REIHE COMPUTATIONAL INTELLIGENCE

COLLABORATIVE RESEARCH CENTER 531

Design and Management of Complex Technical Processes and Systems by means of Computational Intelligence Methods

Ease
Evolutionary Algorithms Scripting Environment

Joachim Sprave

No. CI-54/99

Technical Report ISSN 1433-3325 January 1999

Secretary of the SFB 531 · University of Dortmund · Dept. of Computer Science/XI
44221 Dortmund · Germany

This work is a product of the Collaborative Research Center 531, "Computational Intelligence", at the University of Dortmund and was printed with financial support of the Deutsche Forschungsgemeinschaft.

Contents

1 Introduction

Ease – Evolutionary Algorithms Scripting Environment – is an extension to the Tcl scripting language, providing commands to create, modify, and evaluate populations of individuals represented by real number vectors and/or bit strings.

It is assumed that the reader of this document is familiar with the basics of the Tcl scripting language. Furthermore, basic knowledge of the C programming language is required to implement user-defined fitness functions. General concepts of Evolutionary Algorithms (EAs) are beyond the scope of this document, as well. For a general overview of EAs, the reader is referred to [1].

In the current state, Ease does not cover any of the traditional Evolutionary Algorithms (EAs) entirely – it is missing correlated mutations from Evolution Strategies (ES), proportional selection from Genetic Algorithms (GA), the special tournament selection used in Evolutionary Programming (EP), and much more. It does provide real valued vectors and bit strings, both separately and in combination, ES-style step size adaptation, tournament selection and linear ranking as well as (μ, λ) and $(\mu + \lambda)$ truncation selection.

In the first place, Ease was designed to analyze non-panmictic population structures. As a consequence, it offers a simple method to address subsets of a population. Individuals are identified by their indices, and a subpopulation is formed by a list of indices. Hence, the operators provided by Ease can be restricted to any subpopulation by using index lists.

2 Getting Started

To enable the EA commands, you have to load the Ease module at the beginning of your Tcl/Ease script:

```
load lib/Ease.so
```

Ease.so contains the commands to manage populations, but no evaluation function. This should be loaded before the first evaluations are performed:

```
load lib/Func_sphere.so
```

The creation of user defined evaluation function is quite easy, as described in section 8. Ease actually defines only two commands: `population` and `iset`. The latter one allows an elegant way to address subsets of a population and will be described later in this document. The `population` command creates a new population named by the string following the command. Two parameters must be specified: the number of real variables and the number of bits of the representation used in the objective function. Example:

```
population mypop -vars 30 -bits 0
```

This creates a population of the default number of 50 individuals, each having 30 real parameters and no binary parameter. Furthermore, a new command is created: `mypop`. This command is an object command similar to widget commands in Tk. Consequently, you have to use the newly created `mypop` command in combination with a subcommand to manipulate your population. The following line evaluates the first 3 individuals, addressed by their indices, of the population `mypop`:

```
mypop evaluate {0 1 2}
```

If type this interactively into the Tcl interpreter, it responds a 3, which means that 3 individuals have been evaluated. Repeating the same command returns a 0, because Ease knows that the given individuals have not been altered since the last evaluation. If your fitness function is noisy, you can force the evaluation by adding `-force` at the end of the command.

Offspring is created by the `reproduce` subcommand. It accepts two lists, a children index list and a parents index list. For each child, two parents are selected from the parent list. The parents are recombined into one new individual which is stored in the child's place. In the following example, a single child is created by selecting 2 parents from the second list using tournament selection with a tournament size of 3. The result is stored at index 0 of the population:

```
mypop reproduce {0} {4 5 6 7 8 9} -select tournament -tournsize 3
```

In the next example, the same procedure is done four times:

```
mypop reproduce {0 1 2 3} {4 5 6 7 8 9} -select tournament -tournsize 3
```

Instead of reproduction, mutation works in-place:

```
mypop mutate {0 1 2 3} -mrate 0.01
```

This mutates the individuals residing in the first 4 positions of the population using a mutation rate of 0.01.

The parameters of the subcommands can be set in four ways:

- By default. Most parameters are set to default values if they are not set explicitly.
- As options of the `population` command. The parameters set here are used as long as the population exists.
- As options of the `configure` subcommand. Parameter values set by the `configure` subcommand replace those set by the `population` command, as well as settings made by earlier `configure` calls.
- As option to a subcommand. Options of subcommands override parameter values set by the `population` command or `configure` subcommand temporarily.

There are only three parameters of the `population` command which cannot be altered: `size`, `vars`, and `bits`.

3 Index Sets

Individuals in Ease are addressed by their indices in the population. All Ease commands which accept a list of individuals also understand the following shortcuts (assuming a population size of 8):

Syntax	Description	Example	Expands to
`a:b`	Range from a to b incl.	`0:3`	`0 1 2 3`
`a<n>`	n individuals starting from a	`4<3>`	`4 5 6`
`all`	The entire population	`all`	`0 1 2 3 4 5 6 7`

To allow ranges to be assigned to variables conveniently, the `iset` command is provided and performs the shortcut expansion shown above:

```
set mu 15
set lambda 100
population mypop -vars 10 -bits 0 -size [expr $mu + $lambda]
set parents [iset 0<$mu>]
set children [iset $mu<$lambda>]
```

The `iset` command command also works as a union operator. To select the μ survivors from both the parents and the children in a $(\mu + \lambda)$ Evolution Strategy, just write

```
mypop copy $survivors [mypop best [iset $parents $children] $mu]
```

Note that `iset` does not check for multiple instances of an index.

4 Population Commands

4.1 `configure`

This command can change most of the population parameters. Changes persist until they are overridden by another `configure` call.

Example: Change the mutation rate to 0.02:

```
mypop configure -mrate 0.02
```

4.2 stat

This command accepts a list of indices and returns the following information about the corresponding individuals (rv: real-valued representation only. If not used, values are set to zero. bs: bit string representation only).

- Lowest mutation step size (rv)
- Highest mutation step size (rv)
- Feasibility of the best individual
- Fitness value of the best individual
- Feasibility of the worst individual
- Fitness value of the worst individual
- Bias (bs)
- Number of converged bits (bs)

Example: Calculate some statistics about the first 4 individuals in the population **`mypop`** and write it to the standard output channel:

```
puts stdout [mypop stat {0 1 2 3}]
```

4.3 age

The **`age`** command increases the age of the given individuals by one. The age is automatically set to zero by initialization, mutation, or recombination.

Example: Increase the age of some individuals:

```
mypop age {3 2 5 10}
```

4.4 mutate

The **`mutate`** command mutates the real-valued vectors and bits strings of the given individuals according to the configuration of the current population. The settings can be temporarily overidden by appending the options to the mutate command.

Example: Mutate all individuals using the configured settings. Mutate the first 4 individuals once more using a mutation rate of 0.1 for bit strings and $\tau_0 = 0.5$ for step sizes of real-vallued parameters:

```
mypop mutate all
mypop mutate {0 1 2 3} -mrate 0.1 -tau0 0.5
```

4.5 reproduce

The actual reproduction including optional recombination is performed by the reproduce operator. It take 2 lists of individuals: The first list contains the target individuals, i.e. the populations indices to store the results of the generation process to. The second list contains the possible parents. For each position in the target list, one or two parents are selected from the source list. The parent, or a recombination of the parents, respectively, replaces the individual in the target position. Selection from the possible parent list can be done uniformly, by tournaments, or mating (best 2).

4.6 evaluate

This command evaluates the given individuals using the most recently load objective function.

4.7 best, worst

These commands take 2 parameters: a list of indices, and the number of best/worst individuals to return. If the option `-sort` is present, the result list is sorted according to the given criterion.

4.8 copy

The copy command expects two lists of equals sizes, a target and a source list. Each individual in the second (source) list is copied to the corresponding position in the first list. If the option `-elitist` is present, each individuals from the source list is compared to the corresponding individuals in the target list, and only copied if it is better than the latter one.

4.9 get...

These commands allow the extraction of the internal values of individuals. `getreal` returns the real valued vector of the given individuals, `getsigmas` the step sizes, `getbits` the bit strings, `getfit` the fitness values, `getvalid` the feasibility flags, and `getage` the ages. Note that the result of these commands is a list of lists.

4.10 set...

In contrast to their `get...` counterparts, these commands expect a single individual to be altered. These commands allow an external program to be used as objective function. Let be `simulator` an external program, reading a list of real values from the standard input channel, calculating the fitness of this vector, and writing the evaluated fitness to its standard output. An evaluation loop in Ease is sketched below:

```
foreach indiv $parents {
    set f [open "x.dat" w]
    puts $f [mypop getreal $indiv]
    close $f
    set fx [exec simulator < x.dat]
    mypop setfit $indiv $fx
    mypop setvalid $indiv 1
}
```

5 Example: (μ, λ) Evolution Strategy

The following example implements a (μ, λ) Evolution Strategy. The objective function is the sphere model

$$f(\vec{x}) = \sum_{i=1}^{30} x_i^2$$

.

```
load lib/Ease.so
load lib/Func_sphere.so

set vars 30
set maxgen 1000
set mu 15
set lambda 100
set sigmas n

set parents  [iset 0<$mu>]
set children [iset $mu<$lambda>]

population mypop -vars $vars -sigmas $sigmas -bits 0 -size [expr $mu + $lambda]
set gen 0
set calls 0
```

```
while {$gen <= $maxgen} {
    mypop reproduce $children $parents -select uniform
    mypop mutate $children
    incr calls [mypop evaluate $children]
    mypop copy $parents [mypop best $children $mu]
    puts stdout [mypop stat $parents]
    incr gen
}
```

The example can be invoked from the Ease directory by typing

```
tclsh8.0 examples/ex1.tcl
```

after the command line prompt. A more flexible version of this program can be found in `examples/comma.tcl`. This script can be called directly provided that it is executable for the operating system and that tclsh8.0 is in the search path for executables. Furthermore, it contains a simple command line parser written in Tcl, enabling command line parameters and parameter files. In can be invoked, for example, like this:

```
examples/comma.tcl infile=es.in
```

where `infile` contains parameter settings in Tcl syntax, e.g.

```
set mu 15
set lambda 100
set vars 30
```

Parameter file setting can be overridden by command line setting, e.g.:

```
examples/ex1.tcl infile=es.in vars=10 tau0=0.3
```

If you copy the command line parser into your own scripts, remember to call the `population` command with variables for all parameters you want to be changeable by the command line parser.

6 Example: Linear Ranking Genetic Algorithm

The following example implements a Genetic Algorithm using linear ranking selection. The objective function is a variant of the *counting ones problem*: Minimization of the number of zeroes in a bit string. The following program can be found as *ex2.tcl* in the examples directory:

```
load lib/Ease.so
load lib/Func_countones.so

set bits 128
set mu 50
set maxgen 200
set beta 2.0

set popsize [expr 2 * $mu]

population mypop -size $popsize -bits $bits -vars 0 \
    -mrate 0.01 -breco onepoint -init_zeroes 1.0 -beta $beta

set parents  [iset 0<$mu>]
set children [iset $mu<$mu>]

set gen 0
set calls 0
```

```
incr calls [mypop evaluate $parents]

while {$gen <= $maxgen} {
    mypop reproduce $children $parents -select linrank
    mypop mutate $children
    incr calls [mypop evaluate $children]
    mypop copy $parents $children
    set best [mypop best $parents 1]
    puts stdout [mypop getfit $best]
    incr gen
}
```

A more flexible version of this script can be found in `examples/linrank.tcl`.

7 Genetic Operators

Real Valued Vectors

All operators on real valued vectors are implemented according to Schwefel's book [3]. Discrete recombination is also named *uniform* in analogy to binary uniform crossover. Since geometric intermediate recombination has been added, the original intermediate recombination is also called *arithmetic*.

Bit Strings

Bits are also processed mostly by standard operators. Crossover is provided as one- or two-point crossover, as well as uniform crossover. In analogy to Evolution Strategies, the latter is also called *discrete* in Ease.

Selection

Truncation selection has not been implemented directly. It can be easily expressed by the `best` population command, as shown in the (μ, λ)-ES example. Tournament selection uses deterministic k-ary tournaments. Linear ranking selection is Ease draws from an approximated probability distribution proposed by Whitley [7].

8 User Defined Fitness Functions

The easiest way to define a fitness function is a C-function taking the internal represantation of an individual as arguments and returning the fitness. Since Ease can handle constraints, the fitness must contains a flag indicating the feasibilty of an individual and a fitness value. If the individual is infeasible, the fitness value is interpreted as a measure for the degree of constraint violation. In both cases, feasible or not, lower values denote better quality. The datatype `Fitness` is defined in `ease.h` which must be included into the fitness function's source module:

```
  typedef struct {
      int valid;
      double value;
  }
Fitness;
```

The fitness function itself must be defined using the following prototype:

```
Fitness myfunction(double *x, int n, char *b, int m)
```

The vector `x` contains the real-valued part of the genome, and `b` is the bitstring representation of the binary part of the genome. A particular bit can be extracted from `b` using the `BITTST(b,pos)` macro defined in `ease.h` where `pos` is the position in bitstring b to be tested.

At the end of the fitness function module, the function must be exported using the following macro call:

```
NAME_OF_THIS_FUNCTION_IS(sphere)
```

For each fitness function, a directory must be created in the Ease directory, appending `.dir` to the function name. The directory contains the source module and a header file using the the function name as base names. The header file may be empty. Assuming your function is named `myfunction`, you can now compile it using

```
make f=myfunction
```

9 Modeling of Population Structures

The concepts of index lists used in Ease was developed in conjunction with a unified model of population structures based on hypergraphs [5]. Given a population structure $\Pi = (X, \mathcal{E}, \mathcal{Q})$, $\mathcal{Q}$ can be written as a Tcl list of lists, where each Q_i forms a sublist, and the same can be done with $\mathcal{E}$. The body of an EA with an arbitrary population structure can be sketched as

```
while {$calls <= $maxcalls} {
    foreach offspring $Qtmp deme $E {
        mypop reproduce $offspring $deme -select $select
        mypop mutate $offspring
        incr calls [mypop evaluate $offspring]
    }
    foreach tmp $Qtmp q $Q {
        mypop copy $q $tmp
    }
    set stat [mypop stat $parents]
    puts stdout "$gen $calls $stat"

    if {[lindex $stat 3] < $stopat} {
        break
    }
    incr gen
}
```

Since $\mathcal{E}$ and $\mathcal{Q}$ are two hypergraphs on the same set, we must avoid overlapping modifications. Therefore, each new generation is first stored to `Qtmp`, which has the same structure as Q. When the offspring is complete, it is copied back to the actual population Q.

While this is the general algorithm, it is sometimes easier to implement a particular population structure directly. For a torus based neighborhood model with comma selection, the first step is to define for each individual the set of its neighbors. This can be accomplished as follows:

```
for {set d 0} {$d < $popsize} {incr d} {
    for {set i [expr $d - $nbrad]} {$i <= $d + $nbrad} {incr i} {
        for {set j [expr $d - $nbrad]} {$j <= $d + $nbrad} {incr j} {
            set x [expr ($i + $width) % $width]
            set y [expr ($j + $height) % $height]
            set pos [expr $y * $width + $x]
            lappend deme($d) $pos
        }
        set kid($d) [expr $d + $popsize]
    }
}
```

where `width` and `height` are the width and the height of the torus, `nbrad` is the neighborhood radius as defined in [4]. The body of the neighborhood model is now as simple as

```
while {$gen <= $maxgen} {
    for {set child 0} {$child < $birthrate} {incr child} {
        for {set d 0} {$d < $popsize} {incr d} {
            mypop reproduce $kid($d) $deme($d) -select uniform
        }
        mypop mutate $children
        incr calls [mypop evaluate $children]

        if {$child == 0} {
            mypop copy  $parents $children
        } else {
            mypop copy $parents $children -elitist yes
        }
    }
    puts stdout "$calls [mypop getfit [mypop best $parents 1]]"
    incr gen
}
```

The complete version of this scipt can be found in `examples/nbcomma.tcl`.

10 Modifying the Example Scripts

Tcl is a scripting language developed by John Ousterhout [2]. For those who have not programmed in Tcl before, this section provides some syntactical basics which allow to modify the example scripts. For a good textbook on Tcl, see [6].

The Tcl syntax is simple, but not always intuitive. Everything in Tcl is a list. A Tcl program is a list of lists, where the first element of each list is interpreted as a command. A simple assignment statement, for example, is a list of three elements, where the first element is the `set` command, the second a target variable, and the third the value to be assigned:

```
set mu 15
```

Commands can be separated either by a semicolon or by end of line:

```
set mu 5; set lambda 30
set parents {0 1 2 3 4}
```

In the second line, the value to be assigned is another list. List grouped by brackets can extend over several lines. This makes structured commands more readable:

```
set $i 0
while {$i < $mu} {
    puts $i
    incr i
}
```

Note that the open brace must be in the same line as the `while` statement because a `while` statement is a list of three elements: The statement itself, a conditional expression, and a loop body. The body may be a single statement, but it usually consists of a list of statements. Note that braces also delay the evaluation of the list elements. As a consequence, The value of `i`, written as `$i` in Tcl, is evaluated at each iteration.

Brackets are used in Tcl for command substitution. The statement

```
set popsize [expr $mu + $lambda]
```

first evaluates the `expr` statement, and then assigns the result to the variable `popsize`.

11 Population Parameter Reference

Name	`-size`
Type	Integer
Values	`>0`
Default	`50`
Subcommands	init
Description	Population Size

Name	`-seed`
Type	Integer
Values	`>0`
Default	`0`
Subcommands	init
Description	Random Seed

Name	`-bits`
Type	Integer
Values	`>0`
Default	
Subcommands	init
Description	No Of Binary Genes

Name	`-mrate`
Type	Real
Values	`>0.0`
Default	`0.01`
Subcommands	init configure mutate
Description	Mutation Rate

Name	`-parents`
Type	Integer
Values	`>0`
Default	`2`
Subcommands	init configure reproduce
Description	No of parents

Name	`-select`
Type	Enumeration
Values	`uniform|roulette|tournament|mating|linrank`
Default	`uniform`
Subcommands	init configure reproduce
Description	Selection type

Name	`-sort`
Type	Boolean
Values	`no|yes`
Default	`no`
Subcommands	init configure best worst
Description	Sort best/worst

Name	`-beta`
Type	Real
Values	`>1.0`
Default	`1.5`
Subcommands	init configure reproduce
Description	Linear ranking sel pressure

Name	`-tournsize`
Type	Integer
Values	`>1`
Default	`2`
Subcommands	init configure reproduce
Description	Tournament size

Name	`-xreco`
Type	Enumeration
Values	`none|uniform|discrete|intermediate|arithmetic|geometric`
Default	`discrete`
Subcommands	init configure reproduce
Description	Recombination type for x

Name	`-sreco`
Type	Enumeration
Values	`none|uniform|discrete|intermediate|arithmetic|geometric`
Default	`arithmetic`
Subcommands	init configure reproduce
Description	Recombination type for sigma

Name	`-breco`
Type	Enumeration
Values	`none|onepoint|twopoint|uniform|discrete`
Default	`onepoint`
Subcommands	init configure reproduce
Description	Recombination type for binary genes

Name	`-init_zeroes`
Type	Real
Values	`>0.0`
Default	`0.5`
Subcommands	init
Description	Prob. for initial Zeroes

Name	`-vars`
Type	Integer
Values	`>0`
Default	
Subcommands	init
Description	No Of Variables

Name	`-sigmas`
Type	Enumeration
Values	`1|n`
Default	`1`
Subcommands	init
Description	No Of Sigmas

Name	`-tau0`
Type	Real
Values	`>0.0`
Default	`0.1`
Subcommands	init configure mutate
Description	Global Sigma Variance

Name	`-taul`
Type	Real
Values	`>0.0`
Default	`0.3`
Subcommands	init configure mutate
Description	Local Sigma Variance
Name	`-min_init_x`
Type	Real
Values	`>0.0`
Default	`0.0`
Subcommands	init
Description	Lower Boundary For Initial x[i]
Name	`-max_init_x`
Type	Real
Values	`>0.0`
Default	`1.0`
Subcommands	init
Description	Upper Boundary For Initial x[i]
Name	`-min_init_s`
Type	Real
Values	`>0.0`
Default	`0.001`
Subcommands	init
Description	Lower Boundary For Initial sigma[i]
Name	`-max_init_s`
Type	Real
Values	`>0.0`
Default	`0.1`
Subcommands	init
Description	Upper Boundary For Initial sigma[i]
Name	`-smin`
Type	Real
Values	`>0.0`
Default	`0.0`
Subcommands	init configure
Description	Minimum sigma[i]
Name	`-smax`
Type	Real
Values	`>0.0`
Default	`0.0`
Subcommands	init configure
Description	Maximum sigma[i]
Name	`-elitist`
Type	Boolean
Values	`no|yes`
Default	`no`
Subcommands	init copy
Description	Elitist replacement
Name	`-force`

Type	Boolean
Values	`no|yes`
Default	`no`
Subcommands	init evaluate
Description	Force re-evaluation

12 Platform Dependencies

Ease is written in ANSI C (more or less strictly) and should compile on most platforms. Ease was developed and tested on Solaris 2 and Linux. It compiles on Microsoft 32-bit operationg systems as well. However, the changes required for the dynamic loading of Ease and fitness functions are beyond the scope of this document.

References

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

[2] J. K. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, 1994.

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

[4] J. Sprave. Linear neighborhood evolution strategy. In A. V. Sebald and L. J. Fogel, editors, *Proc. Third Ann. Conf. Evolutionary Programming*, pages 42–51, San Diego, CA, Feb. 1994. World Scientific, Singapur, 1994.

[5] J. Sprave. A unified model of non-panmictic population structures in evolutionary algorithms. Reihe CI 55/99, Sonderforschungsbereich 531, Universität Dortmund, 1999.

[6] B. B. Welsh. *Practical Programming in Tcl and Tk*. Prentice Hall, Upper Saddle River, NJ, 1997.

[7] D. Whitley. The GENITOR algorithm and selection pressure: Why rank–based allocation of reproductive trials is best. In J. D. Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms*, pages 116–121. Morgan Kaufmann Publishers, San Mateo, CA, 1989.