

GENETICA

Genetic Evolution of Novel Entities Through the Interpretation of Composite Abstractions

Documentation of a prototype version



Contents

1	Introduction	1
2	Description of the language	1
2.1	Formulae and terms	1
2.2	Atomic formulae	2
2.2.1	Definition of the interpretation of a list of reals as a probability density function	2
2.2.2	Definitions of atomic formulae	3
2.3	Non atomic formulae	8
2.3.1	Connectives and , or , not , opt	9
2.3.2	Connectives chs and app	10
2.3.3	Connective rec	11
2.4	Code structure	11
3	Definition of the genetic lists and the fitness lists	12
3.1	Genetic lists	12
3.1.1	Definition of the PDF function	12
3.1.2	Definition of the genetic list (GL) of a formula call	12
3.2	Fitness lists	13
3.2.1	Fitness lists of atomic formulae calls	13
3.2.2	Fitness lists of non atomic formulae calls	14
4	The computational system	15
4.1	Genetic operations	15
4.2	The computational process	15
4.2.1	Definition of the application of a probability density function to a list	15
4.2.2	Description of the computational process	15
4.3	Parameters of the computational system	16
5	The environment of GENETICA	17
5.1	Menu “ File ”: projects and file management	17
5.2	Menu “ Compile/Test ”: Compilation and test execution	17
5.3	Menu “ Control ”: Parameters of the computational process	18
5.3.1	The option Basic Control	18
5.3.2	The option Search Basics	19
5.3.3	The option GO Contribution	20
5.3.4	The option Innovation/Extinction	20
5.3.5	The options Crossover Settings , ST-Mut. Settings and Gauss Mut. Settings	21
5.4	Menu “ View ”: Presentation of the computational process	22
5.4.1	The option Population	22

5.4.2	The option Best Solution	23
5.4.3	The option Search Diagrams	23
5.4.4	The option Project Info	24
5.4.5	The option Memory	24
5.5	List Editor and Function Editors	24
5.5.1	The List Editor	24
5.5.2	The Function Editors	26
5.6	The main window	27
6	Tutorial	28
6.1	Application 1: Find Hamilton cycles in graphs	28
6.1.1	Problem statement and problem solving method	28
6.1.2	Presentation of the GENETICA code	29
6.1.3	A case study	32
6.2	Application 2: Construct the control structure if-then-else	35
6.2.1	The aim of the application	35
6.2.2	Presentation of the GENETICA code	36
6.2.3	Testing the if-then-else formula	37



GENETICA: Genetic Evolution of Novel Entities Through the Interpretation of Composite Abstractions

Documentation of a prototype version

1 Introduction

GENETICA is a problem solving computer language integrated in a programming environment that includes a compiler, an evolutionary computational system and a user interface. The problem statement is represented as a GENETICA program while the solution process is realized within the evolutionary computational system. Problem solving is based on the evolution of data generated at runtime while the solution is represented as a data structure. GENETICA can cope with confirmation problems, i.e. problems of which the solution is evaluated in a boolean manner, optimization problems i.e. problems of which the solution is evaluated in a quantitative manner, and problems combining both confirmation and optimization goals.

GENETICA includes the partial recursive functions, the logical operations, the basic arithmetic operations and the quantifiers. GENETICA's atomic terms are integers, reals and symbols represented as strings. Non atomic terms are lists (simple or nested) of atomic terms. Lists can be both constructed and processed in GENETICA by LISP-like formulae. GENETICA's formulae can be also treated as terms, which makes possible high order modes of expression.

Given a GENETICA program, the computational system performs successive executions of the program, where each execution results to a different data generation scenario. Differences between data generation scenarios are caused by the non deterministic elementary decisions occurring during the program execution. These decisions depend on genes both created and structured by the computational system at run-time. Specific gene structures constitute genotypes, referred to as "genetic lists" (GLs), each one deterministically defining a data generation scenario. The successive program executions performed by the computational system result to a population of GLs. The computational system evaluates the performance of each data generation scenario in either confirming a specific formula or optimizing the value of a specific function within the program. The evaluation provides a fitness value for the GL of the data generation scenario. The computational system evolves the population by performing genetic operations on high fitness GLs, estimating the fitness of the GLs resulting from the genetic operations, and substituting low fitness GLs in the population. The best fitness GL of the population, after the evolution procedure, defines the data generation scenario that results to the solution. The latter is a data structure constructed by a specific formula within the program.

2 Description of the language

2.1 Formulae and terms

The notion of a GENETICA "formula" includes all the operations, the functions and the relations that either constitute GENETICA's atomic elements or can be constructed in GENETICA. GENETICA's formulae have input terms and a single output term. Given the input terms the formula can be "called" i.e.

the program represented by the formula can be executed. The output term emerge as a result of the call.

Atomic terms in GENETICA are integers, real numbers, symbols viewed as strings, characters, and the standard value **FALSE**. An empty list is also considered as an atomic term. If t_1, \dots, t_n are terms then the list (t_1, \dots, t_n) is also a term. This recursively defines as a term any nested list whose innermost elements are atomic terms. The characters **c**, **s**, **i**, **f** and **l**, also called “type indication characters”, indicate a character, a symbol, an integer, a real number (float) and a list respectively.

GENETICA’s terms have formula scope and duration, i.e. they can be used only in the formula where they have been declared and only during the execution of a call to the formula. However any list-term can be registered in memory, pointed by an integer-pointer, and stay there until it is explicitly discarded. This is the case of a pointer list. Pointer lists have arbitrary duration and can be accessed by any formula where their integer-pointer has been passed.

A class of formulae return no output term. These formulae, which represent relations, will be referred to as “non constructive” formulae, whereas any other formula will be referred to as a “constructive” formula. The standard value **NULL**, represented in the syntax by the symbol “**NULL**”, substitutes the output term and denotes a non constructive formula, whereas any other symbol denotes a constructive formula.

A class of atomic formulae allow reference to random elements of either countable classes, viewed as lists, or continuous classes, viewed as real intervals. These formulae will be referred to as “multiple confirmation” formulae, in the sense that given the input values different output values confirm the formula. Any non atomic formula that makes reference to a “multiple confirmation” formula is also defined to be a “multiple confirmation” formula (recursive definition). Any non-multiple-confirmation formula will be referred to as a “single confirmation” formula.

A special class of atomic formulae, referred to as “high order” formulae, have a single input term which is a list having the form (F_N, t_1, \dots, t_n) where F_N is the name of a formula (atomic or not) represented as a string. This list will be referred to as a “call list”. When a “high order” formula is called, the formula named F_N is called with input terms t_1, \dots, t_n , while the output term of the call is included in the output of the “high order” formula call. The call to the formula named F_N , resulting from the “high order” formula call, will be referred to as the “forced call”. Any non atomic formula that includes a reference to a “high order” formula is also defined to be a “high order” formula (recursive definition). Any non “high order” formula will be referred to as a “first order” formula.

The standard value **FALSE** constitutes the output term of any constructive formula in the case of disconfirmation.

2.2 Atomic formulae

2.2.1 Definition of the interpretation of a list of reals as a probability density function

Let L be a list of non negative real numbers (r_1, \dots, r_n) where $r_1 + \dots + r_n > 0$. Consider the probability density function $F(x)$, defined in $[0, 1]$ by the following expressions:

$$x < 1 \Rightarrow i = \text{int}(n \cdot x) + 1, \text{ where } \text{int} \text{ denotes the “integer part” function,}$$

$$x = 1 \Rightarrow i = n,$$

$$F(x) = c_L \cdot r_i, \text{ where } c_L \text{ is a constant depending on } L,$$

$$\int_0^1 F(x) dx = 1$$

$F(x)$ will be referred to as the interpretation of L as a probability density function.

2.2.2 Definitions of atomic formulae

Syntactically, an atomic formula is a triplet having the form :

< formula name, name of the output term, list of the names of the input terms >

where names are viewed as strings.

The definitions of GENETICA's atomic formulae are presented here. Within the following presentation, sometimes formulae will be referred to by their names, while terms will be referred to by the symbols representing their names, e.g. the expression "confirmation of **F**" means "confirmation of the formula having name the string **F**" while the expression "**t** is a list" means "the term having name the string represented by the symbol **t** is a list".

If the name of a formula is an abbreviation of a word or a phrase in the natural language, then the latter word or phrase appears in parentheses without constituting part of the definition.

add **T** (**t1**, **t2**)

which is always confirmed. If **t1** is a list and **t2** is a non-**FALSE** term of arbitrary type then **T** is the list produced by appending **t2** at the end of **t1** as an element. Otherwise **T** is the value **FALSE** despite of the confirmation of **add**.

at **T** (**t**) (always true)

which is always confirmed. If **t** is a list having the form (**F_N**, **v1**, ... **vn**) where **F_N** is a formula name viewed as a string, and (**v1**, ... **vn**) is an input vector value for the formula named **F_N**, then **T** is the output value of the "forced call" (see § 2.1), i.e. a call to the formula named **F_N**, with input values **v1**, ... **vn**. Otherwise **T** is the value **FALSE** despite of the confirmation of **at**.

bf **T** (**t**) (butfirst)

which is confirmed if and only if **t** is a non empty list. In the case of confirmation **T** is the remaining part of **t** after removing the first element.

bl **T** (**t**) (butlast)

which is confirmed if and only if **t** is a non empty list. In the case of confirmation **T** is the remaining part of **t** after removing the last element.

call **T** (**t**)

which is confirmed if and only if **t** is a list having the form (**F_N**, **v1**, ... **vn**) where **F_N** is a formula name viewed as a string, and (**v1**, ... **vn**) is an input vector value for the formula named **F_N**, while the "forced call" (see § 2.1), i.e. a call to the formula named **F_N**, with input values **v1**, ... **vn** is confirmed. In the case of confirmation **T** is the output value of the "forced call".

cfp **T** (**t1**, **t2**) (copy from pointer)

which is confirmed if and only if **t1** is a pointer to a term and **t2** is the term's type indication character (see § 2.1). In the case of confirmation **T** is a copy of the term pointed by **t1**.

crd **T** **(t)** (cardinality)

which is confirmed if and only if **t** is a list. In the case of confirmation **T** is the number of the elements of **t**.

cp **T** **(t)** (copy)

which is always confirmed. **T** is a copy of **t**.

dlt **NULL** **(t₁, t₂)** (delete)

which is confirmed if and only if **t₁** is a pointer to a term and **t₂** is the term's type indication character (see § 2.1). In the case of confirmation the term pointed by **t₁** is destroyed while the memory occupied by this term is released.

dvs **T** **(t₁, t₂)** (division)

which is confirmed if and only if both **t₁** and **t₂** are real numbers. In the case of confirmation **T** is the quotient of **t₁** to **t₂**.

eq **NULL** **(t₁, t₂)** (equal)

which is confirmed if and only if **t₁** and **t₂** are non-**FALSE** equal terms of arbitrary type.

ex **T** **(t)** (exponent)

which is confirmed if and only if **t** is a real number. In the case of confirmation **T** = **e^t**.

fi **T** **(t)** (first)

which is confirmed if and only if **t** is a non empty list. In the case of confirmation **T** is the first element of **t**.

fti **T** **(t)** (float to integer)

which is confirmed if and only if **t** is a real number. In the case of confirmation **T** is the integer part of **t** as an integer.

in **T** **(t₁, t₂)**

which is confirmed if and only if both **t₁** and **t₂** are real numbers where **t₂ > t₁**. In the case of confirmation **T** is a random real value in the interval [**t₁**, **t₂**].

ipd **T** **(t₁, t₂, t₃)** (in interval with probability density)

which is confirmed if and only if both **t₁** and **t₂** are real numbers where **t₂ > t₁**, while **t₃** is a list of non negative real numbers (**r₁, ... r_n**) where **r₁ + ... + r_n > 0**. In the case of confirmation **T** = **t₁ + (t₂ - t₁)·u** where **u** is a random real value in [**0**, **1**] having probability density the interpretation of **t₃** as a probability density function (see § 2.2.1).

ism **NULL** **(t₁, t₂)** (is member)

which is confirmed if and only if **t₁** is a list and **t₂** is an element of **t₁**.

itf **T** **(t)** (integer to float)

which is confirmed if and only if **t** is an integer. In the case of confirmation **T** is a real number equal to **t**.

jo **T** **(t₁, t₂)** (join)

which is always confirmed. If both **t₁** and **t₂** are lists, where **t₁ = (x₁, ... x_n)** and **t₂ = (y₁, ... y_k)**, then **T** is the list **(x₁, ... x_n, y₁, ... y_k)**. Otherwise **T** is the value **FALSE** despite of the confirmation of **jo**.

la **T** **(t)** (last)

which is confirmed if and only if **t** is a non empty list. In the case of confirmation **T** is the last element of **t**.

lg **T** **(t)** (logarithm)

which is confirmed if and only if **t** is a positive real number. In the case of confirmation **T** is the natural logarithm of **t**.

lop **T** **(t)** (list of pointers)

which is confirmed if and only if **t** is a pointer to a list. In the case of confirmation **T** is a list of pointers to the homologous elements of the list.

max **T** **(t)**

which is confirmed if and only if **t** is a list of real numbers. In the case of confirmation **T** is the largest number of the list.

mem **T** **(t)** (member)

which is confirmed if and only if **t** is a non empty list. In the case of confirmation **T** is a randomly chosen element of **t**.

min **T** **(t)**

which is confirmed if and only if **t** is a list of real numbers. In the case of confirmation **T** is the smallest number of the list.

mlt **T** **(t₁, t₂)** (multiplication)

which is confirmed if and only if both **t₁** and **t₂** are real numbers. In the case of confirmation **T** is the product of **t₁** and **t₂**.

mns **T** **(t₁, t₂)** (minus)

which is confirmed if and only if both **t₁** and **t₂** are real numbers. In the case of confirmation **T** is the difference **t₁** minus **t₂**.

mpd **T** **(t₁, t₂)** (member with probability density)

which is confirmed if and only if **t₁** is a list and **t₂** is a list of non negative real numbers (**r₁, ... r_n**) where **r₁ + ... + r_n > 0**. In the case of confirmation **T** is an element of **t₁** selected with probability density determined by the interpretation of **t₂** as a probability density function (see § 2.2.1). Specifically, the real interval **[0, 1]** is divided in **n** equal sub-intervals, where **n** is the number of the elements of **t₁**. The **ith** sub-interval (**i = 1, ... n**) corresponds to the **ith** element of **t₁**. The selection probability of each element of **t₁** equals the probability assigned to the corresponding sub-interval by the interpretation of **t₂** as a probability density function.

padd **NULL** **(t₁, t₂, t₃)** (pointer - add)

which is confirmed if and only if **t₁** is a pointer to a list, **t₂** is a pointer to a non-**FALSE** term of arbitrary type and **t₃** is the type indication character (see § 2.1) of the term pointed by **t₂**. In the case of confirmation **t₂** is appended as an element at the end of the list pointed by **t₁**.

pbf **NULL** **(t)** (pointer - butfirst)

which is confirmed if and only if **t** is a pointer to a non empty list. In the case of confirmation the first element of the list is removed.

pbl **NULL** **(t)** (pointer - butlast)

which is confirmed if and only if **t** is a pointer to a non empty list. In the case of confirmation the last element of the list is removed.

pfi **T** **(t)** (pointer - first)

which is confirmed if and only if **t** is a pointer to a non empty list. In the case of confirmation **T** is the pointer to the first element of the list.

pjo **NULL** **(t₁, t₂)** (pointer - join)

which is confirmed if and only if both **t₁** and **t₂** are pointers to lists. In the case of confirmation the list pointed by **t₂** is adjoined at the end of the list pointed by **t₁** while **t₂** is discarded.

pl **T** **(t₁, t₂)** (property list)

which is confirmed if and only if **t₁** is a list of pairs (two-element lists) having the form **((x₁, y₁), ... (x_n, y_n))** where **x₁, ... x_n, y₁, ... y_n** are non-**FALSE** terms of arbitrary type, and **t₂ ∈ {x₁, ... x_n}**. In the case of confirmation **T = y_k** where **k** is the least integer confirming the equation **x_k = t₂**.

pla **T** **(t)** (pointer - last)

which is confirmed if and only if **t** is a pointer to a non empty list. In the case of confirmation **T** is the pointer to the last element of the list.

pls **T** **(t₁, t₂)** (plus)

which is confirmed if and only if both **t₁** and **t₂** are real numbers. In the case of confirmation **T** is the sum of **t₁** and **t₂**.

poc **T** **(t)** (pointer of copy)

which is confirmed if and only if **t** is a non-**FALSE** term of arbitrary type. In the case of confirmation a copy of **t** is created while **T** is the pointer to the copy.

ppl **T** **(t₁, t₂)** (pointer property list)

which is confirmed if and only if **t₁** is a pointer to a list of pairs (two-element lists) having the form **((x₁, y₁), ... (x_n, y_n))**, where **x₁, ... x_n, y₁, ... y_n** are non-**FALSE** terms of arbitrary type, and **t₂ ∈ {x₁, ... x_n}**. In the case of confirmation **T = y_k** where **k** is the least integer confirming the equation **x_k = t₂**.

pri **NULL** **(t₁, t₂)** (pointer – revise integer)

which is confirmed if and only if **t₁** is a pointer to an integer and **t₂** is an integer. In the case of confirmation the integer pointed by **t₁** is assigned the value of **t₂**.

pspl **T** **(t₁, t₂)** (pointer - split)

which is confirmed if and only if **t₁** is a pointer to a list and **t₂** is a pointer to an element of the list pointed by **t₁**. In the case of confirmation the sub-list starting with this element and ending at the end of the list is removed from the list and becomes an independent list while **T** is a pointer to the latter list.

sbs **T** **(t₁, t₂, t₃)** (substitute)

which is confirmed if and only if **t₁** is a list and both **t₂** and **t₃** are non-**FALSE** terms of arbitrary type. In the case of confirmation **T** is the list derived from **t₁** if each element of **t₁** equal to **t₂** is substituted by **t₃**.

ser **T** **(t₁, t₂)** (series)

which is confirmed if and only if **t₁** is a list and **t₂** is a real number where the integer part of **t₂**, name it **n**, is a non-negative integer less than or equal to the size of **t₁**. In the case of confirmation **T** is the **nth** element of **t₁**.

sml **NULL** **(t₁, t₂)** (smaller)

which is confirmed if and only if both **t₁** and **t₂** are real numbers where **t₁ < t₂**.

spl **T** **(t₁, t₂)** (split)

which is confirmed if and only if **t₁** is a list and **t₂** is an element of **t₁**. In the case of confirmation **T** is a pair of sublists of **t₁**: the first sub-list contains the **n - 1** first elements of **t₁** were **n** is the order of the first occurrence of **t₂** in **t₁**, while the second sub-list contains the remaining part of **t₁**. If **n = 1** then the first sublist is empty while the second one is identical to **t₁**.

sq **T** **(t)** (square)

which is confirmed if and only if **t** is a non negative real number. In the case of confirmation **T** is the square root of **t**.

tgf **T** **(t)** (term - genetic list - fitness)

which is confirmed if and only if **t** is a list having the form **(F_N, v₁, ... v_n)** where **F_N** is a formula name viewed as a string, and **(v₁, ... v_n)** is an input vector value for the formula named **F_N**, while the “forced call” (see § 2.1), i.e. a call to the formula named **F_N**, with input values **v₁, ... v_n** is confirmed. In the case of confirmation **T** is the list **(t, g, f)** were **t** is the output value of the “forced call” (see § 2.1), **g** is a copy of the genetic list (see § 3.1) of the “forced call” and **f** is a copy of the fitness list (see § 3.2) of the “forced call”. **(t, g, f)** will be referred to as the “tgf-list” of the “forced call”.

top **T** **(t)** (term of pointer)

which is confirmed if and only if **t** is a pointer to a list. In the case of confirmation the memory occupied by the list is released while **T** is a copy of the list.

vf **NULL** **(t)** (virtual fitness)

which is confirmed if and only if **t** is a list where the first element of the list is the real **0.0**. The fitness list (see § 3.2) of a call to **vf** is **(e₁, e₂)** where **e₁** and **e₂** are respectively the first and the last element of **t**.

2.3 *Non atomic formulae*

Non atomic formulae are constructed by GENETICA’s connectives **and**, **or**, **not**, **opt**, **app**, **chs** and **rec**. The definition of each non atomic formula includes references to other formulae, either atomic or not. A GENETICA program can be represented as a tree structure, referred to as the “syntax tree”, where each formula definition is represented as a node having child nodes the definitions of the referenced formulae. Atomic formulae are represented as terminals (leaf nodes). The formula whose definition is represented as the root of the syntax tree represents the problem statement and will be referred to as the “root formula”. The confirmation state and the output term of a call to a non atomic formula **F**, given the input terms, depends on the calls to the formulae referenced in the definition of **F**. These calls will be referred to as calls “directly caused” by the **F** call. The syntax of the definition, the confirmation conditions and the calculation of the output value of non atomic formulae are described here.

Within the following definitions both formula names and term names are viewed as strings, while both formulae and terms sometimes will be referred to by the symbols representing their names (e.g. “a call to **F**” means “a call to the formula having name the string represented by the symbol **F**”).

2.3.1 Connectives **and**, **or**, **not**, **opt**

a) Syntax

FD **con**
 Tout (**Tin**₁ ... **Tin**_n)
 (**FR**₁ **Tout**₁ (**T**_{1,1} ... **T**_{1,x1})
 FR₂ **Tout**₂ (**T**_{2,1} ... **T**_{2,x2})
 ...
 FR_y **Tout**_y (**T**_{y,1} ... **T**_{y,xy}))

where :

n, y, x1, x2, ... xy ∈ **N**,
FD : is the name of the non-atomic formula
con : is the connective
Tout : is the name of the output term of **FD**
Tin₁, **Tin**₂, ... **Tin**_n : are the names of the input terms of **FD**
FR_i (**i** ∈ {**1, ... y**}) : is the name of a referenced formula
Tout_i (**i** ∈ {**1, ... y**}) : is the name of the output term of **FR**_i
T_{i,1}, **T**_{i,2}, ... **T**_{i,xi} (**i** ∈ {**1, ... y**}) : are the names of the input terms of **FR**_i

Constraints:

- **n** ≥ 1
- **Tin**_i ≠ **Tin**_j, **Tout**_h ≠ **Tout**_m, **Tin**_p ≠ **Tout**_k, (**i** ≠ **j**; **h** ≠ **m**; **1** ≤ **i, j, p** ≤ **n**; **1** ≤ **h, m, k** ≤ **y**)
- **T**_{1,j} ∈ {**Tin**₁, ... **Tin**_n} (**1** ≤ **j** ≤ **x1**)
- **T**_{i,j} ∈ {**Tin**₁, ... **Tin**_n} ∪ {**Tout**₁, ... **Tout**_{i-1}} (**1** < **i** ≤ **y**; **1** ≤ **j** ≤ **x_i**)
- **con** = **and** ⇒ [**Tout**_y = “NULL” ⇒ **Tout** = “NULL”]
- **con** = **or** ⇒ [**T**_{i,j} ∈ {**Tin**₁, ... **Tin**_n} (**1** ≤ **i** ≤ **y**; **1** ≤ **j** ≤ **x_i**)]
- **con** = **not** ⇒ [**Tout** = “NULL” ∧ **y** = 1]
- **con** = **opt** ⇒ [**Tout** ≠ “NULL” ∧ **y** = 2 ∧ **Tout**₁ ≠ “NULL” ∧ **Tout**₂ ≠ “NULL” ∧ **Tout**₁ ∈ {**T**_{2,1}, ... **T**_{2,x2}}]

When **FD** is called each referenced formula is called once. Referenced formulae are called in the order they appear in the definition of **FD**.

b) Confirmation and production of the output value

con = and

A **FD** call is confirmed if and only if all the directly caused calls are confirmed. In the case of confirmation, the output term is either the output term of the directly caused **FR**_y call or **NULL**, depending on whether **FD** is defined as a constructive (**Tout** ≠ “NULL”) or a non-constructive (**Tout** = “NULL”) formula.

con = or

A **FD** call is confirmed if and only if at least one directly caused call is confirmed. If **FD** is defined as a constructive formula (**Tout** ≠ “NULL”), then in the case of confirmation the output term is the list of the

output terms of the directly caused calls that are confirmed. The ordering of these terms in the list respects the ordering of the corresponding referenced formulae in the definition of **FD**.

con = not

A **FD** call is confirmed if and only if the directly caused **FR₁** call is not confirmed.

con = opt (optimize)

opt is a special connective that only constructs the root formula in optimization problems. A formula constructed by the connective **opt** can not be referenced in the definition of another formula. The output term of **FR₁** represents a potential solution. This is used as an input term in **FR₂** whose output term represents a quantity which is to be maximized. A **FD** call is confirmed if and only if the directly caused **FR₁** call is confirmed. In the case of confirmation the output term of the **FD** call is the output term of the directly caused **FR₁** call.

2.3.2 Connectives **chs** and **app**

a) Syntax

FD **con**
 Tout (**Tin₁** ... **Tin_n**)
 (**FR₁**)

where :

n ∈ **N**,
FD : is the name of the non-atomic formula
con : is the connective
Tout : is the name of the output term of **FD**
Tin₁, ... Tin_n : are the names of the input terms of **FD**
FR₁ : is the name of a referenced formula

Constraints:

- If **Tout** ≠ **NULL**, then **FR₁** should be a constructive formula.
- The number of the input terms of **FR₁** should be **n**
- **Tin₁** should be a list

b) Confirmation and production of the output value

con = chs (choose)

Let **e_i** be the **ith** element of the list **Tin₁** while **s** is the size of the list. In a **FD** call, **FR₁** is called **s** times. Each call has the input values of the **FD** call, except the list **Tin₁** which is substituted by **e_i** in the **ith** call. **FD** is confirmed if and only if at least one **FR₁** call is confirmed. If **Tout** ≠ **“NULL”** then in the case of confirmation the output value of the **FD** call is the list of the output values of the **FR₁** calls that are confirmed. The ordering of these values in the list respects the ordering of the corresponding elements in the list **Tin₁**.

con = app (apply)

Let e_i be the i^{th} element of the list \mathbf{Tin}_1 while s is the size of the list. In a **FD** call, **FR₁** is called s times. Each call has the input values of the **FD** call, except the list \mathbf{Tin}_1 which is substituted by e_i in the i^{th} call. **FD** is confirmed if and only if all the **FR₁** calls are confirmed. If $\mathbf{Tout} \neq \text{"NULL"}$ then in the case of confirmation the output value of the **FD** call is the list of the output values of all the **FR₁** calls. The ordering of these values in the list respects the ordering of the corresponding elements in the list \mathbf{Tin}_1 .

2.3.3 Connective **rec** (recursion)

a) Syntax

FD **rec**
 Tout (**Tin₁** ... **Tin_n**)
 (**FR₁**
 FR₂)

where :

- n** \in **N**,
- FD** : is the name of the non-atomic formula
- Tout** : is the name of the output term of **FD**
- Tin₁, Tin₂, ... Tin_n** : are the names of the input terms of **FD**
- FR₁, FR₂** : are the names of the referenced formulae

Constraints:

- **FR₁** should be a non constructive formula
- **FR₂** should be a constructive formula
- **n = n₁ + n₂ - 1** where **n₁** and **n₂** are the numbers of the input terms of the formulae **FR₁** and **FR₂** respectively.

b) Confirmation and production of the output value

Let **n₁** and **n₂** are the numbers of the input terms in the definitions of **FR₁** and **FR₂** respectively. In a **FD** call, **FR₁** is called with input values $v(\mathbf{Tin}_1), \dots, v(\mathbf{Tin}_{n_1-1}), v(\mathbf{Tin}_n)$, where $v(\mathbf{Tin}_x)$ ($x = 1, \dots, n$) denotes the value of the term named \mathbf{Tin}_x . If the **FR₁** call is confirmed, then the **FD** call is also confirmed with output value $v(\mathbf{Tin}_n)$. Otherwise, **FR₂** is called with input values $v(\mathbf{Tin}_{n_1}), \dots, v(\mathbf{Tin}_n)$. If the **FR₂** call is confirmed then **FD** is called again with the initial input values except $v(\mathbf{Tin}_n)$, which is substituted by the output value of the **FR₂** call. If the **FR₂** call is not confirmed, then the **FD** call is not confirmed either.

2.4 Code structure

GENETICA code consists of non atomic formulae definitions. The first formula defined in the code is considered as the root formula which represents the problem to be solved. A formula can be referenced by another formula regardless of the order of the formulae definitions in the code. Successive symbols should be separated by at least one space, tab or "change line" character. However parenthesis symbols "(" and ")" are not necessarily separated from the next and the previous symbol respectively.

The character “!” denotes the start and the end of a comment. Comments can be placed anywhere in the code, but the code cannot end with a comment.

3 Definition of the genetic lists and the fitness lists

3.1 Genetic lists

3.1.1 Definition of the PDF function

Let \mathbf{L} be a list of non negative real numbers (r_1, \dots, r_n) where $r_1 + \dots + r_n > 0$. Let the function $\mathbf{F}(\mathbf{x})$, defined in $[0, 1]$, be the interpretation of \mathbf{L} as a probability density function (see § 2.2.1). Define the function $\mathbf{PDF}(\mathbf{L}, \mathbf{w})$, where \mathbf{PDF} stands for “probability density function” and $\mathbf{w} \in [0, 1]$, by the expression:

$$\mathbf{PDF}(\mathbf{L}, \mathbf{w}) = t \Leftrightarrow \int_0^t \mathbf{F}(\mathbf{x}) d\mathbf{x} = \mathbf{w}, \text{ where } t \in [0, 1]$$

3.1.2 Definition of the genetic list (GL) of a formula call

- The GL of a disconfirmed call to any atomic formula is the empty list.
- The GL of a confirmed call to the atomic formula **in** with input vector $(\mathbf{t}_1, \mathbf{t}_2)$ and output \mathbf{t} is the list (\mathbf{w}) where $\mathbf{t} = \mathbf{t}_1 + \mathbf{w} \cdot (\mathbf{t}_2 - \mathbf{t}_1)$ and $\mathbf{t}_1 = \mathbf{t}_2 \Rightarrow \mathbf{w} = 0$.
- The GL of a confirmed call to the atomic formula **mem** with input vector (\mathbf{t}) where $\mathbf{t} = (\mathbf{e}_1, \dots, \mathbf{e}_k)$ and output \mathbf{e}_i ($i \in \{1, \dots, k\}$) is the list (\mathbf{w}) where $\mathbf{w} \cdot \mathbf{k} = i - 0.5$.
- The GL of a confirmed call to the atomic formula **ipd** with input vector $(\mathbf{t}_1, \mathbf{t}_2, \mathbf{t}_3)$ and output \mathbf{t} is the list (\mathbf{w}) where $\mathbf{t} = \mathbf{t}_1 + \mathbf{PDF}(\mathbf{t}_3, \mathbf{w}) \cdot (\mathbf{t}_2 - \mathbf{t}_1)$ and $\mathbf{t}_1 = \mathbf{t}_2 \Rightarrow \mathbf{w} = 0$.
- The GL of a confirmed call to the atomic formula **mpd** with input vector $(\mathbf{t}_1, \mathbf{t}_2)$, where $\mathbf{t}_1 = (\mathbf{e}_1, \dots, \mathbf{e}_k)$, and output \mathbf{e}_i ($i \in \{1, \dots, k\}$) is the list (\mathbf{w}) where $\mathbf{PDF}(\mathbf{t}_2, \mathbf{w}) \cdot \mathbf{k} = i - 0.5$.
- The GL of a call to the atomic formula **call**, **at** or **tgf** equals the GL of the “forced call” (see § 2.1) if the latter is realized (see the definitions of **call**, **at** and **tgf** in § 2.2.2). Otherwise the GL is the empty list.
- The GL of a call to any first order non constructive atomic formula is the empty list.
- The GL of a call to a non atomic formula \mathbf{F} is the list of the GLs of the directly caused calls if at least one of the latter GLs is a non empty list. Otherwise the GL of the call to \mathbf{F} is the empty list.

Due to the previous definitions, a GL is either an empty list or a nested list which can be represented as a tree structure where inclusion relations are interpreted as parent-child relations while innermost lists (which are either empty lists or lists each one including a single real value) are interpreted as terminals (leaf nodes). Given a formula \mathbf{F} and the input terms of \mathbf{F} , the previous definitions specify a one-to-one map between the class \mathbf{C} of the potential calls to \mathbf{F} , where each call represents a specific data generation scenario, and the class \mathbf{G} of the GLs of these calls. Due to this map, the GLs can be considered as genotypes of data generation scenarios. GLs are both constructed and evolved by the computational system of GENETICA’s environment. Genetic operations on GLs (see § 4.1) may produce lists that do not belong to \mathbf{G} , as a consequence they can not be considered as genotypes. Name \mathbf{S} the class of the lists that are either GLs or they can be produced by genetic operations on GLs. The computational system provides a transformation procedure, executed at runtime, that takes each element of \mathbf{S} to an element of \mathbf{G} . The transformation, which will be referred to as “reconstruction”, has the following properties:

- The empty list, can be potentially reconstructed to any element of **G**.
- Any element of **G** is always reconstructed to itself.
- If $\mathbf{s} \in \mathbf{S}$, $\mathbf{s} \notin \mathbf{G}$, $\mathbf{s} \neq ()$ and \mathbf{s} is reconstructed to $\mathbf{g} \in \mathbf{G}$ then the parts of \mathbf{s} that are consistent with the definition of a GL are preserved in \mathbf{g} .

3.2 Fitness lists

The fitness of a GENETICA formula call is defined by a list referred to as the fitness list, which is characteristic of the call. The fitness list is typically a pair of real numbers: the first number represents a magnitude referred to as the “distance from confirmation” of the call while the second number represents a magnitude referred to as the “distance from disconfirmation” of the call. The only exception from this description concerns the fitness list of a call to a non atomic formula constructed with **opt** (see § 3.2.2).

The definition of the fitness list of a formula call is presented in the following paragraphs.

3.2.1 Fitness lists of atomic formulae calls

a) General definition of the fitness list of an atomic formula call

The fitness list of an atomic formula call is the list **(0, 1)** in the case of confirmation and the list **(1, 0)** in the case of disconfirmation. Exceptions from this general definition are presented in the following cases.

b) Fitness list of a call to the atomic formula **eq**

The fitness list of a call to the atomic formula **eq** (see § 2.2.2) is the list **(d_{eq} , d'_{eq})** where:

- d_{eq} is **0** in the case of confirmation of the call, while in the case of disconfirmation it is defined by the following rules:

Let **(t_1 , t_2)** be the input vector of the call.

- If t_1 and t_2 are either terms of different type, or both different integers, different characters, different strings or lists of different sizes, then $d_{eq} = 1$.
- If t_1 and t_2 are both real numbers, then $d_{eq} = 1 - \frac{1}{1 + |t_1 - t_2|}$.
- If $t_1 = (x_{1,1}, \dots, x_{1,n})$ and $t_2 = (x_{2,1}, \dots, x_{2,n})$, where $n \in \mathbf{N}$ and $x_{1,1}, \dots, x_{1,n}, x_{2,1}, \dots, x_{2,n}$ are arbitrary terms, then $d_{eq} = \max_{i=1}^n \{D_{eq}(x_{1,i}, x_{2,i})\}$, where $D_{eq}(x_{1,i}, x_{2,i})$ ($i = 1, \dots, n$) is the first element of the fitness list of a call to **eq** with input vector **($x_{1,i}, x_{2,i}$)**.

- d'_{eq} is **1** in the case of confirmation and **0** in the case of disconfirmation.

c) Fitness list of calls to the atomic formulae **call**, **tgf** and **vf**

The fitness list of a call to either the atomic formula **call** or the atomic formula **tgf** is the fitness list of the “forced call” (see § 2.1). The fitness list of a call to the atomic formula **vf** is explicitly defined by an input term as described in the definition of **vf** (see § 2.2.2).

d) Fitness list of calls to the “always true” atomic formulae

A special class of GENETICA’s formulae are characterized as “always true”. The fitness list of a call to an “always true” formula is $(0, -1)$ for any combination of input values consistent with the formula’s definition, where the value -1 denotes ∞ (infinity). The “always true” atomic formulae are the formulae **add, at, cfp, cp, crd, dlt, dvs, ex, fti, itf, jo, lg, lop, max, min, mlt, mns, padd, pjo, pls, poc, pri, sbs, sq** and **top**.

The fitness list of a call to an “always true” formula where the input terms are inconsistent with the formula’s definition is $(1, 0)$ except the formulae **add, at** and **jo** where the fitness list is $(0, -1)$ even in the case of inconsistency.

3.2.2 Fitness lists of non atomic formulae calls

Let (d_k, d'_k) ($k = 1, \dots, h$) be the fitness list of the k^{th} call directly caused by a call to a non atomic formula **F**, where **h** is the number of the calls directly caused by the **F** call.

If **F** is constructed with **opt**, let d_1, d_2 and **Q** be respectively the “distance from confirmation” of the first directly caused call (i.e. the call to the formula producing a potential solution: see § 2.3.1.b, connective **opt**), the “distance from confirmation” of the second directly caused call (i.e. the call to the formula performing evaluation of the potential solution), and the output value of the second directly caused call (i.e. the value to be maximized).

The fitness list of the **F** call depends on the connective in the definition of **F** and it is defined in the following table.

<i>Connective</i>	<i>Fitness list</i>
and, app	$(d_1 + \dots + d_h, \min(d'_1, \dots, d'_h))$
or, chs	$(\min(d_1, \dots, d_h), d'_1 + \dots + d'_h)$
not	(d'_1, d_1)
opt	(d_1) if $d_1 \neq 0$ (d_1, d_2) if $d_1 = 0$ and $d_2 \neq 0$ (d_1, d_2, Q) if $d_1 = 0$ and $d_2 = 0$
rec	$(\min(d_1, d_3), \min(d'_2, d'_3))$ if $d_1 \neq 0$ and $d_2 = 0$ (d_1, d'_1) if $d_1 = 0$ or $d_2 \neq 0$

The objective of a call to a formula not constructed with **opt** is to minimize the first element of the fitness list. The objective of a call to a formula constructed with **opt** is to minimize d_1 , to minimize d_2 and to maximize **Q**, in this order of importance. As a consequence the fitness of different calls to the same formula are comparable: a better fitness call is a call with higher performance with respect to the aforementioned objectives.

The fitness list of a call can be directly assigned to the GL of the call, since the GL deterministically defines the data generation scenario, the confirmation state and the output value of the call.

4 The computational system

4.1 Genetic operations

GENETICA's computational system provides three genetic operations on GLs: "homologous crossover", "subtree mutation" and "Gaussian mutation". The genetic operations are described here:

1 Homologous Crossover

Two parent GL-trees are aligned, the "common region", i.e. the largest tree structure whose nodes have homologous nodes in both parent trees, is identified and different nodes of the "common region" specify pairs of homologous nodes of the parent trees. Paired nodes are swapped. The expected depth of the nodes to be swapped in the parent trees can be regulated by a probability function. Within a GL, high-depth nodes, resulting to small building blocks, are usually associated to low-level semantic properties, whereas low-depth nodes, resulting to larger building blocks, are usually associated to higher-level semantic properties.

2 Subtree Mutation

Random nodes in a GL-tree are substituted by empty lists. This results to cropping of subtrees within the GL-tree. The cropped subtrees of the resultant list will be randomly redeveloped during the "reconstruction" procedure (see § 3.1.2). Like in crossover, the expected depth of the mutated node can be regulated by a probability function.

3 Gaussian Mutation

Gaussian mutation changes the real values within GL terminals. The new values have probability density specified by Gaussian functions maximized at the initial value.

4.2 The computational process

4.2.1 Definition of the application of a probability density function to a list

Let \mathbf{L} be a list and $\mathbf{F}(\mathbf{x})$ be a probability density function defined in the real interval \mathbf{D} . Divide \mathbf{D} in \mathbf{n} equal sub-intervals where \mathbf{n} is the size of \mathbf{L} . We will say that $\mathbf{F}(\mathbf{x})$ is applied to \mathbf{L} if the probability of the i^{th} ($i = 1, \dots, \mathbf{n}$) element of \mathbf{L} equals the probability assigned by $\mathbf{F}(\mathbf{x})$ to the i^{th} sub-interval of \mathbf{D} .

4.2.2 Description of the computational process

The computational process can start when both a GENETICA program and the input vector for the root formula within the program have been defined. It includes the phase of the population initialization and the phase of the execution of computational cycles. Various parameters of the computational system can be adjusted by the user.

a) Population initialization

An initially empty list \mathbf{P} is defined and calls to the root formula are made. For each call, a GL is constructed by a "reconstruction" of the empty list (see § 3.1.2), and a triplet $(\mathbf{g}, \mathbf{f}, \mathbf{s})$ is defined, where \mathbf{g} is

the GL, \mathbf{f} is the fitness list of \mathbf{g} , and \mathbf{s} is a magnitude with initial value $\mathbf{1}$. The triplet will be referred to as a “species”, while \mathbf{s} will be referred to as the “species size”. Informally, a species represents a set of identical individuals, while the “species size” represents the size of the set. If the GL of a new species is identical to the GL of a species already existing in \mathbf{P} , then the “size” of the existing species increases by $\mathbf{1}$ while the size of the population remains intact. Otherwise the new species is appended to \mathbf{P} . As a consequence, no species having identical GLs exist in the population. Elements in \mathbf{P} are ordered by fitness, with the best-fitness element at the end. \mathbf{P} will be referred to as the “population”.

b) Execution of computational cycles

In each computational cycle :

1. the computational system creates three lists, each one associated to a different genetic operator, whose elements belong to the population. These lists will be referred to as the “selection lists”. The selection of the species forming a “selection list” is done randomly with respect to a probability density function applied to the population. Such functions will be referred to as the “selection functions”.
2. the GL of each species, within each “selection list”, is transformed by the respective genetic operator. A call to the root formula “reconstructs” (see § 3.1.2) the transformed list to a GL. The GL defines a new species referred to as a “test”.
3. tests form a “revision list”, in a best fitness ordering with the best fitness test at the end of the list.
4. for each test that includes a GL similar to the one included in a species of the population, the “size” of the latter species increases by $\mathbf{1}$. High-fitness tests from the remaining in the “revision list” form an “innovation list”, while low-fitness species of the population form an “extinction list”. The elements of the former list substitute the elements of the latter list in the population. If the user does not intervene to the definition of these lists (see § 5.3.2, 5.3.4) then the size of both lists is \mathbf{n} where \mathbf{n} is the largest number having the property that the \mathbf{n}^{th} in reverse order element of the “revision list” has better fitness than the \mathbf{n}^{th} element of the population.

The computational process ends either with the finding of an acceptable solution or with the transition of the population to a state where no species of better fitness can be produced.

4.3 Parameters of the computational system

The probability of finding an acceptable solution and the computation time are roughly analogous to the total number of tests performed. The search dynamics depend on parameters of the computational system, which can be adjusted before the beginning of, or during the computational process, affecting the expected computation time and the search localization. These parameters are summarized here:

1. The size of the population (see § 5.3.2).
2. The number of tests per computational cycle (see § 5.3.2).
3. The sizes of the “selection lists” (see § 5.3.3). These determine the computational method: e.g. a relative increase of the size of the “selection list” associated to the crossover, subtree mutation or Gaussian mutation operation would turn the computational method respectively to a GA, a stochastic search method or an evolutionary strategy.
4. The form of the “selection functions” (see § 5.3.2).
5. The definition of the “innovation list” (see § 5.3.4).

6. The definition of the “extinction list” (see § 5.3.4).
7. The desirable percentage, number or probability of the terminal nodes of a GL affected by each genetic operation (see § 5.3.5).
8. The distribution of the probability of a mutated node with respect to the depth in a GL, in subtree mutations and crossovers (see § 5.3.5).
9. The standard deviation of the probability density function in Gaussian mutations (see § 5.3.5).
10. The contribution of the “species size” to the selection probability of a species (see § 5.3.2: “**Species Factor**”).

5 The environment of GENETICA

5.1 Menu “File”: projects and file management

Three types of files are involved in GENETICA projects: GENETICA source files, which are ASCII files having extension **gen**, list files having extension **lst** and project files having extension **prj**. A project file registers:

- the values of the parameters of the computational system
- the pathname of a source file (optional)
- the pathname of an input list file (optional), which includes the input vector for the root formula of the source file
- the pathname of a population file (optional), which is a list file including a population compatible with the root formula of the source file, as defined in § 4.2.2.a, and information about the history of the population evolution.

The options **New**, **Open**, **Save** and **Save as** refer to project files. The option **Set Input** allows the definition of the pathname of the current project’s input list file which should be an existing file. When the pathname is defined, the content of the file is immediately defined to be the input list. The option **Set Output** is inactive. The option **Set Population** provides the suboptions **Source** and **Target**. The first one allows the definition of the population of the current project from an existing population file, whereas the second one allows the definition of the pathname of a population file which is to be created (or revised) when the current project will be saved. The population file is saved (or revised) each time the current project is saved. The current project cannot be saved if it includes a population while the pathname of a population file has not been defined.

5.2 Menu “Compile/Test”: Compilation and test execution

The option **Compile** generates executable code, given the source code of the current project. Compilation is possible only if the source code file of the current project has been specified. The executable code is registered only in the heap memory (not in a file). During the compilation process the syntax of the source code is checked. If there are errors in the source code the first error is reported in a message box. Otherwise the box presents the message “**Successful Compilation**” which indicates that the executable code has been created. The compilation is based directly on the content of the source code file, since GENETICA’s environment does not include a built-in text editor. As a consequence, if a compilation error occurs, the user has to revise the source code file and then select the option **Compile** again.

The option **Execute**, which works only if both the source code has been compiled and the input list has been defined, calls the root formula with the GL resulting from a “reconstruction” (see § 3.1.2) of a list referred to as the “gene structure”. The “gene structure”, which can be defined by the user, is by default an empty list. When the call has been executed, the output term of the call is presented as the only element of a list named “**Root**” in a dialog box named “**List Editor**”. The List Editor (see § 5.5.1) is a general tool for editing lists and opening or saving list files, which can be activated with the option **Edit List**.

The option **Edit Input** allows the definition of the input list in the List Editor. If the input list has been already defined then it is presented in the List Editor while redefinition is allowed. However the input list file is not affected when the current project is saved after an input list redefinition.

The option **Edit GL-Fitness** presents the GL and the fitness list of the last call to the root formula performed via the option **Execute**. The lists are presented in this order as a two-element list in the List Editor. The option **Edit Gene Structure** allows redefinition of the “gene structure” in the List Editor.

The options **GaussDef** and **SigmDef** activate an editor of Gaussian functions and an editor of sigmoid functions respectively (see § 5.5.2). These options do not affect the current project.

5.3 Menu “Control”: Parameters of the computational process

5.3.1 The option **Basic Control**

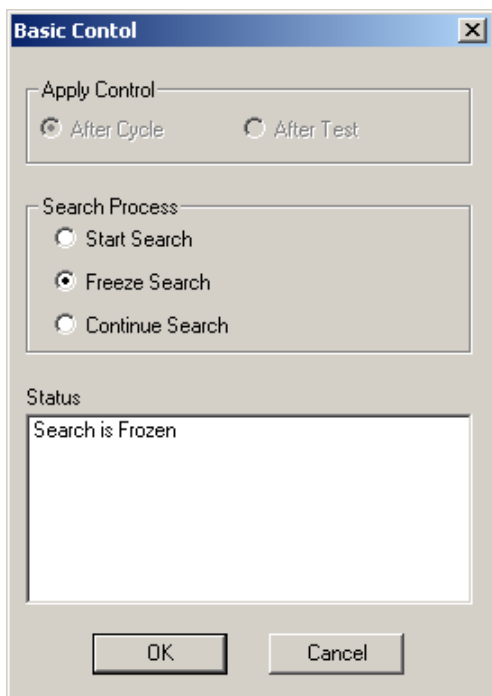


Figure 1

The option **Basic Control** presents a dialog titled “**Basic Control**” (Figure 1) that allows the user to start a new computational process, to freeze (temporarily stop) a computational process which is in progress or to let a frozen computational process to be continued. These options are presented in the area titled “**Search Process**”, while a message declaring the current status of the search is presented in the area titled “**Status**”.

If the dialog is called while no computational process has been performed yet within the current project, the message “**New Search is to Start**” is presented in the “**Status**” area, while **Start Search** and **Freeze Search** are the only available options in the area “**Search Process**”. If the **Start Search** option is active, a new computational process starts when the **OK** button is pressed. In this case, if the source code file has not been specified or the input list has not been defined, the respective error message is presented, while if the source code has not been compiled then compilation is performed automatically. The **Freeze Search** option has no effect if the **OK** button is pressed under the “**New Search is to Start**” status.

If the dialog is called when the computational process is in progress then the message “**Search in Progress**” is presented in the “**Status**” area, while **Freeze Search** is the only available option in the “**Search Process**” area. When the **OK** button is pressed the computational process is programmed to freeze at the end of the current computational cycle. If the dialog is called while the computational process is programmed to freeze the message “**Search is to be Frozen**” is presented in the “**Status**” area. When the

computational process is frozen the user is allowed either to save the project through the options of the **File** menu or redefine the values of the parameters of the computational system through the options of the **Control** menu.

If the dialog is called while the computational process is frozen then the message “**Search is Frozen**” is presented in the “**Status**” area. If the **Start Search** option is active when the **OK** button is pressed then a new computational process starts with the current settings. If the **Continue Search** option is active when the **OK** button is pressed then the current computational process is continued with the current settings.

Note that the option **Basic Control** is the only option which is safe to use when the computational process is in progress. Any other option within GENETICA’s environment should be used only when the computational process is frozen, otherwise a fatal error may occur.

5.3.2 The option **Search Basics**

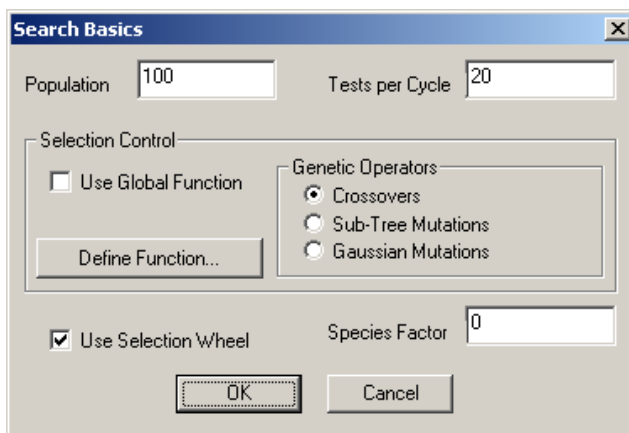


Figure 2

The option **Search Basics** presents a dialog titled “**Search Basics**” (Figure 2). The purpose of this dialog is to control the size of the population, the number of the tests performed per computational cycle and the selection of the elements of the selection lists (see § 4.2.2.b.1).

The number of the elements of the population and the number of tests per computational cycle are declared in the edit boxes titled “**Population**” and “**Tests per Cycle**” respectively. If the user reduces the size of the population from n_1 to n_2 (where $n_1 > n_2$) during the search process then the

first $n_1 - n_2$ elements of the population (which are the worse-fitness elements) will be removed. If the user increases the size of the population from n_1 to n_2 (where $n_1 < n_2$) during the search process then the size of the “extinction list” (see § 4.2.2.b.4), which otherwise is equal to the size of the “innovation list”, will be reduced by $n_2 - n_1$ for the next computational cycle. If the original size of the “extinction list” is less than $n_2 - n_1$ then the “extinction list” will be defined to be empty for as many computational cycles as necessary so that the population will be finally reduced to n_2 elements.

The area titled “**Selection Control**” concerns the selection functions (see § 4.2.2.b.1). The selection functions are sigmoid functions defined with the Sigmoid Function Editor (see § 5.5.2) which is called with the button titled “**Define Function**”. If the check box titled “**Use Global Function**” is checked then the selection function defined through the “**Define Function**” button refers to all the selection lists (see § 4.2.2.b.1). Otherwise it refers only to the selection list associated to the genetic operator selected in the area titled “**Genetic Operators**”.

The check box titled “**Use Selection Wheel**” affects the selection method. If it is not checked then each selection is realised randomly, otherwise the first selection is realised randomly while the remaining selections are defined by equal probability intervals. In both cases the selection probability density is specified by the respective selection function.

The selection probability of a species in the population is multiplied by s^x , where s is the “species size” (see § 4.2.2.a) and x is the real number written in the edit box titled “**Species Factor**”.

5.3.3 The option **GO Contribution**

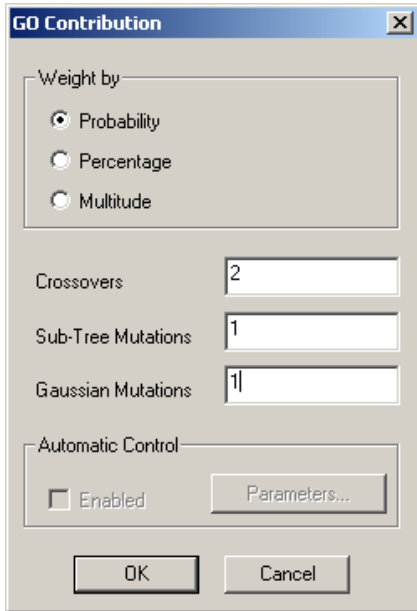


Figure 3

The option **GO Contribution**, where **GO** stands for “Genetic Operator”, presents a dialog titled “**GO Contribution**” (Figure 3). This dialog controls the sizes of the selection lists (see § 4.2.2.b.1), i.e. the number of tests generated by each genetic operator.

Each genetic operator is assigned a weight in the corresponding edit box titled “**Crossovers**”, “**Sub-Tree Mutations**” or “**Gaussian Mutations**”. The interpretation of the weights depends on the active option in the area titled “**Weight by**”: the options **Probability**, **Percentage** and **Multitude** denote relative probability, relative number and absolute number of tests per genetic operator respectively. If the **Multitude** option is active then the number of tests per computational cycle (see § 5.3.2) is redefined to be equal to the sum of the values written in the three edit boxes.

The area **Automatic Control** is inactive.

5.3.4 The option **Innovation/Extinction**

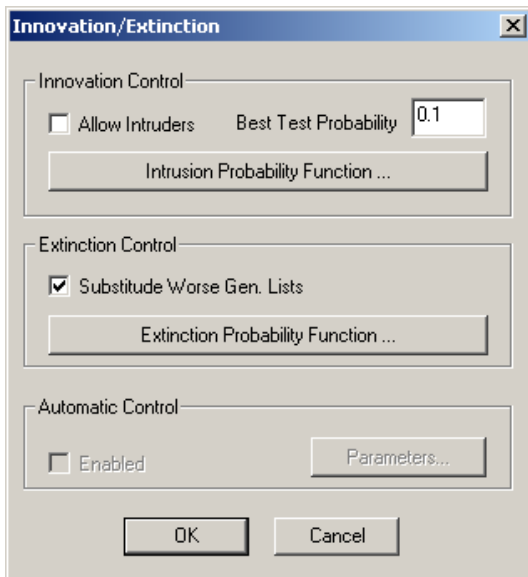


Figure 4

The option **Innovation/Extinction** presents a dialog titled “**Innovation/Extinction**” (Figure 4). The purpose of this dialog is to control the definition of both the “innovation list” and the “extinction list” (see § 4.2.2.b.4).

If the check box **Allow Intruders** in the area titled “**Innovation Control**” is not checked then the “innovation list” consists of the last **n** elements of the “revision list” (see § 4.2.2.b.3), where **n** is the size of the “extinction list” (see § 4.2.2.b.4, § 5.3.2). If the check box **Allow Intruders** is checked then additional elements of the “revision list” are allowed to become members of the “innovation list”. These elements, which are called “intruders”, are selected by a sigmoid probability density function applied to the “revision list” (see § 4.2.1). This function can be defined through the button **Intrusion Probability Function** which calls the Sigmoid Function Editor

(see § 5.5.2). The size of the “extinction” list is increased by the number of the “intruders”. The probability of each element, name it **e**, of the “revision list” to be an “intruder” is $\frac{\mathbf{P} \cdot \mathbf{E}}{\mathbf{P}_{\text{best}}}$ where **P** and **P_{best}** are respectively the probability assigned to **e** and the probability assigned to the last element of the “revision

list” by the intruders’ probability density function, while **E** is the real value written in the edit box **Best Test Probability**, where $E \in (0, 1]$.

If the check box **Substitute Worse Gen. Lists** in the area titled “**Extinction Control**” is checked then the “extinction list” consists of the first **n** elements of the population, where **n** is the size of the “extinction list”. Otherwise the “extinction list” consists of **n** elements selected by a sigmoid probability density function applied to the population (see § 4.2.1). This function can be defined through the button **Extinction Probability Function** which calls the Sigmoid Function Editor.

5.3.5 The options **Crossover Settings**, **ST-Mut. Settings** and **Gauss Mut. Settings**

Each one of these options presents the dialog titled “**Genetic Operator Settings**” (Figure 5). The purpose of this dialog is to adjust the parameters of each genetic operator. The options **Crossover Settings**, **ST-Mut. Settings** and **Gauss Mut. Settings** refer to the crossover, the subtree mutation and the Gaussian mutation genetic operator respectively.

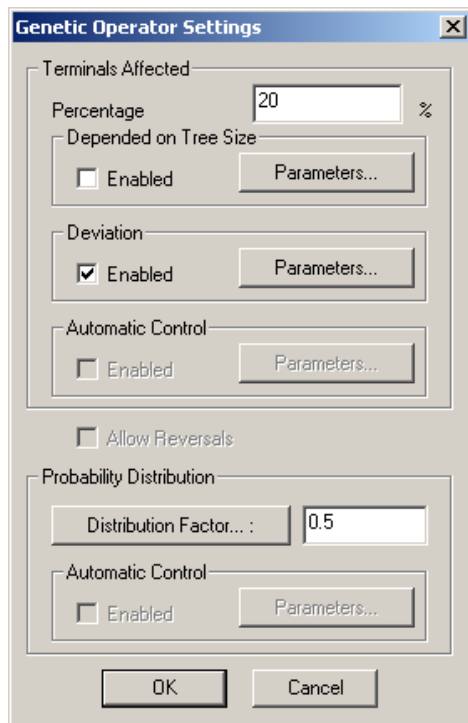


Figure 5

Distribution Factor in the area titled “**Probability Distribution**” defines a probability density function whose functionality depends on the genetic operator. In the case of a subtree mutation (respectively a crossover) the function defines the probability of each node of the GL (respectively the “common region” of the GLs) to be the target node of the genetic operation. Lower values define low probability for the low-depth nodes whereas higher values have the opposite effect. The value **0** defines equal probability for all the nodes while the value **1** defines the probability of a node **N** to be equal to the sum of the probabilities of all the other nodes of the subtree rooted at **N**. In the case of a Gaussian mutation the “distribution factor” value represents the standard deviation of the Gaussian probability density functions affecting the terminals of a GL (see § 4.1.3) and can be defined also through the button **Distribution Factor** which activates the Gaussian Function Editor. The sub-area titled “**Automatic Control**” is inactive.

The edit box **Percentage** in the area titled “**Terminals Affected**” declares the desirable percentage of the terminals of a GL, in the case of a subtree or a Gaussian mutation, or the terminals of the “common region” (see § 4.1.1) of two GLs, in the case of a crossover, which should be affected by the genetic operation. Here, only the GL’s terminals that include real values are considered. A terminal is considered to be affected by a genetic operation if its value is redefined because of the operation, even if the terminal is not the target node of the operation. If the check box **Enabled** in the sub-area titled “**Deviation**” is checked then the desirable percentage depends on a Gaussian probability density function defined in the interval **[0, 100]**, maximized at the value written in the edit box **Percentage**. This function can be defined through the button **Parameters**, in the “**Deviation**” sub-area, which calls the Gaussian Function Editor (see § 5.5.2). The sub-areas titled “**Depended on Tree Size**” and “**Automatic Control**” are inactive.

The check box **Allow Reversals** is inactive.

The value in the edit box next to the button

The algorithm which realizes a crossover or a subtree mutation is described here:

Let **k** be the number of the terminals of a GL (or the “common region” of two GLs) that corresponds to the desirable percentage of terminals affected by the operation.

1. Define the number **n** of the terminals affected by the genetic operation to be equal to **0**.
2. Define the target node of the operation. Let **m** be the number of the terminals of the subtree rooted at the target node.
3. If the step 2 has been executed only once then apply the genetic operation. Otherwise apply it only if $|n + m - k| < |k - n|$.
4. If the genetic operation has been applied in the step 3 then increase **n** by **m** and repeat the process from step 2, otherwise stop.

5.4 Menu “View”: Presentation of the computational process

5.4.1 The option **Population**

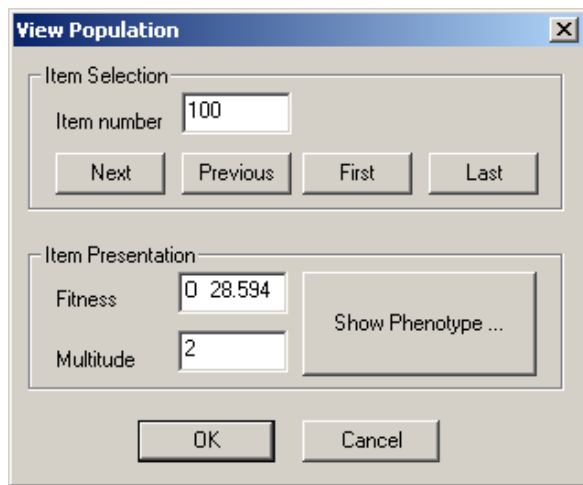


Figure 6

buttons **Next**, **Previous**, **First** and **Last** which respectively refer to the element next to the current species, the element previous to the current species, the first element of the population and the last element of the population.

The edit boxes **Fitness** and **Multitude** in the area titled “**Item Presentation**” present the fitness and the “size” (see § 4.2.2.a) of the current species respectively. In the case of a confirmation problem the fitness is represented by the first element of the fitness list (see § 3.2) having the prefix **F** which stands for “Find Solution”. The same notation is used in optimization problems if the formula constructing a potential solution (see § 2.3.1.b, connective **opt**) has not been confirmed. Otherwise, if the formula performing evaluation of a potential solution has not been confirmed, the fitness is represented by the first element of the fitness list having the prefix **E** which stands for “Evaluate Solution”. If both the solution construction formula and the solution evaluation formula have been confirmed then the fitness is represented by the evaluation magnitude having the prefix **O** which stands for “Optimize Solution”.

The button **Show Phenotype** in the area titled “**Item Presentation**” calls the List Editor (see § 5.5.1) where the output term of the root formula call defined by the GL of the current species is presented as the only element of the list named “**Root**”.

The option **Population**, which should be selected only while the computational process is frozen (see § 5.3.1), presents a dialog titled “**View Population**” (Figure 6). This dialog presents information that concerns the population and the individual species within the population.

The integer number in the edit box **Item Number** in the area titled “**Item Selection**” specifies the order of a species within the population. This species will be called the “current species”. The user can define any species within the population to be the current species either by writing a number in the edit box **Item Number** or by pressing one of the

5.4.2 The option **Best Solution**

The option **Best Solution** calls the List Editor (see § 5.5.1) where the best solution (i.e. the output term of a call to the root formula) found so far throughout the whole computational process is presented as the only element of the list named “**Root**”. This is equivalent to the result of the **Show Phenotype** button of the **View Population** dialog while the “current species” is the last element of the population (see § 5.4.1).

5.4.3 The option **Search Diagrams**

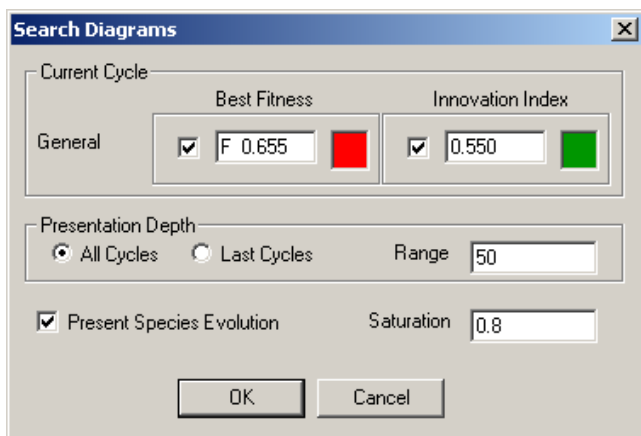


Figure 7

computational cycle. Fitness is presented in the notation introduced in 5.4.1. Each sub-area includes a check box and a coloured button. If the check box is checked then the respective magnitude is to be presented in the main window graphically as a function of time, where time is expressed in computational cycles. The colour of the function’s curve can be defined through the coloured button.

If the check box **Present Species Evolution** is checked then the “evolution diagram” is to be presented in the main window. The “evolution diagram” consists of coloured zones, each one representing a species of the population (see § 4.2.2.a). Species are ordered by fitness, with the best fitness species at the top of the diagram while the horizontal axis represents time in computational cycles. The thickness of each zone represents the relative “species size” (see § 4.2.2.a). Critical innovations (i.e. new best-fitness species) emerge at the top of the diagram, while extinction takes place at the bottom of the diagram. The saturation of the colours of the evolution diagram can be controlled through the edit box **Saturation** so that the evolution diagram could be clearly presented in combination with the best fitness function and the innovation index function. A saturation of **1** indicates full colour intension whereas a saturation of **0** indicates completely desaturated colours turned to white.

The area titled “**Presentation Depth**” includes the options **All Cycles** and **Last Cycles** and the edit box **Range**. If the option **All Cycles** is active then the horizontal axis of the best fitness function, the innovation index function and the evolution diagram includes all the computational cycles performed from the beginning of the search process. If the option **Last Cycles** is active then the horizontal axis includes no more than the last **n** cycles, where **n** is the integer value written in the edit box **Range**.

The presentation configured through the dialog box titled “**Search Diagrams**” is activated immediately with the button **OK**. When the computational process is continued the presentation of the evolution diagram is automatically suppressed.

The option **Search Diagrams**, which can be selected when the computational process is frozen, presents a dialog titled “**Search Diagrams**” (Figure 7). This dialog presents information concerning the current computational cycle and controls the graphic presentation of the computational process in the main window.

The sub-areas titled “**Best Fitness**” and “**Innovation Index**”, within the area titled “**Current Cycle**”, respectively present the best fitness encountered and the percentage of the tests that introduced new species in the population within the current

An example of the appearance of the best fitness function, the innovation index function and the evolution diagram in the main window, is presented in Figure 11 in § 5.6.

5.4.4 The option **Project Info**

The option **Project Info** presents the pathnames of the source file, the input list file and the population file of the current project (see § 5.1).

5.4.5 The option **Memory**

The option **Memory** presents the memory (in bytes) occupied by the basic elements of the current project. The indication **Lost Memory**, appearing last in the presentation, should be always **0**, otherwise a critical failure of the computational process occurs. The **Lost Memory** should be checked after each test of the code execution (see § 5.2: option **Execute**) because the bad use of GENETICA's pointers may result to lost memory even if there is no syntax error in the code.

5.5 List Editor and Function Editors

5.5.1 The List Editor

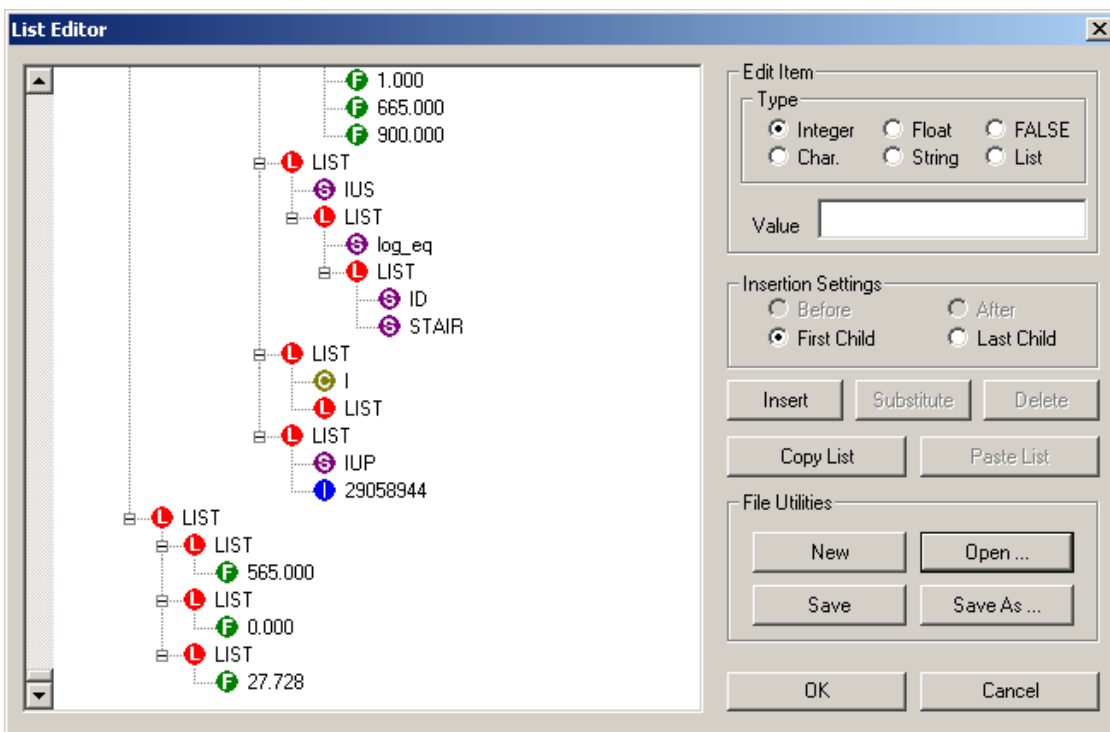


Figure 8

The List Editor is a general tool for editing lists and opening or saving list files. It appears as a dialog titled “**List Editor**” (Figure 8).

A list is presented at the left part of the dialog as a tree structure where inclusion relations are interpreted as parent-child relations. Each item of the tree structure represents a term and appears with the

type indication character (see § 2.1) as a capital in a coloured circle and the value of the term. The value **FALSE** (see § 2.1) appears as a black circle with the indication “**FALSE**”. The root node of the tree structure, which is always a list, is followed by the indication “**Root**”.

A click on an item of the tree structure makes the item current. A double click makes the item current and adjusts all the elements of the area titled “**Edit Item**” with respect to the current item. The area titled “**Edit Item**” includes six options representing the term types, in the sub-area titled “**Type**”, and the edit box **Value** where the value of an item can be written. If the option **List** or **FALSE** is active in the sub-area titled “**Type**” then the words “**LIST**” or “**FALSE**” respectively are written in the edit box which becomes inactive.

The user can define an item by adjusting the content of the area “**Edit Item**” and then insert this item in the tree structure by pressing the button **Insert** or substitute the current item by pressing the button **Substitute**. In the first case, the position of the new item in the tree structure is determined, with respect to the current item, by the active option in the area titled “**Insertion Settings**”. The options **Before** and **After**, which are available only if the current item is not the “**Root**” item, indicate that the new item will be inserted just before or just after the current item respectively in the list that includes the current item as an element. The options **First Child** and **Last Child**, which are available only if the current item is a list, indicate that the new item will be inserted as the first or as the last element of the current item respectively. The button **Delete** removes the current item from the tree structure.

The button **Copy List**, which is available only if the current item is a list, copies the current item in the List Editor’s clipboard. The content of this clipboard is available to the same or any other use of the List Editor within the same project and the same GENETICA session. The button **Paste List**, which is available only if the List Editor’s clipboard is not empty, inserts the content of this clipboard in the tree structure. The insertion position depends both on the current item and the active option of the area “**Insertion Settings**”.

The button **New** in the area titled “**File Utilities**” resets the tree structure as an empty list. The button **Open** in the same area sets the tree structure to be the content of a list file (see § 5.1: files having extension **lst**), while the buttons **Save** and **Save As** register the tree structure as a list file.

5.5.2 The Function Editors

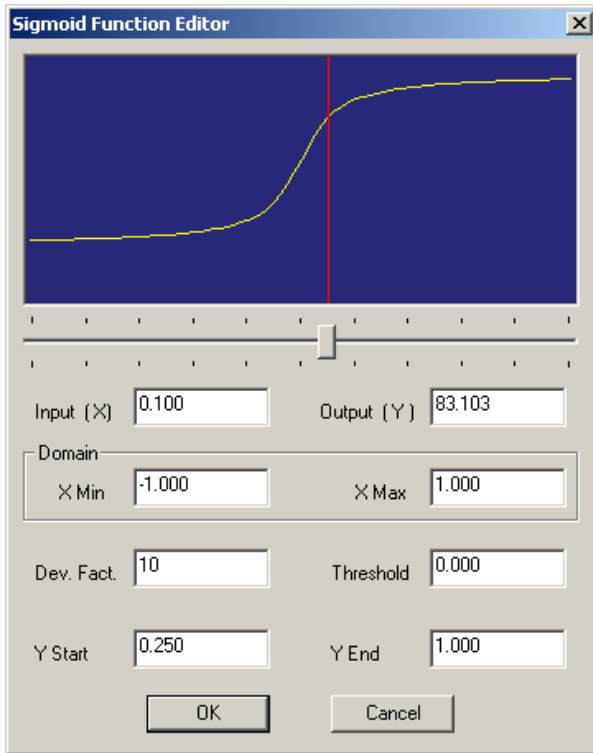


Figure 9

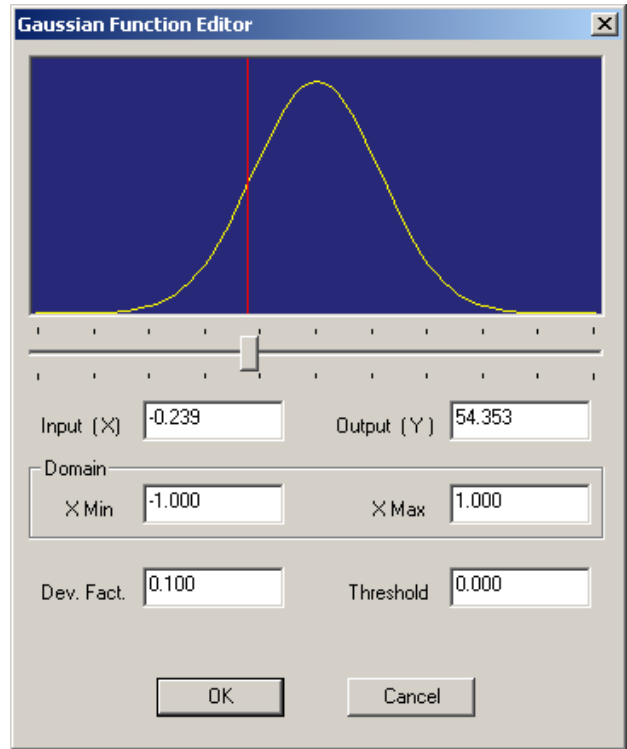


Figure 10

The Sigmoid Function Editor and the Gaussian Function Editor appear as dialogs titled “**Sigmoid Function Editor**” (Figure 9) and “**Gaussian Function Editor**” (Figure 10) respectively. Both dialogs include a graphic area presenting the function to be edited: the horizontal axis represents the domain of the function while the vertical axis represents the range of the function. The domain is specified by its minimum and maximum value appearing in the edit boxes **X Min** and **X Max** in the area titled “**Domain**”. The domain definition serves only presentation purposes and has no effect in the function definition. The user can change the indicative input value of the function either by writing a value in the edit box **Input (X)** or by using the slide bar under the graphic area. This changes the indication of the output value of the function in the edit box **Output (Y)**. Output values are scaled so that the maximum output value in the specified domain is **100**. Indicative input values should be limited in the specified domain.

The function definition depends on the values in the edit boxes **Dev. Fact.** (which stands for “Deviation Factor”), **Threshold**, **Y Start** and **Y End**. The last two edit boxes, appearing only in the “**Sigmoid Function Editor**” dialog box, define the output values (before scaling) of a sigmoid function at the limits of the specified domain.

Let **d**, **t**, **y_s** and **y_e** be the values in the aforementioned edit boxes respectively. A sigmoid function is

defined by the expression $f(x) = y_s + (y_e - y_s) \cdot \frac{\frac{\pi}{2} + \arctan((x - t) \cdot d)}{\pi}$ while a Gaussian function is

defined by the expression $f(x) = \frac{1}{e^{\frac{(x-t)^2}{d}}}$.

5.6 The main window

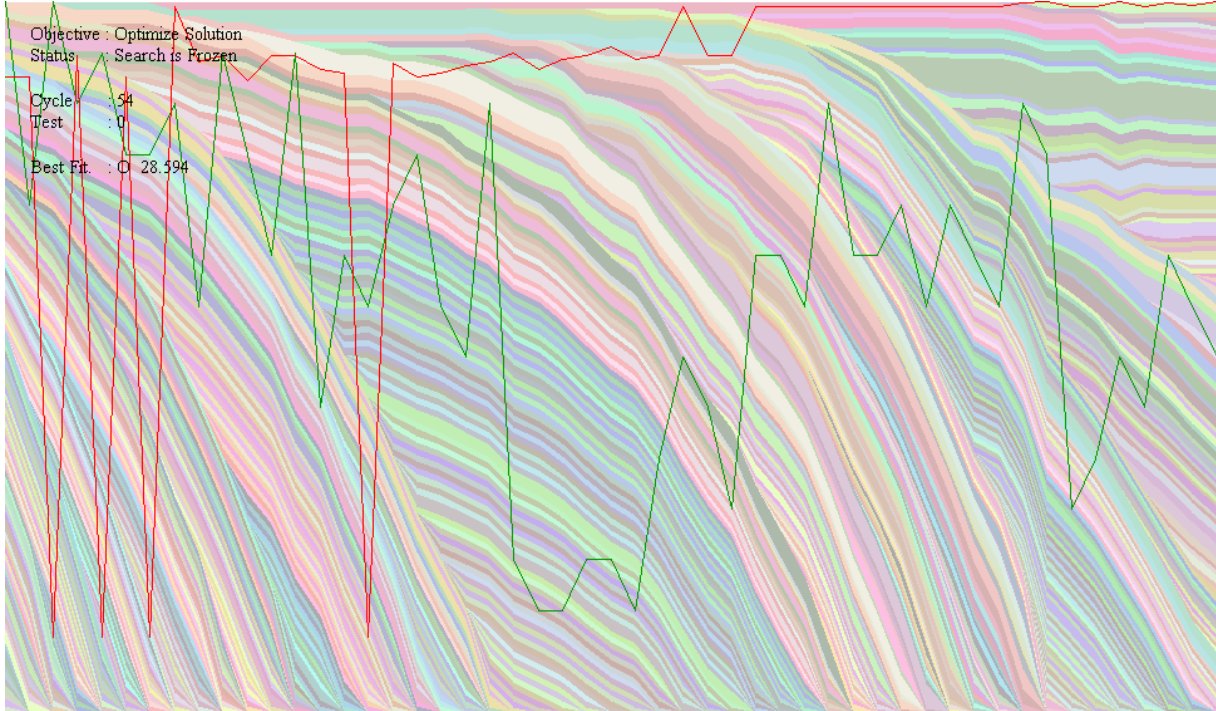


Figure 11

The main window presents both text and graphic information concerning the status of the computational process. Text information is presented under the following titles:

Objective

In the case of a confirmation problem the phrase “**Find Solution**” follows the title **Objective**. The same happens in the case of an optimization problem if the formula producing a potential solution (see § 2.3.1.b, connective **opt**) has not been confirmed. If this formula has been confirmed but the formula performing evaluation of a potential solution has not been confirmed then the phrase “**Evaluate Solution**” follows the title **Objective**. If the latter formula has been confirmed then the phrase “**Optimize Solution**” follows the title **Objective**.

Status

The current status of the computational process, described by the phrase “**New Search is to Start**”, “**Search in Progress**”, “**Search is to be Frozen**” or “**Search is Frozen**”, follows the title **Status**, under the situations described for the “**Status**” area of the dialog box titled “**Basic Control**” (see § 5.3.1).

Cycle

It is followed by the order of the current computational cycle.

Test

It is followed by the order of the current test (root formula call) performed either during the population initialization phase (see § 4.2.2.a) or during the current computational cycle.

Best Fit.

It is followed by the best fitness found so far throughout the whole computational process, in the notation presented in 5.4.1 for the **Fitness** edit box of the **View Population** dialog.

Graphic information includes the presentation of the best fitness function, the innovation index function and the evolution diagram with respect to the definitions in the **Search Diagrams** dialog box (see § 5.4.3). The bottom edge of the window is considered as the X axis while the left edge is considered as the Y axis of both the diagram and the functions. Both the diagram and the functions are scaled in both axes to fit the size of the main window.

An example of the content of the main window is presented in Figure 11.

6 Tutorial*6.1 Application 1: Find Hamilton cycles in graphs*

6.1.1 Problem statement and problem solving method

The aim of the application is to find Hamilton cycles in graphs. A Hamilton cycle in a graph \mathbf{G} is a cyclic sub-graph that includes all the nodes of \mathbf{G} .

\mathbf{G} is represented as a list of nodes. Each node is represented as a list having the form $(\mathbf{k}, (\mathbf{k}_1, \dots, \mathbf{k}_n))$ where \mathbf{k} is an integer that represents the name of the node while \mathbf{k}_i ($i = 1 \dots n$) are integers that represent the names of the nodes of \mathbf{G} that are connected to the node named \mathbf{k} . A path in \mathbf{G} is represented as a list of node names, where successive elements in the list represent nodes connected in \mathbf{G} .

Problem solving is based on the construction of a path in \mathbf{G} and the confirmation of the assumption that this path constitutes a Hamilton cycle. The path is initially defined as a one-element list, whose single element is the name of the first node in \mathbf{G} . Then the path increases by the successive appending of node names at the end of the list.

We introduce two properties required for the nodes that are to be appended to the path: “potential connection” and “valid connection” of the last node represented in the path. A “potential connection” is the name \mathbf{v} of a node, if one of the following conditions is satisfied:

- a) \mathbf{v} does not belong to the path
- b) \mathbf{v} represents the first node of the path, while the path includes all the nodes of \mathbf{G} (in this case the path is a Hamilton cycle).

A “potential connection” will be referred to as a “valid connection” if it represents a node connected (in \mathbf{G}) to the last node represented in the path. The path increases by the successive appending of “valid connections”.

6.1.2 Presentation of the GENETICA code

The source GENETICA code for this application, included in the file **Hamilton_Cycles.gen**, is presented here. Within the code the term **l0** represents the empty list, while the graph and the path are represented by the lists **G** and **Path**, having sizes **G_size** and **P_size** respectively.

! ----- HAMILTON CYCLE ---- !

! Returns a Hamilton Cycle of the Graph G as a list of node names.

Standard input values:

l0 = ()

init_call = ("HML_make_path") !

```
HML_H_Cycle      and
T      (G      l0      init_call)
      (crd      G_size      (G)
      add      l1      (init_call      G)
      add      l2      (l1      G_size)
      add      call_make_path      (l2      l0)
      at      Path      (call_make_path)
      HML_check_path      HC      (G_size      Path))
```

! Creates and returns a path.

Standard input value:

l0 = () !

```
HML_make_path      and
T      (G      G_size l0)
      (HML_init_path      P_1      (G      l0)
      HML_extend_path      Path      (G      G_size      G      G_size      P_1))
```

! This formula is confirmed if Path is a Hamilton Cycle of a Graph with G_size nodes. In the case of confirmation Path is returned without its last element (since this equals the first element) !

```
HML_check_path      and
T      (G_size      Path)
      (bl      P_bl      (Path)
      crd      P_size      (P_bl)
      eq      NULL      (G_size      P_size)
      cp      HC      (P_bl))
```

! Path is a one-element list that contains a node name of G. This formula returns Path increased by a number of names of nodes that constitute a path, such that there is no next “valid connection” of the last element of Path !

```
HML_extend_path      rec
T      (G      G_size      G      G_size      Path)
      (HML_not_next
      HML_incr_path)
```

! Path can not be increased, i.e. there do not exist “valid connections” for the last element of Path !

```
HML_not_next        not
NULL (G      G_size      Path)
      (HML_has_next      NULL      (G      G_size      Path))
```

! Path can be increased, i.e. there exist “valid connections” for the last element of Path !

```
HML_has_next        and
NULL (G      G_size      Path)
      (HML_valid_cons      T      (G      G_size      Path))
```

! Returns Path increased by a node-name that represents a “valid connection” for the last element of Path !

```
HML_incr_path        and
T      (G      G_size      Path)
      (HML_valid_cons      v_cons      (G      G_size      Path)
      mem      nnn      (v_cons)
      add      P_inc      (Path      nnn))
```

! Returns a one-element list that includes the first node name of the Graph G !

```
HML_init_path        and
T      (G      l0)
      (fi      n1      (G)
      fi      nn1      (n1)
      add      P_1      (l0      nn1))
```

! Returns the list of “valid connections” of the last element of the list Path. !

```
HML_valid_cons        and
T      (G      G_size      Path)
      (la      nn      (Path)
      pl      nn_con      (G      nn)
      crd      P_size      (Path)
      HML_sel_pot      v_cons      (nn_con      Path      G_size      P_size))
```

! Returns the “potential connections” for the last element of Path as the list of the elements of the list nn_con that confirm HML_ret_pot !

```
HML_sel_pot   chs
T   (nn_con   Path   G_size   P_size)
    (HML_ret_pot)
```

! Returns the node name nn if nn represents a “potential connection” for the Path !

```
HML_ret_pot   and
T   (nn   Path   G_size   P_size)
    (HML_pot_con   NULL   (nn   Path   G_size   P_size)
     cp   Tout   (nn))
```

! The node named nn is a “potential connection”, i.e. either it does not belong to the Path or it is the first node of Path while Path contains all the nodes of G !

```
HML_pot_con   or
NULL (nn   Path   G_size   P_size)
     (not_ism   NULL   (Path   nn)
      HML_close_cycle   NULL   (Path   G_size   P_size   nn))
```

! Path contains all the nodes of G (since G_size = P_size) and nn is the name of its first node !

```
HML_close_cycle   and
NULL (Path   G_size   P_size   nn)
     (eq   NULL   (G_size   P_size)
      fi   nn_1   (Path)
      eq   NULL   (nn   nn_1))
```

! The element nn is not contained in the list Path !

```
not_ism   not
NULL (Path   nn)
     (ism   NULL   (Path   nn))
```

Since the formulae appear in the code in descending semantic level order, it is easier to start the description of the code from the end.

- The condition (a) (see § 6.1.1) for a node named **nn** to be a “potential connection” of the path **Path** is represented by the formula **not_ism**, while the condition (b) is represented by the formula **HML_close_cycle**.
- The formula **HML_pot_con**, which is confirmed if and only if either the condition (a) or the condition (b) is confirmed, says that a node named **nn** constitutes a “potential connection” of the last node named in the path.
- The formula **HML_ret_pot**, which is logically equivalent to **HML_pot_con**, returns the node-name **nn**, which represents a “potential connection” of the last node named in the path, in the case of confirmation.
- The formula **HML_sel_pot** selects “potential connections” for the last node named in the path, from a node-name list **nn_con**. **HML_sel_pot** returns the list of the elements of **nn_con** that, confirming

HML_ret_pot, constitute “potential connections”.

- The formula **HML_valid_cons** defines **nn_con** as the list of the names of the nodes connected to the last node named in the path. Then selects (through **HML_sel_pot**) those constituting “valid connections” for the latter node. The list of the “valid connections”, named **v_cons**, is returned.
- The path is initialized by the formula **HML_init_path**, as a one-element list containing the name of the first node of **G**.
- The formula **HML_incr_path** selects the name of a “valid connection” for the last node named in the path, then appends it at the end of the path and returns the increased path.
- The confirmation of **HML_has_next** indicates that the path can be increased, i.e. there exist “valid connections” for the last node named in the path (formula **HML_valid_cons**). The formula **HML_not_next** is confirmed in the opposite case.
- The formula **HML_extend_path** extends the path by successively appending node names (via recursive calls to **HML_incr_path**) until no further appending can be made (i.e. **HML_not_next** is confirmed).
- The full job of the path construction is performed by **HML_make_path**, which initializes the path by calling **HML_init_path** and then extends it by calling **HML_extend_path**.
- If the path is a Hamilton cycle in **G**, then it includes the names of all the nodes of **G**, while its first node-name has been appended at the end of the path (then the path size is **G_size + 1**). This is confirmed by the formula **HML_check_path** which removes the last node-name from the path while it is confirmed if the size of the reduced path is **G_size**. In the case of confirmation, **HML_check_path** returns the reduced path which is a Hamilton Cycle.
- The root formula is the formula **HML_H_Cycle** which creates a “call list” (see § 2.1), named **call_make_path**, to **HML_make_path**, by successively appending the terms **G**, **G_size** and **10** to the list **init_call** whose initial value is (“**HML_make_path**”), and then calls it via **at**. The call returns the list **Path**, which represents the path. Then **HML_check_path** is called, which both confirms that **Path** represents a Hamilton cycle and returns the solution. The fitness of the **HML_H_Cycle** call equals the fitness of the **HML_check_path** call, since the atomic formulae **crd**, **add** and **at** are “always true” (see § 3.2.1.d).

The critical point of this algorithm is the way the path is increased (formula **HML_incr_path**). The selection of the “valid connections” to be appended at the end of the path is performed randomly by the atomic formula **mem** referenced from **HML_incr_path**. The computational system within GENETICA’s environment guides randomness in order to confirm the root formula **HML_H_Cycle**: the confirmation depends on the fitness of the call to the atomic formula **eq** within **HML_check_path**.

6.1.3 A case study

The Hamilton Cycles application will be tested in the graph presented in Figure 12. The files **proto.exe**, **Hamilton_Cycles.gen** and **G31.lst**. will be used. **proto.exe** is a prototype version of GENETICA’s programming environment, **Hamilton_Cycles.gen** is the source GENETICA code presented in 6.1.2, and **G31.lst** is a GENETICA list file that includes a list of input values for the root formula in the source code (i.e. the formula **HML_H_Cycle**). The first input value is a list that represents the 31-node graph presented in Figure 12.

Double-click on **proto.exe**. GENETICA’s programming environment appears in a window named **proto**. The indications **Objective : Find Solution** and **Status : New Search is to Start** appear at the upper left corner of the window. Open the **File** pull-down menu and select **Set Source**. Open the file **Hamilton_Cycles.gen**.

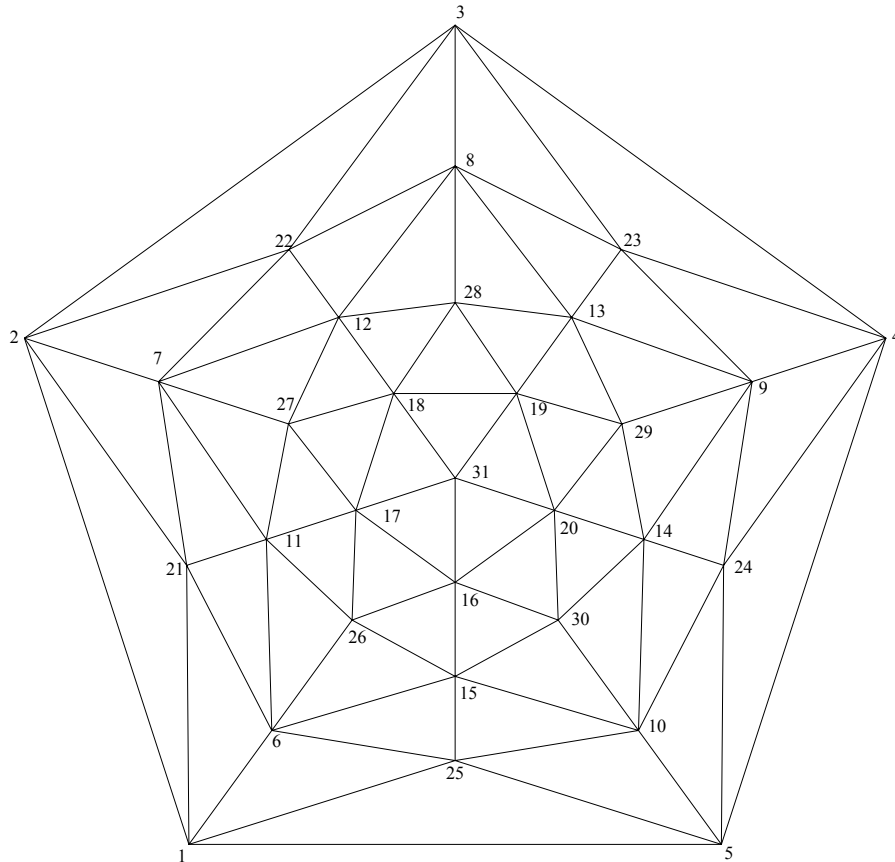


Figure 12

Open the **File** pull-down menu and select **Set Input**. Open the file **G31.lst**. You can see the input list by opening the **Compile/Test** pull-down menu and selecting **Edit Input**: the input list will appear in the List Editor. Close the List Editor by clicking **Cancel**.

Make the following changes to the default settings of the computational system:

- Open the **Control** pull-down menu and select **Search Basics**. Within the **Search Basics** dialog click the **Define Function** button. The Sigmoid Function Editor appears. Write **0.5** in the **Threshold** edit box. Click **OK** in the Editor and then **OK** in the **Search Basics** dialog.
- Open the **Control** pull-down menu and select **GO_Contribution**. The **GO_Contribution** dialog appears. Write **2** in the **Crossovers** edit box. Click **OK**.
- Open the **Control** pull-down menu and select **Crossovers Settings**. The **GO_Settings** dialog appears. Within the **Terminals Affected** area of the dialog, write **20** in the **Percentage** edit box. Click **OK**.
- Open the **Control** pull-down menu and select **Gauss. Mut. Settings**. The **GO_Settings** dialog appears. Within the **Terminals Affected** area of the dialog, write **10** in the **Percentage** edit box. Write **100** in the edit box within the area **Probability Distribution**. Click **OK**.
- Open the **View** pull-down menu and select **Search Diagrams**. The **Search Diagrams** dialog appears. Write **0.8** in the **Saturation** edit box. Click **OK**.

At this point you can save the project. Open the **File** pull-down menu and select either **Save** or **Save As**. Define the pathname of a file having extension **prj**. This file registers all the work you have done until now, including the definition of the pathnames of the source code and the input file. Whenever you want to

restore the current settings you only have to open this file by opening the **File** pull-down menu and selecting **Open**.

Open the **Compile/Test** pull-down menu and select **Compile**. The message **Successful Compilation** appears in a message window. Click **OK**. At this point the source code has been compiled and the computational process can start.

Open the **Control** pull-down menu and select **Basic Control**. The **Basic Control** dialog appears. Within the area **Search Process** of the dialog, select **Start Search**. Click **OK**.

The indication **Initializing Population** appears under the title **Status** at the upper left corner of the main window, while successively increasing numbers, indicating the order of the test currently performed during the population initialization phase, appear under the title **Test**. When the test counter reaches the population size, which is set to **100**, the indication under the title **Status** changes to **Search in Progress** while successively increasing numbers, indicating the order of the current computational cycle and the order of the current test within the current computational cycle, appear under the titles **Cycle** and **Test** respectively. At the same time the best fitness encountered so far appears under the title **Best Fit**, and it is revised each time a better fitness test occurs.

When the second computational cycle has been completed, the graphic representations of the best fitness function and the innovation index function (see § 5.4.3) appear in the main window, in red colour and green colour respectively. The graphic representations are revised at the end of each computational cycle. The graphic representations present the history of the whole computational procedure until the 50th computational cycle, while only the history of the last 50 cycles is presented (this is determined by the default settings in the **Search Diagrams** dialog).

The first time the root formula is confirmed (i.e. a Hamilton cycle has been found) the indication **F 0.000** appears under the title **Best Fit**, in the main window while the red line, representing the “best fitness per cycle” function, reaches the bottom of the window (i.e. the value **0**) at the end of the computational cycle. During the next computational cycles new confirmations occur, while the frequency of the confirmations increases with the computational cycles.

At about the 40th computational cycle open the **Control** pull-down menu and select **Basic Control**. The **Basic Control** dialog appears. Within the area **Search Process** of the dialog, select **Freeze Search**. Click **OK**. The current computational cycle is continued but the indication under the title **Status** changes to **Search is to be Frozen**. At the end of the computational cycle the computational process stops while the indication **Search is Frozen** appears under the title **Status**.

When the search is frozen, open the **View** pull-down menu and select **Search Diagrams**. The **Search Diagrams** dialog appears. Click on the check box **Present Species Evolution**. Click **OK**. The evolution diagram appears in the main window. Note the groups of parallel zones of species appearing in the diagram: each group represents calls to the root formula having the same “distance from confirmation” (see § 3.2). Groups showing lower “distance from confirmation” appear higher and to the right of the diagram. The group of the horizontal zones appearing at the upper right corner of the main window represents calls having a zero “distance from confirmation”, i.e. includes species that confirm the root formula and result to solutions (Hamilton cycles). Note the history of this group: the emergence of each species of the group coincides with a **0** fitness value indicated by the red line.

Open the **View** pull-down menu and select **Population**. The **View Population** dialog appears. Within the **Item Selection** area, the value **100** in the **Item** edit box indicates that the current species, presented within the **Item Presentation** area, is the 100th (i.e. the last) species of the population. Within the **Item Presentation** area, the value **F 0.000** in the **Fitness** edit box indicates that the current species confirms the root formula. Within the **Item Presentation** area click the button **Show Phenotype**. The solution specified by the current species appears in the List Editor as a list of node names. Verify that this

list constitutes a Hamilton cycle in the graph represented in the input list. Click **OK** in the List Editor. Within the **Item Selection** area click successively the button **Previous**. Each time you click, the value in the **Item** edit box is reduced by one, while the presentation within the **Item Presentation** area is revised. As far as the value **F 0.000** appears in the **Fitness** edit box the current species confirms the root formula, while you can see the solution specified by the current species by clicking on the button **Show Phenotype**. If the fitness value for the current species is **F** followed by a positive number then the current species disconfirms the root formula while clicking on the button **Show Phenotype** appears the value **FALSE** in the List Editor. Click **OK** in the **View Population** dialog.

Open the **File** pull-down menu. Note that the options **Save** and **Save As** are inactive: you can not save the current project until you specify a pathname for the population file that is to be created. Select **Set Population** and then **Target**. Write the name of a file and specify the pathname. This will be the population file. Select **Save As** and save the project with a different name. Both the current configuration of the parameters of the computational system and the current status of the search process will be registered in the new project file while the population file will be created.

Open the **Control** pull-down menu and select **Basic Control**. The **Basic Control** dialog appears. Within the area **Search Process** select **Continue Search**. Click **OK**. The computational process is continued while the indication under the title **Status** changes to **Search in Progress** and the evolution diagram disappears.

At about the 70th computational cycle the innovation index function, represented by the green line, is stabilized to zero. This indicates that no new species are introduced in the population any more.

Stop the search again by opening the **Basic Control** dialog and selecting **Freeze Search**. When the indication **Search is Frozen** appears under the title **Status** in the main window, open the **View** pull-down menu and select **Search Diagrams**. The **Search Diagrams** dialog appears. Within the **Presentation Depth** area select **All Cycles** (this will present the whole history of the computational process), activate the check box **Present Species Evolution** and then click **OK**. The evolution diagram appears again in the main window. Note that the group of the horizontal zones appearing at the upper right corner of the main window has been extended occupying now the whole vertical dimension of the main window. This indicates that the whole population consists of species that confirm the root formula and result to solutions. You can see these solutions (100 different Hamilton cycles) by opening the **View** pull-down menu, selecting **Population** and then using the button **Show Phenotype** for each species specified in the edit box **Item**.

While the computational process is frozen open the **Compile/Test** pull-down menu and select **Execute**. This calls the root formula with the GL resulting from a “reconstruction” (see § 3.1.2) of the “gene structure”, which is by default an empty list (see § 5.2). This is a random (non evolved) attempt to confirm the root formula. Unless you are really lucky the value **FALSE** will appear in the List Editor as the only element of the root list. This indicates that the root formula has been disconfirmed. In any case you can see both the GL and the fitness list of the call by opening the **Compile/Test** pull-down menu and selecting **Edit GL-Fitness**.

6.2 Application 2: Construct the control structure if-then-else

6.2.1 The aim of the application

The control structure **if-then-else** is common in many computer languages, but it is not available as a standard GENETICA expression. However it can be constructed in GENETICA as a high order formula (see § 2.1). The aim of this application is to construct this formula which could be used then in any GENETICA program. Specifically, the always true formula **ITE**—which stands for “If Then Else”—will

be constructed, having basic input terms the “call lists” (see § 2.1) **Alst**, **Blst** and **Clst** where **Alst** represents the confirmation condition while **Blst** and **Clst** refer to constructive formulae called in the case of confirmation and disconfirmation of **Alst** respectively. The output value of the latter call constitutes the output value of the **ITE** call.

6.2.2 Presentation of the GENETICA code

The source GENETICA code for this application, which included in the file **ITE.gen**, is presented here.

! Always true version of ITE_call_BC.

Standard input values: str = “ITE_call_BC”, l0 = () !

```
ITE    and
T      (str    Alst  Blst  Clst  l0)
      (ITE_at_inp      at_inp      (str    Alst  Blst  Clst  l0)
      at              T_out      (at_inp))
```

! Returns the call list to ITE_call_BC.

Standard input values: str = “ITE_call_BC”, l0 = () !

```
ITE_at_inp    and
T      (str    Alst  Blst  Clst  l0)
      (add  l1    (l0  str)
      add  l2    (l1  Alst)
      add  l3    (l2  Blst)
      add  l4    (l3  Clst))
```

! Returns the output term of the Blst call if the Alst call is confirmed. Otherwise returns the output term of the Blst call !

```
ITE_call_BC    and
T      (Alst  Blst  Clst)
      (ITE_BC_lst      BC_lst      (Alst      Blst  Clst)
      fi              BC          (BC_lst)
      call            BC_out      (BC))
```

! Returns (Blst Clst) if the Alst call is confirmed. Otherwise returns (Clst) !

```
ITE_BC_lst    or
T      (Alst  Blst  Clst)
      (ITE_ret_B  B    (Alst  Blst)
      cp          C    (Clst))
```

! Returns **Blst** if the **Alst** call is confirmed !

```

ITE_ret_B    and
T           (Alst  Blst)
           (call      NULL      (Alst)
           cp         B         (Blst))
    
```

- The formula **ITE_ret_B** is confirmed if and only if the **Alst** call is confirmed. In the case of confirmation the call list **Blst** is returned.
- The formula **ITE_BC_lst** is always confirmed since it is constructed with **or** while the second referenced formula (namely the formula **cp**) is always confirmed. Note (see also § 2.3.1.b: **con = or**) that **ITE_BC_lst** returns a list whose first element is the “call list” that is to be called: the list returned by **ITE_BC_lst** is either (**Blst, Clst**) or (**Clst**) depending on **Alst** confirmation.
- The formula **ITE_call_BC** calls **ITE_BC_lst** and gets the first element of the call’s output value. This element is the “call list” that is to be called. The latter call is realized by the formula **call** whose output value is finally returned by **ITE_call_BC**. **ITE_call_BC** is a full **if-then-else** formula, however it has not the “always true” property (defined in § 3.2.1.d). This property is achieved through the formulae **ITE_at_inp** and **ITE**. The former constructs a “call list” to **ITE_call_BC** while the latter executes the “call list”, through the formula **at**. **ITE** has the same functionality with **ITE_call_BC** but it has also the “always true” property since both referenced formulae (i.e. **ITE_at_inp** and **at** are “always true”).

6.2.3 Testing the **if-then-else** formula

Double-click on **proto.exe**. Open the **File** pull-down menu and select **Set Source**. Open the file **ITE.gen**.

Open the **File** pull-down menu and select **Set Input**. Open the file **ITE.lst**. You can check the input list by opening the **Compile/Test** pull-down menu and selecting **Edit Input**: the input list will appear in the List Editor. Note that the input list constitutes a valid input vector for the root formula, i.e. the formula **ITE**, where the “call lists” **Alst**, **Blst** and **Clst** are assigned the values (“**eq**”, **1**, **1**), (“**pls**”, **8.0**, **4.0**) and (“**mns**”, **8.0**, **4.0**) respectively. Close the List Editor by clicking **Cancel**.

Open the **Compile/Test** pull-down menu and select **Compile**. The message **Successful Compilation** appears in a message window. Click **OK**. Select **Execute** from the same menu. The List Editor appears the real value **12.0** as the only element of the root list. This is because the “call list” **Alst** has been confirmed and the “call list” **Blst** has been executed returning **8.0 + 4.0**.

Open the **Compile/Test** pull-down menu and select **Edit Input**. Change the list (“**eq**”, **1**, **1**) to (“**eq**”, **1**, **2**) in the List Editor (see § 5.5.1) and then click **OK**.

Open the **Compile/Test** pull-down menu and select **Execute**. The List Editor appears the real value **4.0** as the only element of the root list. This is because the “call list” **Alst** has been disconfirmed and the “call list” **Clst** has been executed returning **8.0 - 4.0**.

Open the **Compile/Test** pull-down menu and select **Edit GL-Fitness**. Note that the GL of the call to the root formula is the empty list. This is because the root formula, as any formula in the code, is a single confirmation one. Note that the fitness list is (**0**, **-1.0**) which indicates an “always true” formula (see § 3.2.1.d).

Do not try to activate the computational process. This will cause a fatal error since the search space is not defined for single confirmation root formulae (and this is only a prototype version).