermGene` for genes which are represented by permutations of integers (Geyer-Schulz, 2024h). The package provides position-based crossover (Syswerda, 1991) and a collection of mutation operators. The ideas for mutation operators come from three areas:

   a. From the area of genetic and evolutionary algorithms, order-based mutation (Syswerda, 1991) is implemented.
   b. From the rich field of heuristic algorithms, randomized Lin-Kernighan heuristics (Croes (1958) and Lin and Kernighan (1973)) and greedy randomized adaptive search procedures (see e.g. Resende and Ribeiro (2016)) are provided.
   c. From statistics and decision theory, a random mix operator for randomly selecting a mutation operator.

4. `xegaGpGene` for derivation trees (Geyer-Schulz, 2024g). The package implements the grammar-based genetic programming algorithms described in

Geyer-Schulz (1997) with the help of the abstract data type *derivation tree* of the package `xegaDerivationTrees`. The genetic operators are based on depth-bounded random subtree extraction and subtree insertion operations.
5. `xegaGeGene` for binary coded genes with a grammar-based decoder (Geyer-Schulz, 2024f). This package implements grammatical evolution algorithms (see O'Neil and Ryan (2003) and Ryan et al (2018)) on fixed length binary genes by providing a gene decoder which generates a derivation tree: The gene is decoded into a sequence of integers which represent the choices of production rules available for a non-terminal symbol. The process of generating the derivation tree is controlled by repeatedly choosing the production rule and inserting it into the derivation tree. All other genetic operations are reused from the package `xegaGaGene`. The current version can be improved by allowing for variable-length binary coded genes and repair mechanisms for completing derivation trees.

The package `xegaBNF` (Geyer-Schulz, 2024b) is a preprocessor necessary for grammar-based programming algorithms and grammatical evolution algorithms. The package `xegaBNF` compiles a context-free grammar in Backus-Naur form into a symbol table, a production table, a short production table which consists of all possible shortest substitutions of non-terminal symbols by sequences of terminal symbols, and the start-symbol of the grammar.

The package `xegaDerivationTrees` (Geyer-Schulz, 2024c) provides functions for

- generating random derivation trees,
- decoding derivation trees,
- filtering derivation trees,
- extracting derivation trees from derivation trees and inserting derivation trees into derivation trees, and
- measuring derivation tree attributes (depth, size as well as number of terminal and non-terminal symbols).

4 The Architectural Design of the xegaX Family of R-Packages

Recent developments like attention-based genetic algorithms (Lange et al (2023a) and Lange et al (2023b)) require genetic operators with a sufficiently flexible parametrization to discover entirely new genetic algorithms in a data-driven fashion. To achieve the resulting architectural goals of flexibility, extensibility, and reconfigurability of evolutionary and genetic algorithms in the `xegaX` family of R-packages, the fact that the R language treats functions as first order objects is used: An R function consists of formal arguments, a function body, and the parent environment (the name/value pairs) in which the R function was created. R functions can be assigned to lists and called from function lists. Function lists can be passed as arguments to R functions. These properties of the R language allow a direct realization of the separation of the implementation of a framework from its changeable parts (called hot spots) as presented by Backus (1978) in his Turing award lecture.

4.1 The Separation of a Framework and Its Changeable Parts

The following example shows the functional programming approach used in the `xegaX` family of R-packages with a small, but working example of a generate-and-test function for approximately solving small random k-city traveling salesman problems (TSPs). The implementation clearly separates the framework (the generate-and-test function) from its changeable parts (the objective function of the problem, the initialization of a solution, and the generation of random neighbors).

1. We define the objective function `f` for the problem (changeable parts). For the TSP, we compute the length of a round-trip represented by a permutation of *k* cities:

   ```
   f<-function(distances, permutation)
   { cost<-0
     l<-length(permutation)-1
     for (i in 1:l)
        { cost<- cost+distances[permutation[i], permutation[i+1]]}
     cost<-cost+distances[permutation[l+1], permutation[1]]
     return(cost)}
   ```

2. We define the function `init()` which generates a random start solution candidate (a permutation of size k) for the k-city TSP problem (changeable parts). Note, that this functions requires information about the number of cities in the problem.

   ```
   init<-function(k)
   {sample(k, k, replace=FALSE)}
   ```

3. We define three functions `neighbor()` for generating a random neighbor (changeable parts). The first neighbor function is a swap of two randomly selected cities. The second neighbor function performs two swaps and uses the definition of the first neigbor function, while the third neighbor function is defined as a complete random permutation of k-cities.

```
neighbor2<-function(permutation)
{ pos<-sample(1:length(permutation), 2, replace=FALSE)
tmp<-permutation[pos[1]]
permutation[pos[1]]<-permutation[pos[2]]
permutation[pos[2]]<-tmp
return(permutation) }

neighbor3<-function(permutation)
{ return(neighbor2(neighbor2(permutation))) }

neighborRnd<-function(permutation)
{return(init(length(permutation)))}
```

   Note that all neighbor functions have the same function arguments and return the same type of R object. A common interface is the requirement to have exchangeable neighbor functions.

4. We define the function `generateAndTest()` which takes the problem definition, the number of trials and a list of local functions `lF` as arguments (the framework). The implementation of the changeable parts is represented by the function bodies in `lF` and passed as a list of functions to the framework function `generateAndTest()`. `t` is a random trial solution and `vt` its value.

```
generateAndTest<-function(problem, problemsize,  trials, lF)
{t<-lF$init(problemsize); vt<-lF$f(problem, t)
 for (i in (1:trials))
   {cat("Value␣t:", vt, "\n")
    newt<-lF$neighbor(t); vnewt<-lF$f(problem, newt)
    if(vt>vnewt) {t<-newt; vt<-vnewt}
 return(t)}
```

5. Before we use the framework, we must set up the local functions list `lF` with concrete instantiations of its changeable parts:

```
lF<-list(); lF$init<-init; lF$neighbor<-neighbor2; lF$f<-f
```

6. Last, but not least, we provide a random problem generator for k-city TSPs, namely the function `kTSP`. The framework `generateAndTest` expects the `problem` to be a $k \times k$ random numeric matrix. The return object of `kTSP()`, therefore, is a $k \times k$ matrix:

```
kTSP<-function(k)
{matrix(nrow=k, ncol=k, byrow=TRUE, sample(1000, k*k, replace=
    TRUE))}
```

7. We set up a 10 city problem and start to solve it with the simple random inversion operator `neighbor2`.

   ```
   a<-kTSP(10)
   b<-generateAndTest(a, 10, 20, lF)
   ```

8. However, we may also use a combination of two inversion operations:

   ```
   lF$neighbor<-neighbor3
   b<-generateAndTest(a, 10, 20, lF)
   ```

   Or retry just random permutations from a fixed initial permutation:

   ```
   lF$init<-function(k) {1:k}
   lF$neighbor<-neighborRnd
   b<-generateAndTest(a, 10, 20, lF)
   ```

In this example, we have only shown how different operators of computing random neighbors of a solution can be used by a framework function. We have made no effort to make the framework reusable across different classes of problems or to identify (and use) other exchangeable parts (or hot spots) as e.g. the termination condition of the main loop.

The identification of hot spots can be supported by the use of pseudocode. E.g. for the `xegaX` family of packages the pseudocode of the main loop of a genetic algorithm with an extended genetic operator pipeline has been used to identify the hot spots (the reconfigurable functions).

The crucial idea is the use of the local function list `lF`: It serves as a container which collects all functions implementing the exchangeable parts needed by the framework: All functions of the framework which have a changeable part function (e.g. `cpf0`) must have a local function list in their argument list und use the needed functions by proper function calls (e.g. `lF$cpf0()` for a constant function without arguments). Obviously, a changeable part function `cpf1` may itself have a changeable part (e.g. `cpf2` with an integer argument). Just pass `lF` as argument to `cpf1` (e.g. `lF$cpf1(lF)`) and use it by calling `lF$cpf2(k)` in the body of `lF$cpf1` with *k* having an integer value.

Function bodies can be dynamically assigned to named list elements in the container `lF`. However, they can only be used after the assignment and they can only be passed as arguments to functions at the same level or at a lower level of the function call hierarchy.

The advantages of the function container pattern used are that it

1. supports the construction of abstract and uniform interfaces which hide the details of the actual dependencies of a concrete implementation. In the `xega` family of packages, different low-level parametrizations of the behavior of concrete implementations of genetic operators are achieved by adding constant functions for the parameters to the local function list. E.g the function `lF$UCrossSwap()` for parametrized uniform crossover in the package `xegaGaGene` (Geyer-Schulz, 2024e).
2. defers the decision about the parameters needed for new concrete functions for extending the options available for a hot spot.
3. decouples the need to know about the concrete implementation of a hot spot. The functions using a hot spot do not need to know about its concrete implementation. Adding additional implementations of a hot spot or changes in the internal structure of the hot spot do not require changes in the code of their clients.
4. allows dynamic reconfiguration of the container `lF` in the top-down direction. The implementation shown above is an elegant example of the *reflection* pattern of Buschmann et al (1996c):

   a. The function container `lF` implements the *meta-level* of the system and makes the system self-aware.
   b. The framework represents the *base-level* of the system which implements the application logic by abstract calls to hot spot functions in the local function container `lF`. Changes to `lF` affect subsequent base-level behavior.
   c. The *meta-level protocol* which specifies how meta-objects are manipulated is simply implemented by the native assignment operator of R which allows the assignment of function bodies to list elements.

5. improves readability of the source code of the `xega` family of packages by using the convention that

   - a call to a hot spot function is always from the local function list `lF`.
   - a framework function has `lF` in its argument list.
   - a function may be a framework and a hot spot function simultaneously.

Container patterns have been suggested in the context of creating polymorphic interfaces to containers for container classes or iterators mainly with the goal of reusing control structures (e.g. loops) for different abstract data types (e.g. sets, bags, list, ...). See Martin (1995, pp. 367-369) and Rising (2000a, p.80). However, the container `lF` in the `xega` family of packages has been introduced with the purpose of changing the semantics of the hot spots of several classes of search and optimization algorithms both statically (by configuring) and dynamically (by adaptation). Note that Swan et al (2019) have presented a similar example (in `Scala`) and discussed a slightly more general solution for handling state-dependent components by *state-threading* in such a manner that the open-closed architectural principle holds.

4.2 Function Factories and the Organization of the Concrete Function Implementations for Hot Spots

In the `xegaX` family of packages, we distinguish between

1. hot spots which are independent of the gene representation and the algorithm chosen and
2. sets of hot spots which depend on the gene representation and the algorithm chosen.

For both variants, function factories (Wickham, 2019) are crucial for the selection of concrete functions for hot spots in a uniform way.

But let us walk through the process of designing and implementing a hot spot step by step. Unfortunately, the R language does not offer built-in support for specifying and checking abstract interfaces. As soon as a hot spot is identified, an abstract interface is specified which each concrete implementation of the hot spot must respect. For the `xegaX` family of R-packages, the abstract interfaces are specified as function signatures in the package documentation of a `xegaX` package by convention. For example, the abstract interface of selection functions is:

```
<vector of indices> <-
    <function name>(fit=<vector of reals>,
                    lF=<local function list>,
                    size=<number of indices returned>)
```

All concrete function implementations of a hot spot are collected in a single source file together with the function factory for the hot spot. A function factory for independent hot spots has a signature of the form

```
<body of concrete function> <-
    <function factory name>(method=<label>)
```

By convention, the last part of the name of a function factory is `Factory`. The factory takes the single argument `method` whose value is a text string with the name of the concrete function implementation selected. If no implementation for a text string is found, a run time error occurs. The function factory returns the body of a concrete function with which the named list element in the container `lF` is instantiated. This design pattern is essentially a function factory with an implementation of a case control structure to select the concrete implementation of a function. It is closely related to the design pattern *Factory Method (107)* of Gamma et al (1995b).

For example, the selection function stochastic universal sampling of the `xegaSelectGene` package is configured by:

```
lF$SelectGene<-xegaSelectGene::SelectGeneFactory(method="SUS")
lF$SelectMate<-xegaSelectGene::SelectGeneFactory(method="SUS")
```

For representation and algorithm dependent hot spots, a function factory has a signature of the form

```
<body of concrete function> <-
<function factory name>(algorithm=<factory label>, method=<label>)
```

The selection process is now implemented by two function factories: The factory label selects the representation and algorithm dependent function factory, the method label the concrete function implementation. All function factories of this type are part of the package `xega`. For example, configuring order based mutation for the TSP is done by

```
lF["MutateGene"]<-xega::sgXMutationFactory(algorithm="sgperm",
            method="MutateGeneOrderBased")
```

The function `xega::sgXMutationFactory()` chooses the function `xegaPermGene::xegaPermMutationFactory()` (algorithm="sgperm") which in turn chooses (method="MutateGeneOrderBased") and returns the function body of `xegaPermGene::xegaPermMutateGeneOrderBased()`. This combination of two function factories – one for the selection of a function factory for concrete methods and the second for choosing the proper method – is similar in its effect to the design patterns *Abstract Factory (87)* of Gamma et al (1995a) and *Pluggable Factory* in Rising (2000b, p. 132).

The advantages of this design are:

High Flexibility: Almost all function factories are *orthogonal*. This means that all possible combinations of the labels of the `method` arguments of the function factories configure an executable evolutionary or genetic algorithm. In an orthogonal design, the number of configurable different evolutionary or genetic algorithms is the product of the numbers of options of all function factories (see Table 1).

Improved Code Reuse: Additional methods implemented for a hot spot in a representation independent layer are automatically usable by all other algorithm variants. For a hot spot in a representation dependent layer, code reuse is restricted to the variants of one of type of algorithm (one of sga, sgde, sgperm, sgp, sge).

Dynamic Reconfigurability: The reflection pattern allows reconfiguration of the algorithm on the fly by standard assignment operations. This makes the implementation of all kinds of adaptive, self-adaptive, and meta-level evolutionary algorithms and genetic algorithms easy.

High Performance: The function factory pattern avoids the overhead of searching a method in the search path of a class hierarchy (method dispatch) of an object-oriented system for each method invocation. The cost of a method invocation is constant and equals the cost of accessing a named list element. Avoiding the cost of method dispatch of R's object-oriented programming versions (S3, S4, S7, R6, and RC) has been the main motivation for the function factory based architecture of the `xegaX` R-packages.

4.3 Hot Spots in `xegaX` *R-packages*

Table 1 groups the currently available hot spots in the `xegaX` R-packages in

1. representation independent and behavior related hot spots (in black in Fig. 5),
2. representation dependent and behavior related hot spots. (in red in Fig. 5),
3. and representation independent and performance related hot spots (in blue in Fig. 5).

Fig. 5 shows a simplified representation of the gene life cycle for the genetic algorithm variants (not for differential evolution) without population level details. Fitness scaling, mutation and crossover rate adaptation, gene and mate selection, as well as gene acceptance are presented in section 4.3.1, gene creation and decoding, mutation and crossover in section 4.3.2, and, last but not least, different gene evaluation strategies as well as the parallel evaluation of genes in section 4.3.3.

Hot Spot	Number of Variants				
xegaSelectGene::DispersionMeasureFactory()	6				
xegaSelectGene::ScalingFactory()	4				
xegaSelectGene::SelectGeneFactory()	14				
xegaPopulation::AcceptFactory()	4				
xegaPopulation::CoolingFactory()	6				
xegaPopulation::CrossRateFactory()	2				
xegaPopulation::MutationRateFactory()	2				
Representation Independent (Behavior)	32256				
xegaSelectGene::evalGeneFactory()	4				
xegaPopulation::ApplyFactory()	4+1				
Representation Independent (Performance)	20				
Independent Variants	645120				
Algorithm	sga	sgde	sgperm	sgp	sge
xega::sgXDecodeGeneFactory()	1	1	1	1	1
xega::sgXGeneMapFactory()	4	1	1	1	2
xega::sgXInitGeneFactory()	1	1	1	1	1
xega::sgXMutationFactory()	2	1	7	2	2
xega::sgXCrossoverFactory()	3	3	1	2	3
xega::sgXReplicationFactory()	2	1	2	2	2
xegaDfGene::xegaDfScaleFactory	-	2	-	-	-
Representation Dependent (Behaviors)	48	6	14	8	24
Variants	30965760	3870720	9031680	5160960	15482880
All Variants	64512000				

Table 1 Number of structurally different algorithms which can be configured in `xega` (Version 0.9.0.0, published 2024-03-20)

4.3.1 Representation independent and behavior related hot spots

Fitness Scaling. `xegaSelectGene::ScalingFactory()` provides 4 methods of fitness scaling:

1. Taking the identity function of the fitness (`"NoScaling"`, default) or
2. taking a power of the fitness with constant scaling exponent (`"ConstantScaling"`) as well as
3. the adaptive method threshold scaling (`"ThresholdScaling"`, and
4. the adaptive method continuous scaling `"ContinuousScaling"`.

Both adaptive methods change the scaling exponent depending on the change of the dispersion of fitness in the population after each generation. The change of dispersion is measured by the ratio of the dispersion at time t and $t-k$ where k specifies the time lag.

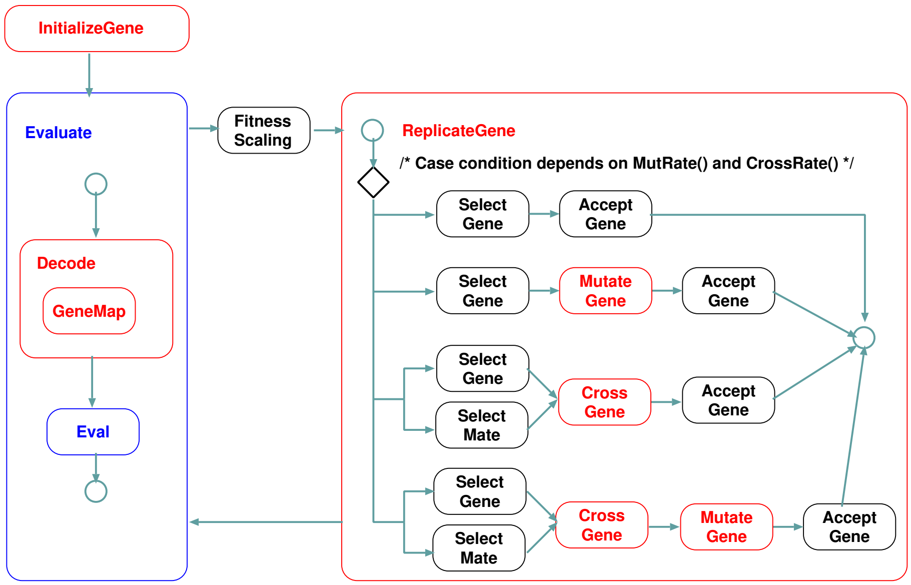

Fig. 5 The Gene Life Cycle of `xega` (simplified)

`xegaSelectGene::DispersionMeasureFactory()` allows the user to select the dispersion measure used. Available for selection are the variance (default), the standard deviation, the median absolute deviation, the coefficient of variation, the range and the interquartile range.

Mutation and Crossover Rates. The case condition which selects the sequence of genetic operators (the genetic operator pipeline) shown in Fig. 5 depends on the crossover and mutation rate functions which specify the probability that a crossover or mutation operator is applied to a gene.

They are configured by `xegaPopulation::CrossRateFactory()` and `xegaPopulation::MutationRateFactory()`. For both operators either a constant rate (`"Const"` or an individually variable rate (`"IV"`) can be configured. The idea of individually variable rates is that a gene with a low fitness compared e.g. to the population mean which is selected for reproduction should be changed (either by mutation or by crossover) with a higher probability. This is a generalization of an idea of Stanhope and Daida (1996).

Gene and Mate Selection. Selection functions implement Darwin's principle of the survival of the fittest. Their choice determines the selection pressure and thus the speed of convergence of genetic algorithms. Selection functions are an exhibit of different programming styles with functionally equivalent algorithms with different space and time complexity.

`xegaSelectGene::SelectGeneFactory()` provides a large menu of selection functions for gene and mate selection:

Proportional To Fitness. `"ProportionalOnln"`, `"Proportional"`, and `"ProportionalM"` choose selection functions proportional to fitness with different space and time complexity. `"SUS"` provides Baker's Stochastic Universal Sampling algorithm which has zero bias and minimal spread (Baker, 1987). These selection functions have the property that the selection pressure decreases over the run of the algorithm.

Proportional to Fitness Differences. The labels `"PropFitDiffOnln"`, `"PropFitDiff"`, and `"PropFitDiffM"` choose selection functions proportional to fitness differences. Taking the fitness differences instead of the fitness is a dynamic scaling operation which keeps the scaling pressure high over the run of the algorithm. However, if the population consists of copies of a single gene, selection of genes is performed with equal propability.

Linear Rank. `"LRSelective" and "LRTSR"` choose linear rank selection functions which implement Whitley's as well as Grefenstette and Baker's rank selection algorithms (see Whitley (1989) and Grefenstette and Baker (1989)). Rank selection functions are not influenced by fitness scaling.

Tournament. Selection pressure in tournament selection is regulated by tournament size. With deterministic tournament selection (`"Duel"` (2 genes) and `"Tournament"` (k genes)), the least fit gene in the population is not reproduced to the next generation. In stochastic tournament selection (`"STournament"`), the probability of survival of a gene participating in a tournament is proportional to the fitness of the other genes participating in the tournament. Deterministic tournament selection is not influenced by fitness scaling, stochastic tournament selection is.

Uniform. `"Uniform"` and `"UniformP"` both implement the selection of a gene with equal probability. `"UniformP"` produces a random permutation of the population, whereas `"Uniform"` may lose genetic material in reproduction. Both selection functions are needed for the configuration of evolutionary algorithms and simulated annealing algorithms as well as for the study of the evolution of populations without selection pressure.

Gene Acceptance. The classic versions of genetic, evolutionary, and simulated annealing algorithms (Van Laarhoven and Aarts, 1992) differ in the way the gene pipeline is configured (see Fig. 6). The combination of the choices of the two hot spots

1. `xegaSelectGene::SelectGeneFactory()` (for the selection functions) and
2. `xegaPopulation::AcceptFactory()` (for the acceptance functions)

determine the type of the resulting algorithm:

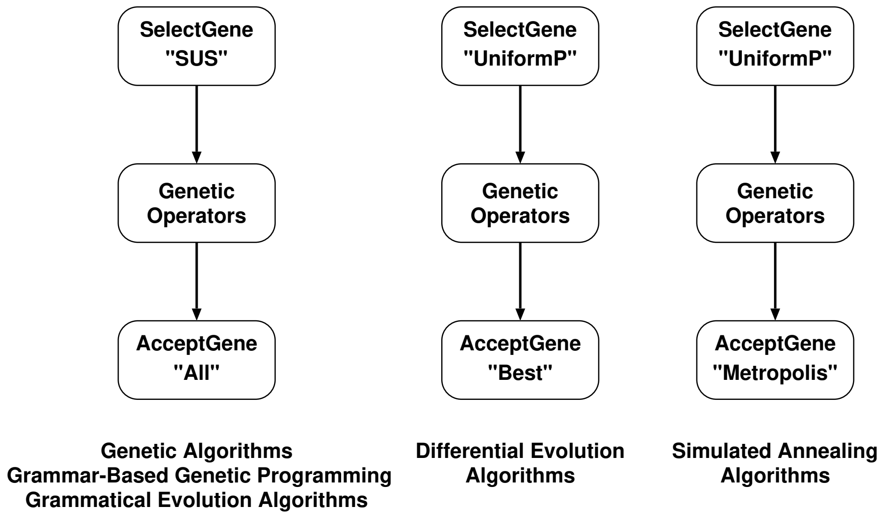

Fig. 6 Gene Pipelines for Different Algorithm Types: Choices of Selection and Acceptance Functions.

- Genetic algorithms select the candidates for reproduction in a fitness biased way, produce the kids by applying genetic operators (random neighbourhood operators), and accept all kids to the next generation (label `"All"` as argument of `xegaPopulation::AcceptFactory()`).
- Differential evolution algorithms randomly pick the candidates for reproduction, produce the kids by applying genetic operators (random neighbourhood operators), and accept only improved kids to the next generation (label `"Best"` as argument of `xegaPopulation::AcceptFactory()`).
- Simulated annealing algorithms randomly pick the candidates for reproduction, produce the kids by applying genetic operators (random neighbourhood operators), and accept worse kids to the next generation with a probability which decreases over time. With label `"Metropolis"` as argument

of `xegaPopulation::AcceptFactory()` the Metropolis acceptance rule (Metropolis et al, 1953) is configured.

`"IVMetropolis"` configures Mario Locatelli's upward temperature correction which depends on the distance between the gene under consideration and the best gene in the population. This correction improves the convergence of the algorithm to a global optimum (Locatelli, 2000).

The speed of convergence of simulated annealing algorithms and their chance to find a global optimum depend on the choice of a cooling schedule for updating the temperature. The hot spot for cooling is located at the end of the main loop of `xega::xegaRun()` and the configuration is done by choosing one of the 6 popular cooling schedules – namely exponential multiplicative and additive cooling, logarithmic multiplicative, power multiplicative and additive cooling, and trigonometric additive cooling – available in `xegaPopulation::CoolingFactory()`.

4.3.2 Representation dependent and behavior related hot spots

In the `xegaX` family of R-packages, there exist two groups of representation dependent and behavior related hot spots:

1. Hot spots for the abstract functionality of the main loop of the algorithm which are configured by function factories in `xega`:

   Decoding Genes. `xega` provides two hot spots for configuring the decoding process of a gene, namely `xega::sgXDecodeGeneFactory()` and `xega::sgXGeneMapFactory()`. Only binary-coded genetic algorithms (`"sga"`) and grammatical evolution algorithms (`"sge"`) (which also use binary-coded genes) provide a gene map factory with several configurable options.

   Initializing Genes. `xega::sgXInitGeneFactory()` configures for each gene-representation the proper gene initialization function.

   Replicating Genes. The gene replication process for genetic algorithms (`"sga"`, `"sgperm"`, `"sgp"`, and `"sge"`) provides variants which produce one or two kids which receive genetic material from one parent (mutation only) or two parents (crossover). The two kid variant has the property that crossover operations do not lose genetic information in a population of genes, whereas the one kid variants may lose genetic information. For some evolutionary algorithms, the biological analogy

with sexual reproduction by two parents fails: The number of genes used by differential evolution (as an example of an evolutionary algorithm) is three. Other derivative-free methods like the Nelder-Mead simplex algorithm use an even higher number of parents (Conn et al, 2009). A rich set of biological analogies for such replication processes can be found in the area of viral replication processes (Lostroh, 2019). `xega::sgXReplicationFactory()` configures this process.
Mutation operators generate a random neighbor of a parent gene. Random generation of a neighbor of a gene strongly depends on the underlying gene representation and, usually, for each gene representation several mutation operations have been suggested in the literature. Therefore, mutation operators are configured in a two step process: The `algorithm` argument of `xega::sgXMutationFactory()` chooses a representation dependent function factory, the `method` argument chooses the concrete implementation of the mutation operator from the selected function factory.
Crossover operators randomly mix the genetic information of two parent genes. The configuration process of `xega::sgXCrossoverFactory()` is the same as for mutation operators.

2. Hot spots specific to a class of algorithms. For `xega` (Version 0.9.0.0) only one hot spot exists which implements 2 variants of setting a scale factor in differential evolution algorithms (see Sharma et al (2019)) by the function factory `xegaDfGene::xegaDfScaleFactory()`.

Grammar-based genetic programming algorithms as well as grammatical evolution algorithms are genetic programming systems. The compilation of the BNF (Backus-Naur Form) of a grammar (by `xegaBNF::compileBNF()`) shown in red in Fig. 7 is independent of `xegaRun()`. This step is implemented by the R-package `xegaBNF`.

However, the two genetic programing systems differ considerably with regard to their internal workings:

- Grammatical evolution essentially requires only a decoder which translates a random binary-coded gene into a random program conforming to the grammar specified. (The program may not be complete in the sense that some non-terminal symbols are part of the program). For the genetic machinery, the operations for binary-coded genes from the R-package `xegaGaGene` are reused. The R-package `xegaGeGene` implements such a decoder by

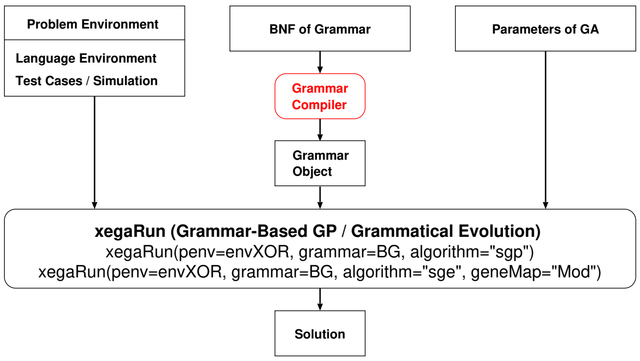
Problem Environment
Language Environment
Test Cases / Simulation
BNF of Grammar
Grammar Compiler
Grammar Object
Parameters of GA
xegaRun (Grammar-Based GP / Grammatical Evolution)
xegaRun(penv=envXOR, grammar=BG, algorithm="sgp")
xegaRun(penv=envXOR, grammar=BG, algorithm="sge", geneMap="Mod")
Solution

Fig. 7 Genetic Programming Systems in `xegaRun`

using the following 2 functions on derivation trees from the R-package `xegaDerivationTrees`:

1. `xegaDerivationTrees::generateDerivationTrees()`
2. `xegaDerivationTrees::decodeDT()`

- Grammar-based genetic programming, on the other hand, operates on derivation tree genes: The complete genetic machinery is based on the random operations on derivation trees which are implemented in the R-package `xegaDerivationTrees`.

Grammatical evolution and grammar-based genetic programming provide two fundamentally different solutions how evolutionary search progresses through the set of computer programs specified by the grammar. For a short discussion, see Geyer-Schulz (2021).

4.3.3 Representation independent and performance related hot spots

Two representation independent hot spots provide performance related functionality:

1. `xegaSelectGene::evalGeneFactory()` and
2. `xegaPopulation::ApplyFactory()`.

First, `xegaSelectGene::evalGeneFactory()` delivers evaluation related functionality which is configured by the `xega::xegaRun` option evalmethod. All evaluation functions implement the following abstract interface:

```
<gene> <- <evaluation function name>(gene=<gene>, lF=<local
    function list>)
```

xegaSelectGene::evalGeneFactory() offers 4 different methods of the evaluation of fitness functions:

1. The default method EvalGeneU() is to evaluate the gene by decoding it and calling the fitness function of the problem environment in the local function lF$penv$f(). Evaluation errors of the fitness function are handled by setting the fitness value to the minimum fitness of the previous population and to set the $evalFail flag to TRUE. Every gene in the population is evaluated in each generation.
2. If the fitness function is deterministic, the method EvalGeneDet() uses the strategy of testing the $evaluated flag of the gene. Only genes which have not been evaluated, are evaluated. This implements a modified version of the memoization design pattern which uses the population of genes instead of look-up table for function values. Compared with the standard pattern for memo functions (e.g. Michie (1968) and Geyer-Schulz (1989)), the following differences exist:

   - The additional memory requirements are lower (one flag per gene).
   - No lookup operations on the population are performed. Genes with the same genotype are evaluated repeatedly.
   - A gene is evaluated only once in its life-time. This means that the gene initialization operation, the crossover operation, and the mutation operation set the flag $evaluated to FALSE and the gene evaluation operation sets the flag $evaluated to TRUE.

   Therefore, the performance gain depends on the percentage of genes which are copied unchanged (without mutation or crossover) to the next generation. The assumption made in this approach is that gene lookup and comparison operations are too time consuming and expensive to implement for complex gene representations (e.g. derivation trees).
3. If the fitness is a random variable, the function EvalGeneStoch() reevaluates unchanged genes and computes incrementally the expected value (mean) and the variance of the random value with the update for the mean $\overline{f}_{i+1} = \frac{i}{i+1}\overline{f}_i + \frac{1}{i+1} f_i(x)$ and the update for the variance $\sigma_{i+1}^2 = \frac{1}{i+1}(\sigma_i^2 + (f_i(x) - \overline{f}_i) * (f_i(x) - \overline{f}_{i+1}))$. With the number of evaluations i, the mean

fitness value approximates the expected value of the random variable fitness. The `$sigma` list element indicates the standard deviation of the random variable `$fit` with $\sigma_i = \sqrt{\sigma_i^2}$. In the current version of `xega`, all selection and scaling operators use the mean of the random variable fitness as fitness value and ignore the standard deviation.
4. The function `EvalGeneR()` allows a decoder to repair a gene. This may be used by grammar-based genetic programming and grammatical evolution.

Second, the current basic approach to parallelization in `xega` version 0.9.0.0 is to configure the evaluation loop over a population of genes in `xegaPopulation::xegaEvalPopulation()` with an implementation of the desired execution model. Use of an execution model requires the following abstract three step process:

Startup. Define and setup of the infrastructure needed.
Evaluation Loop. Distribute tasks, execute tasks, collect results.
Finalize. Shutdown the infrastructure.

Depending on the execution model, the *Startup* and the *Finalize* step can be omitted or must be partially implemented outside `xega::xegaRun()` and/or in scripts outside of R (e.g. on HPC clusters with the SLURM workload manager). From an architectural point of view, it is recommended to keep the *Startup* and *Finalize* code out of `xega::xegaRun` and to provide a hot spot for injecting the actual implementation of the *Evaluation Loop*. This guarantees, that an end-user can configure the parallel backend which fits to the available computing infrastructure.

Efficiently executing a set of tasks in parallel on a computing infrastructure requires proper configuration of the load balancing strategy used: Ideally, all nodes of the computing infrastructure need the same time to execute the set of tasks assigned to them. Population-based genetic and evolutionary algorithms execute one evaluation loop per generation. The run-time of an algorithm is the sum of the maxima of the execution times of the task sets assigned to the nodes of the computing infrastructure for each generation. We distinguish two main situations:

- The run-time of the evaluation of the fitness of a gene is almost the same for all genes and it is almost the same on all computing nodes of the infrastructure. (Homogeneous).

- The run-time of the evaluation of the fitness of a gene varies considerably for genes or it varies considerably on different computing nodes of the infrastructure. (Heterogeneous).

`xegaPopulation::ApplyFactory()` is the hot spot for selecting the implementation of the execute loop step for the execution models directly supported by by the `xega` family of R-packages. In addition, injection of a user-defined parallel apply function is possible by calling `xega::xegaRun()` with `uParApply=UserDefinedApplyFunction`. The second approach is used when using the `Rmpi` package. All (parallel) apply functions have the following signature:

```
<pop> <- <apply function name> (
                pop=<list of genes>,
                EvalGene=<function for evaluation of a gene>,
                lF=<local function list>)
```

`xegaPopulation::ApplyFactory()` currently allows selection of the execute loops of the following execution models:

1. `Sequential` (Default): Uses `base::lapply()`.
2. `MultiCore` and `MultiCoreHet` (`xega` version 0.9.0.1):
   Both use `parallel::mclapply()`, the second is for tasks with a high variability of execution time. The computing infrastructure used are the cores of the local machine. The *Startup* and *Finalize* steps need no additional code. For MultiCore, the tasks are split into as many parts as there are cores, and then one process is forked on each core and the results are collected via socket connections. For MultiCoreHet, a separate job is forked for each task. As soon as the number of jobs running equals the number of cores, the master process waits for a child process to finish before the next fork.
3. `Cluster` (for homogeneous tasks) and `ClusterHet` (for heterogeneous tasks, `xega` version 0.9.0.1): Use `parallel::parLapply()` and `parallel::parLapplyLB()`, respectively. The `parallel` package uses socket connections. The computing infrastructure used is a local area network which is specified by a list of host names:

   a. *Startup*: Set up a cluster object:

   ```
   wcl<-parallely::makeClusterPSOCK(workers=
   'a.iism.kit.edu', 'b.iism.kit.edu', 'c.iism.kit.edu')
   ```

   b. *Finalize*: Set up an exit handler:

```
on.exit(parallel::stopCluster(wcl))
```

c. *Evaluation Loop*: The hot spot for the evaluation loop is embedded in the function `xega::xegaRun()` and specified with `executionModel="Cluster"` or `executionModel="ClusterHet"`.

The code for the *Startup*, *Finalize*, and *Evaluation Loop* step must be embedded into a function.
Note, that the definition of the cluster as well as the specification of an event handler for shutting down the cluster is done outside of the `xega` package.

4. `FutureApply` and `FutureApplyHet` (for heterogeneous tasks, `xega` version 0.9.0.1): Use `future.apply::future_lapply()`.
H. Bengtsson's futureverse packages empower the end-user to have full control on how and where to parallelize (Bengtsson (2024) and Bengtsson (2021)). `future.apply::future_lapply()` can be resolved by using any future-supported backend. All future backends are built on the Future API which provides an implementation of futures. A future is an abstraction for a value that may be available at some time in the future. The value has two states, namely resolved or unresolved. If an unresolved value is accessed, the current process is blocked until the future is resolved. How the value of a future is resolved depends on the evaluation strategy. For example, a future may be resolved asynchronously by evaluating the future expression concurrently on a compute cluster. The execution model of future backends consists of the steps *Startup*, *Evaluation loop*, and *Finalize*. The *finalize* step may be omitted. The `callr` future backend communicates via files and is not subject to constraints in the number of parallel processes. On a notebook with multiple cores, the future backend and the number of cores (called workers) have to be specified as arguments of the `future::plan()` function:

a. *Startup*:

```
plan(callr, workers=8)
```

b. *Evaluation Loop*: The hot spot for the evaluation loop is embedded in the function `xega::xegaRun()` and specified with `executionModel="FutureApply"` or `executionModel="FutureApplyHet"`.

Using `xega` with the `Rmpi` package (Yu, 2002) is possible by code injection:

- Define a function with a parallel evaluation loop (`mpi.parLapply()`) from the package `Rmpi` with the signature of a (parallel) apply function:

```
rmpiLapply<-function(pop, EvalGene, lF)
{ Rmpi::mpi.parLapply(pop, FUN=EvalGene, lF=lF) }
```

- Second, use the argument `uParApply=rmpiLapply` to inject the code of the `rmpiLapply()` function into `xega::xegaRun()` and to override the argument `executionModel`.

The reason for injecting `rmpiLapply` with `uParApply` is the need to decouple `xega` from middleware packages for parallel and distributed computing which may not be available or which may not be easily installable at cran sites (or by the end-user). Integrating `rmpiapply` into the R-package `xegaPopulation` establishes a dependency from the R-package `Rmpi`. This is problematic, because `Rmpi` is not available at all cran-sites.

For some code examples of parallelizing `xega` with the execution models described above, see

https://github.com/ageyerschulz/xega/tree/main/examples/executionModel/

Two well-known limitations (mainly for end-users) are:

- `MultiCore` and `MultiCoreHet` rely on forking a process and are not available under the Windows operating system, because Windows does not support forking. This implies that `parallel::mclapply()` does not work on Windows for more two or more cores.
- The number of simultaneous IP socket connections is restricted to 125. This means that at most 125 parallel processes are available for `MultiCore`, `MultiCoreHet`, `Cluster`, and `ClusterHet`. Users needing more than 125 workers can use e.g. either `FutureApply` or `FutureApplyHet` with `future::plan(callr)` or `Rmpi`. A complete example of using `Rmpi` is in the subdirectory `mpi`.

`xega` (Version 0.9.0.17) implements the compilation of abstract genetic operator pipelines (Geyer-Schulz and Zamani Shandiz, 2025). Compilation shifts the computation of large parts of the genetic machinery, namely crossover, mutation, and acceptance rules, into the gene evaluation process. The effect is that more than 95% of the execution time of the sequential execution of `xegaRun` is spent in the evaluation phase which can be parallelized. Only fitness scaling and gene selection are executed sequentially. The following code example optimizes a small 15-city TSP in parallel:

```
library(xega)
j<-xegaRun(penv=lau15, max=FALSE, algorithm="sgperm",
   popsize=20, generations=5, max2opt=20,
   genemap="Identity",  mutation="MutateGeneMix",
   executionModel="MultiCore", replication="Kid1Pipeline",
       pipeline=TRUE)
```

4.4 *Other Performance Improvements: Eliminating Recomputation in Selection Functions*

The function `xegaSelectGene::TransformSelect()` precomputes the indices of genes to be selected and converts the selection function into a function which accesses and returns the next index when called. Recomputation of expressions used in selection functions is avoided. All transformed selection functions are index lookup functions which use the same code.

The memory needed by the transformed function increases linearly with population size. The complexity of computation shifts from $n \cdot f(n)$ to $f(n) + n.c$ where $f(n)$ is the time complexity of computing the selection function $f(n)$ once for a population of size n and c is the constant cost of performing an index operation. Table 2 illustrates the time effects of this transformation.

Population Size	100		1000		10000	
	$\bar{x}$	$\sigma(x)$	$\bar{x}$	$\sigma(x)$	$\bar{x}$	$\sigma(x)$
Selection Functions						
Uniform	0.000440	0.000012	0.004136	0.000062	0.046940	0.003248
Duel	0.000927	0.000015	0.009363	0.000094	0.103515	0.006724
LRSelective	0.003176	0.000060	0.039912	0.000735	1.212198	0.013774
ProportionalOnln	0.003557	0.000078	0.126240	0.005169	9.951252	0.114069
PropFitDiffOnln	0.003674	0.000084	0.129804	0.003774	10.271100	0.084511
SUS	0.004002	0.000066	0.320285	0.009316	29.408860	0.131657
LRTSR	0.010622	0.000114	0.415450	0.010621	32.552430	0.113895
Transformed Selection Functions						
Uniform C	0.000485	0.000009	0.004393	0.000157	0.048173	0.000703
Duel C	0.000506	0.000013	0.004382	0.000036	0.049263	0.003203
LRSelective C	0.000563	0.000012	0.004521	0.000065	0.050030	0.005200
ProportionalOnln C	0.000580	0.000010	0.004730	0.000039	0.051614	0.000599
PropFitDiffOnln C	0.000626	0.000019	0.004743	0.000059	0.051689	0.003176
SUS C	0.000545	0.000015	0.004515	0.000055	0.049819	0.003214
LRTSR C	0.000688	0.000016	0.004691	0.000036	0.050053	0.000383

Table 2 Benchmark of Selection Functions. 50 Trials. (Time in seconds)

The first block of results in Table 2 shows that all selection functions (except uniform and duel) have a non-linear time complexity because of repeated recomputations whereas the second block of results has almost linear time complexity. For instance, for stochastic universal sampling (SUS), the transformed function is in the mean 590 times faster than the original implementation. For all forms of uniform and tournament selection, the transformation is slightly slower and should be turned off (option `selectionContinuation = FALSE` for `xegaRun`).

The benchmark shown in Table 2 has been computed in sequential mode on a Thinkpad notebook with an AMD Ryzen 7 7730U with Radeon Graphics chip set from Advanced Micro Devices (AMD) with 8 physical cores with 16 threads (virtual cores) under x86_64 GNU/Linux (Fedora release 41 (Forty One)) with R version 4.5.1 (2025-06-13) R Core Team (2025) with R's default random number generator Mersenne-Twister without seed. Times are measured as differences of R's `Sys.time()` function (on Linux systems microsecond accuracy, on Windows millisecond accuracy). A single trial version of the benchmark is part of the R-package `xegaSelectGene` (Geyer-Schulz, 2024j) and can be repeated (but not reproduced) by the following R code:

```
library(xegaSelectGene)
options(scipen=999)
options(width=100)
runSelectBenchmarks(c(100, 1000, 10000))
```

5 Summary and Future Extensions

The `xegaX` family of R-packages is based on the architecture first presented by Geyer-Schulz (2021) at the DSSV-ECDA 2021 conference in section 3. The design of this architecture has the goals of being highly configurable, extendible, scalable, and efficient. All packages of the `xegaX` family of R-packages are completely written in the R language and their complete implementation is available from CRAN. Therefore, the cross-platform compatibility of the `xegaX` family of R-packages corresponds to that of R. In section 4.1. the concept of hot spots for the separation of a framework and its changeable parts is introduced and a tutorial example of the implementation of function containers and function factories is shown which together allow a direct implementation of the reflection pattern. In section 4.2 the consequences of representation dependence for the configuration of `xega::xegaRun()` are shown and the advantages of this architecture are highlighted. Last, but not least, in section 4.3 a snapshot of the hot spots configurable in `xega` Version 0.9.0.0 as of 2024-03-20 (and of its imports from the other packages of the `xegaX` family of R-packages) is presented.

Repeatability and reproducibility of algorithms in `xega` is complex:

- For sequential execution, exact repeatability is guaranteed when using fixed seeds (option `replay` of `xegaRun()` > 0) with regard to fitness, but not with regard to execution time: Execution times may vary. `xegaReRun()` supports the repetition of experiments with exactly the same configuration.
- Transformation of R-functions and/or parallel execution may change the order in which random numbers are generated and used. Therefore, exact repeatability can not by guaranteed across function transformations (transformation of selection functions and compilation of abstract genetic operator pipelines) and not across different execution (and evaluation) models.
- First experiments confirm stochastic reproducibility of fitness and execution times. However, due to the large configuration space of `xegaRun()` this is almost impossible to test experimentally.

The following list of future extensions of the `xegaX` family of R-packages reflects a few of the author's wishes:

1. Vector-valued fitness functions and their underlying mechanisms. These extensions allow the handling of multi-objective optimization problems. For example, hyperparameter-tuning of algorithms requires balancing of the computational resources used with the quality of solutions reached. One candidate algorithm is the NSGA-II algorithm of Deb et al (2002).
2. Adding a hot spot and a library of standard termination conditions. `xegaRun` has the capability to handle user-defined termination conditions in the problem environment. However, several standard applications exist for which standard termination conditions exist which can be reused across large sets of problem environments. For example, training meta-genetic algorithms on large sets of benchmark functions with known optima as provided by the R-package `smoof` (Bossek, 2017).
3. Introduction of variants of derivative-free optimization algorithms as e.g. Schwefel's evolutionary algorithms (Bäck and Schwefel, 1993) or the Nelder-Mead simplex algorithm (Conn et al, 2009).
4. Variants of adaptive algorithms: Meta-genetic algorithms, self-adaptive algorithms, adaptive grammars. For additional ideas, see Aleti and Moser (2016).

5. Improvements in the area of grammar-based genetic programming and grammatical evolution with regard to efficient sampling algorithms for programs from grammars (Böhm and Geyer-Schulz, 1997), evolutionary grammar design, as well as improved domain-specific genetic operators. See e.g. Geyer-Schulz (1997).
6. Multiploid chromosomes with different representations and their configuration.

Acknowledgements The author acknowledges support by the state of Baden-Württemberg through bwHPC.

References

Aleti A, Moser I (2016) A systematic literature review of adaptive parameter control methods for evolutionary algorithms. ACM Computing Surveys 49(3(56)):1 – 35, DOI 10.1145/2996355

Bäck T, Schwefel HP (1993) An overview of evolutionary algorithms for parameter optimization. Evolutionary Computation 1(1):1 – 23, DOI 10.1162/evco.1993.1.1.1

Backus J (1978) Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. Communications of the ACM 21(8):613 – 641, DOI 10.1145/359576.359579

Baker JE (1987) Reducing bias and inefficiency in the selection algorithm. In: Grefenstette JJ (ed) Genetic Algorithms and their Applications (ICGA'87). Proceedings of the Second International Conference on Genetic Algorithms, MIT, Lawrence Erlbaum Associates, pp 14 – 21

Bengtsson H (2021) A unifying framework for parallel and distributed processing in R using futures. The R Journal 13(2):208 – 227, DOI 10.32614/RJ-2021-048

Bengtsson H (2024) future.apply: Apply function to elements in parallel using futures. Tech. Rep. 2024-09, University of California, San Francisco, San Francisco, URL `https://CRAN.R-project.org/package=future.apply`

Böhm W, Geyer-Schulz A (1997) Exact uniform initialization for genetic programming. In: Belew RK, Vose M (eds) Foundations of Genetic Algorithms 4, Morgan Kaufmann Publishers, San Francisco, chap 19, pp 379 – 407

Bossek J (2017) smoof: Single- and multi-objective optimization test functions. The R Journal 9(1):103 – 113

Buschmann F, Meunier R, Rohnert H, Sommerlad P, Stal M (1996a) Layers. In: Pattern-Oriented Software Architecture. A System of Patterns (Vol.1), vol 1, John Wiley, Chichester, chap 2.2.1, pp 31 – 52

Buschmann F, Meunier R, Rohnert H, Sommerlad P, Stal M (1996b) Pattern-Oriented Software Architecture. A System of Patterns (Vol.1), vol 1. John Wiley, Chichester

Buschmann F, Meunier R, Rohnert H, Sommerlad P, Stal M (1996c) Reflection. In: Pattern-Oriented Software Architecture. A System of Patterns, vol 1, John Wiley, Chichester, chap 2.5.2, pp 193 – 220

Conn AR, Scheinberg K, Vicente L (2009) Introduction to Derivative-Free Optimization. SIAM, Philadelphia

Croes GA (1958) A method for solving traveling-salesman problems. Operations Research 6(6):791 – 812

Deb K, Pratap A, Agarwal S, Meyarivan T (2002) A fast and elitist multiobjective genetic algorithm: NSGA-II. IEEE Transactions on Evolutionary Computation 6(2):182 – 197, DOI 10.1109/4235.996017

Foster I, Kesselmann C (1999) The Grid. Blueprint for a New Computing Infrastructure: Blueprint for a New Computing Infrastructure. Morgan Kaufmann, San Francisco

Gamma E, Helm R, Johnson R, Vlissides J (1995a) Abstract factory (87). In: Design Patterns. Elements of Reusable Object-Oriented Software, Addison-Wesley Professional Computing Series, Addison-Wesley Publishing Company, Reading, chap 3.1, pp 87 – 95

Gamma E, Helm R, Johnson R, Vlissides J (1995b) Factory method (107). In: Design Patterns. Elements of Reusable Object-Oriented Software, Addison-Wesley Professional Computing Series, Addison-Wesley Publishing Company, Reading, chap 3.3, pp 107 – 116

Geyer-Schulz A (1989) Memo. SIGAPL APL Quote Quad 20(2):12 – 27, DOI 10.1145/379209.379211

Geyer-Schulz A (1997) Fuzzy Rule-Based Expert Systems and Genetic Machine Learning, 2nd edn. Studies in Fuzziness and Soft Computing, Physica-Verlag, Heidelberg

Geyer-Schulz A (2021) Architectural design of a unified GA/GP-package for R. Talk at DSSV-ECDA 2021.

Geyer-Schulz A (2024a) xega: Extended evolutionary and genetic algorithms. Tech. Rep. 2024-10, IISM, KIT, Karlsruhe, URL `https://CRAN.R-project.org/package=xega`

Geyer-Schulz A (2024b) xegaBNF: Compile a Backus-Naur form specification into an R grammar object. Tech. Rep. 2024-02, IISM, KIT, Karlsruhe, URL `https://CRAN.R-project.org/package=xegaBNF`

Geyer-Schulz A (2024c) xegaDerivationTrees: Generating and manipulating derivation trees. Tech. Rep. 2024-03, IISM, KIT, Karlsruhe, URL `https://CRAN.R-project.org/package=xegaDerivationTrees`

Geyer-Schulz A (2024d) xegaDfGene: Gene operations for real-coded genes. Tech. Rep. 2024-05, IISM, KIT, Karlsruhe, URL `https://CRAN.R-project.org/package=xegaDfGene`

Geyer-Schulz A (2024e) xegaGaGene: Binary gene operations for genetic algorithms. Tech. Rep. 2024-04, IISM, KIT, Karlsruhe, URL `https://CRAN.R-project.org/package=xegaGaGene`

Geyer-Schulz A (2024f) xegaGeGene: Grammatical evolution. Tech. Rep. 2024-08, IISM, KIT, Karlsruhe, URL `https://CRAN.R-project.org/package=xegaGeGene`

Geyer-Schulz A (2024g) xegaGpGene: Genetic operations for grammar-based genetic programming. Tech. Rep. 2024-07, IISM, KIT, Karlsruhe, URL `https://CRAN.R-project.org/package=xegaGpGene`

Geyer-Schulz A (2024h) xegaPermGene: Operations on permutation genes. Tech. Rep. 2024-06, IISM, KIT, Karlsruhe, URL `https://CRAN.R-project.org/package=xegaPermGene`

Geyer-Schulz A (2024i) xegaPopulation: Genetic population level functions. Tech. Rep. 2024-09, IISM, KIT, Karlsruhe, URL `https://CRAN.R-project.org/package=xegaPopulation`

Geyer-Schulz A (2024j) xegaSelectGene: Selection of genes and gene representation independent functions. Tech. Rep. 2024-01, IISM, KIT, Karlsruhe, URL `https://CRAN.R-project.org/package=xegaSelectGene`

Geyer-Schulz A, Zamani Shandiz M (2025) Compiling abstract genetic operator pipelines in the R-package xega. Tech. Rep. 2025-10-1, Information Services and Electronic Markets, Institute of Customer Insights, Karlsruhe Institute of Techology (KIT), Karlsruhe, submitted to Evolutionary Computation

Goldberg DE (1989) Genetic Algorithms in Search, Optimization and Machine Learning, 1st edn. Addison-Wesley, Reading

Grefenstette JJ, Baker JE (1989) How genetic algorithms work: A critical look at implicit parallelism. In: Schaffer JD (ed) Proceedings of the Third International Conference on Genetic Algorithms, George Mason University, Morgan Kaufmann Publishers, Los Altos, pp 20 – 27

Lange R, Schaul T, Chen Y, Lu C, Zahavy T, Dalibard V, Flennerhag S (2023a) Discovering attention-based genetic algorithms via meta-black-box optimization. In: Proceedings of the Genetic and Evolutionary Computation Conference, Association for Computing Machinery, New York, NY, USA, GECCO ’23, pp 929 – 937, DOI 10.1145/3583131.3590496

Lange RT, Schaul T, Chen Y, Lu C, Zahavy T, Dalibard V, Flennerhag S (2023b) Discovering attention-based genetic algorithms via meta-black-box optimization. arXiv 2023:1 – 14

Lin S, Kernighan BW (1973) An effective heuristic algorithm for the traveling-salesman problem. Operations Research 21(2):498 – 516, DOI 10.1287/opre.21.2.498

Locatelli M (2000) Convergence of a simulated annealing algorithm for continuous global optimization. Journal of Global Optimization 18:219 – 233, DOI 10.1023/A:1008339019740

Lostroh P (2019) Molecular and Cellular Biology of Viruses. CRC Press, Boca Raton

Martin R (1995) Discovering patterns in existing applications. In: Coplien J, Schmidt D (eds) Pattern Languages of Program Design 1, Addison-Wesley, Reading, Massachusetts, Pattern Languages of Program Design, vol 1, pp 365 – 393

Metropolis N, Rosenbluth AW, Rosenbluth MN, Teller AH, Teller E (1953) Equation of state calculations by fast computing machines. Journal of Chemical Physics 21(6):1087 – 1092, DOI 10.1063/1.1699114

Michie D (1968) Memo functions and machine learning. Nature 218(6):19 – 22

O’Neil M, Ryan C (2003) Grammatical Evolution: Evolutionary Automatic Programming in an Arbitrary Language. Kluwer, Dordrecht

Price KV, Storn RM, Lampinen JA (2005) Differential Evolution. A Practical Approach to Global Optimization. Springer, Berlin, DOI 10.1007/3-540-31306-0

R Core Team (2025) R: A Language and Environment for Statistical Computing. R Foundation for Statistical Computing, Vienna, Austria, URL `https://www.R-project.org/`

Resende MGC, Ribeiro CC (2016) Optimization by GRASP: Greedy Randomized Adaptive Search Procedures. Springer New York, New York, NY

Rising L (2000a) The Pattern Almanac 2000. The Software Pattern Series, Addison-Wesley, Boston

Rising L (2000b) Pluggable factory. In: The Pattern Almanac 2000, The Software Pattern Series, Addison-Wesley, Boston, chap 132, p 132

Rothlauf F (2006) Representations for Genetic and Evolutionary Algorithms. Springer, Heidelberg

Ryan C, O'Neill M, Collins J (eds) (2018) Handbook of Grammatical Evolution, vol 47. Springer International Publishing, Cham, DOI 10.1007/978-3-319-78717-6

Sharma P, Sharma H, Kumar S, Bansal JC (2019) A review on scale factor strategies in differential evolution algorithm. In: Bansal JC, Das KN, Nagar A, Deep K, Ojha AK (eds) Soft Computing for Problem Solving, Advances in Intelligent Systems and Computing, vol 817, Springer Singapore, Singapore, pp 925 – 943

Stanhope SA, Daida JM (1996) An individually variable mutation-rate strategy for genetic algorithms. In: Koza JR (ed) Late Breaking Papers at the Genetic Programming 1996 Conference, Stanford University Bookstore, Stanford, pp 177 – 185

Swan J, Adriaensen S, Barwell AD, Hammond K, White DR (2019) Extending the "open-closed principle" to automated algorithm configuration. Evol Comput 27(1):173 – 193, DOI 10.1162/evco_a_00245

Syswerda G (1991) Schedule optimization using genetic algorithms. In: Davis L (ed) Handbook of Genetic Algorithms, Van Nostrand Reinhold, New York, chap 21, pp 332 – 349

Van Laarhoven PJM, Aarts EHL (1992) Simulated Annealing. Theory and Applications. Springer, Dordrecht, DOI 10.1007/978-94-015-7744-1

Whitley D (1989) The GENITOR algorithm and selection pressure: Why rank-based allocation of reproductive trials is best. In: Schaffer JD (ed) Proceedings of the Third International Conference on Genetic Algorithms, George Mason University, Morgan Kaufmann Publishers, Los Altos, pp 116 – 121

Wickham H (2019) Function factories. In: Advanced R (2nd), The R Series, CRC Press, Boca Raton, chap 10, pp 292 – 330

Yu H (2002) Rmpi. Parallel statistical computing in R. R News 2(2):10 – 14