libGE

Grammatical Evolution Library
for version 0.27alpha1, 14 September 2006

Miguel Nicolau
Darwin Slattery

This manual is for libGE, version 0.27alpha1, a C++ library that implements the Grammatical Evolution mapping process.

Table of Contents

1 Overview

1.1 Purpose

libGE is a C++ library that implements the Grammatical Evolution mapping process. This mapping process, associated with any kind of search algorithm, translates a string of objects onto a program to be evaluated.

Typically, libGE is used by an evolutionary computation algorithm, providing a mapping from a genotypic structure onto a phenotypic structure. On its default implementation, it maps a string provided by a variable-length genetic algorithm onto a syntactically-correct program, whose language is specified by a BNF (Backus-Naur Form) context-free grammar.

The data structures provided with libGE are higly configurable, so then can be easily used for different kinds of problems. A set of mappers is provided, which can be extended to comply with different demands (e.g. different grammar formats, different mapping processes, etc).

There are also several methods provided with libGE, to ease the integration of the library with most available evolutionary computation packages. These methods help to translate the data structures native to those packages into the structures handled by the libGE classes.

1.1.1 Compatibility

libGE has been developed under GNU/Linux (Mandrake 10.2, GCC v3.4.3). The GNU utils autoconf (http://www.gnu.org/software/autoconf/) and automake (http://www.gnu.org/software/automake/) have been used, to ensure a high level of compatibility with other *nix flavours.

1.1.2 Documentation

The documentation provided with libGE was made with the GNU Texinfo software (http://www.gnu.org/software/texinfo/). To render it in PS or PDF format, simply use `make ps` or `make pdf` after installing libGE; the resulting documentation file will be in the '`doc`' directory.

Please mail any suggestions to Miguel Nicolau (Miguel.Nicolau@gmail.com).

1.2 Installation

The libGE library was developped for linux systems, but should work on most Unix flavours. It comes bundled in a file called '`libGE-xxx.tar.gz`', where '`xxx`' is the latest version (currently 0.27alpha1). You should start by unpackaging this file, using the following command:

```
tar -xvzf libGE-0.27alpha1.tar.gz
```

This will create a directory called '`libGE-0.27alpha1`', containing all the files required to build the library. Within that directory, the file '`INSTALL`' contains generic installation instructions for projects that use the standard GNU packaging of autoconf and automake. In short, three steps are required:

```
./configure
make
make install
```

The `configure` script will try to guess certain hardware and software components of your system, and configure the source code of libGE accordingly. Run `./configure --help` for details on the options and arguments available. This will show you the standard options (common to all `configure` scripts), as well as libGE options, such as the inclusion or otherwise of support for specific search engines.

Running `make` will build the library, and `make install` will install the library (typically on '`/usr/local`', depending on your system). You might need superuser permissions to install the library on a system-wide directory; for a user-specific install, try `make install prefix=path`, where *`path`* is the path in which you which to install the library.

1.2.1 Examples

To configure and compile libGE using all the provided wrappers, and installing in directory '`/home/user/libGE`', the sequence of commands to use should be:

```
./configure
make
make install prefix=/home/user/libGE
```

To configure libGE without the GE_ILLIGALSGA wrapper, on the standard installation directory (usually '`/usr/local`'), and having the GALib headers installed in '`/home/user/galib246/ga`' and the GALib library in '`/home/user/galib246/libga.a`', the commands to use should be:

```
./configure --with-MITGALIBINCLUDES=/home/user/galib246/ga/ --with-
MITGALIBLIBS=/home/user/galib246/libga.a --enable-GE_ILLIGALSGA=no
make
make install
```

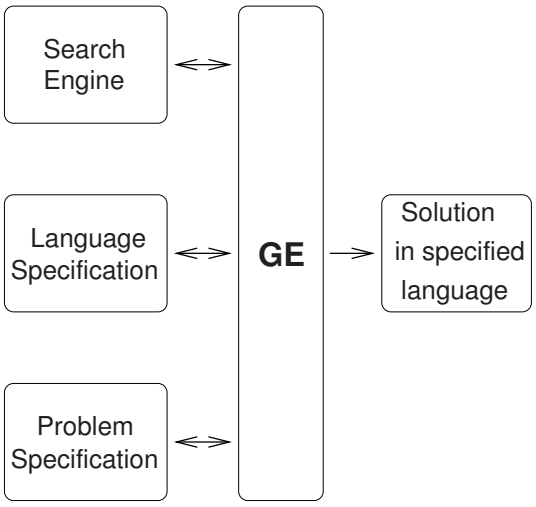
2 Introduction to libGE

2.1 Grammatical Evolution

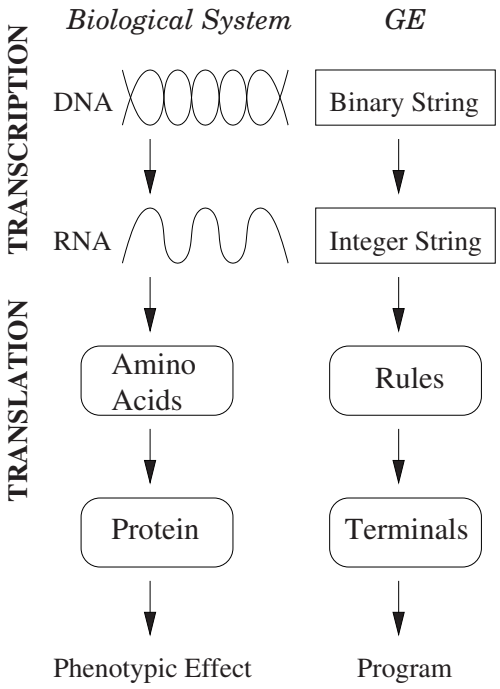
Grammatical Evolution (*O'Neill and Ryan, 2003*) (GE) is an evolutionary algorithm approach to automatic program generation, which evolves strings of binary values, and uses a grammar in BNF (Backus-Naur Form) notation to map those strings into programs, which can then be evaluated. The modular design behind GE means that any search algorithm can be used to evolve a population of binary strings, and after mapping each individual onto a program using GE, any program/algorithm can be used to evaluate those individuals.

The life-cycle of a Grammatical Evolution run is as follows: a chosen *Search Engine* (typically a variable-length genetic algorithm) creates a population of individuals. Each of these strings represents a potential solution for the problem to be solved, and therefore needs to be evaluated so that a fitness score can be attributed to each (that is, how well does the program that each string represents fairs on the current problem). To that end, these strings will be mapped onto programs by GE, through the use of a *Language Specification* (typically a BNF grammar). The resulting programs can then be evaluated through a *Problem Specification* (for example, an interpreter/compiler), which will attribute each program a fitness score. These scores can then be sent back to the search engine, which will use then to evolve a new population of individuals, which will also need to be evaluated.

This cycle goes on until a predefined stop condition is met (usually if a solution is found, or if a maximum number of evaluations has been reached).



This process is based on the idea of a *Genotype* to *Phenotype* mapping: an individual comprised of binary values (genotype) is evolved, and, before being evaluated, is subjected to a mapping process to create a program (phenotype), which is then evaluated by a fitness function. This creates two distinct spaces, a search space and a solution space. The following figure shows the mapping process employed in GE, and its biological equivalent.



Typically, the language specification is done through a BNF context-free grammar. A BNF grammar is represented by a tuple $\{N, T, P, S\}$, where T is a set of *Terminal* symbols, i.e., items that can appear in legal sentences of the grammar, and N is a set of *Non-Terminal* symbols, which are temporary items used in the generation of terminals. P is a set of *Productions* that map the non-terminal symbols to a sequence of terminal (or non-terminal) symbols, and S is a *Start Symbol*, from which all legal sentences must be generated.

The following is an example of a BNF grammar:

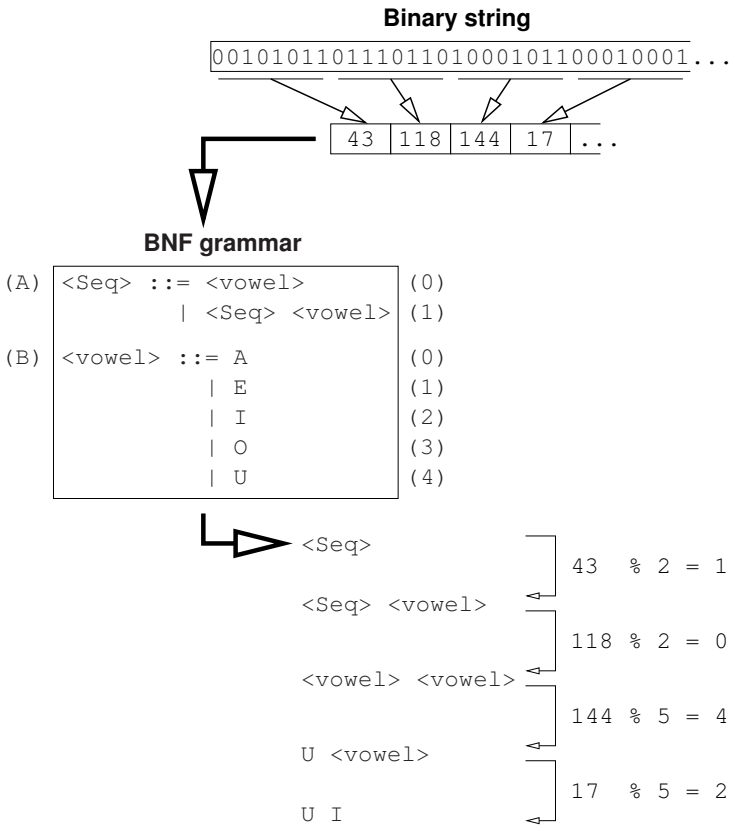
```
<Expr> ::= <Item> | <Expr> <Oper> <Item>

<Item> ::= 1.0 | x

<Oper> ::= + | - | * | /
```

This grammar can be used by GE to evolve mathematical expressions. The symbols surrounded by angled brackets (`<Expr>`, `<Oper>`, `<Item>`) are non-terminal symbols, whereas all other symbols (`1.0`, `x`, `+`, `-`, `*`, `/`) are terminal symbols. The `::=` symbol represents the separator for a production rule. In a context-free grammar, using BNF notation, a production rule is specified as a non-terminal symbol (to the left of the `::=` separator), and a series of possible transformations, to the right of the `::=` symbol, and separated by a `|` symbol (which should be read as "or").

The first non-terminal symbol to be defined is called the start symbol. In this case, the start symbol is `<Expr>`.

2.1.1 Example of Mapping Process

The following figure shows an example string mapping in GE. Starting with the variable length binary string shown, the first step is to interpret that binary string as an integer string (the *Transcription* process); as a default, GE uses 8 bits to encode each integer.



The process of *Translation* can then begin; each integer is used to choose rules from a BNF grammar. The grammar used in the example consists of two rules with two productions and five productions, respectively, with its start symbol being the `<Seq>` symbol. The first integer is therefore used to choose a production from the `<Seq>` rule; this is done by *modding* (i.e. calculating the remainder of the division) the value 43 by 2, which gives the value 1, and therefore choosing the second production, which transforms the non-terminal symbol `<Seq>` into the sequence `<Seq> <vowel>`.

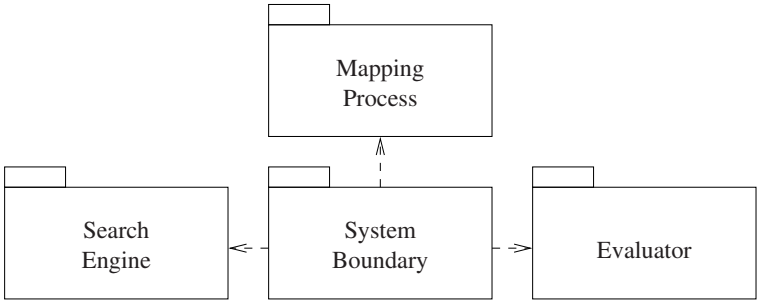
As the mapping process continues, the first non-terminal symbol of the mapping expression is used to decide which rule to use. In this case, the `<Seq>` symbol is still the first symbol on the mapping expression, so the next integer from the integer string, 118, is used to choose a production. Using the same method as before, 118 is *modded* by 2, giving the value 0, which chooses the first production; the `<Seq>` symbol is therefore replaced with `<vowel>`, and the mapping expression becomes `<vowel> <vowel>`.

This process continues until there are no non-terminal symbols left in the mapping expression. If the end of the integer string is reached and there are still non-terminal

symbols left in the mapping expression, the integer string is re-read from the start, to attempt to fully map the individual, in a process known as *Wrapping*. If, after a chosen upper limit of wrapping events, the individual is still not mapped, then the mapping process is said to fail, and the individual which generated the integer string is punished accordingly.

2.2 Using libGE

libGE is based on the idea of *Mappers*, that implement the translation process. The concept of a mapper is best understood through the following diagram:



As mentioned before, the search engine can be any algorithm that evolves binary strings. The evaluator is usually a fitness function, that evaluates how well a program solves the given problem (usually an interpreter or compiler, when evolving computer programs). The *Mapping Process* and *System Boundary* is where libGE works. Its objective is to transform (i.e. map) a genotypic structure received from the search engine onto a program recognisable by the evaluator.

2.2.1 The Mapping Process

The mapping process is where the data generated by the search engine is transformed into a program for the evaluator, using an appropriate grammar. To that purpose, a mapper is employed.

A `Mapper` is a data structure that transforms a `Genotype` structure into a `Phenotype` structure, usually by employing a grammar. By setting the `Genotype` of a mapper (using the `setGenotype()` method), a `Phenotype` is automatically generated, through the use of a `genotype2phenotype()` function, and can be retrieved from the mapper through a call to the `getPhenotype()` method.

The easiest way to use libGE in your application is to maintain a global `Mapper` object. This object should be initialised at the start of the evolutionary run, if necessary (with actions such as loading a grammar, setting a maximum number of wrapping events, etc).

When an individual from the search engine is ready for evaluation, your fitness function passes a `Genotype` structure corresponding to that individual (either created by libGE or by your own application) to the `Mapper`; when this happens, the `Mapper` will automatically update its `Phenotype` structure, which can then be retrieved by your fitness function, which in turn can pass it on to the evaluator. Once the evaluator has returned a fitness value, this value can finally be passed back to your search engine.

The following is an example (in pseudo-code) of the way a mapper is employed. Note that the program flow is controlled by the chosen search engine, and calls are made to a mapper's methods both at the start of the run (for initialisation) and each time an individual is ready for evaluation. Actions related to the use of a mapper are marked by an asterisk (*).

```
Create global mapper; *

Initialise mapper; *

Do evolutionary run

    Generate population of individuals;

    Evaluate each individual

        Transform individual to Genotype structure;

        Call Mapper.setGenotype(individual); *

        Call Mapper.getPhenotype(); *

        Pass Phenotype structure to evaluator;

        Collect fitness score from evaluator;

        Pass fitness back to search engine;

    Done

Done
```

2.2.2 The System Boundary

This is where the different data structures, used in the search engine, mapping process, and evaluator, are interfaced. To this end, libGE provides a variety of methods which ease the transformation of any search engine's data structures onto libGE `Genotype` structures. Similarly, there are several forms in which the resulting `Phenotype` structure can be retrieved, to ease the integration into the required evaluator.

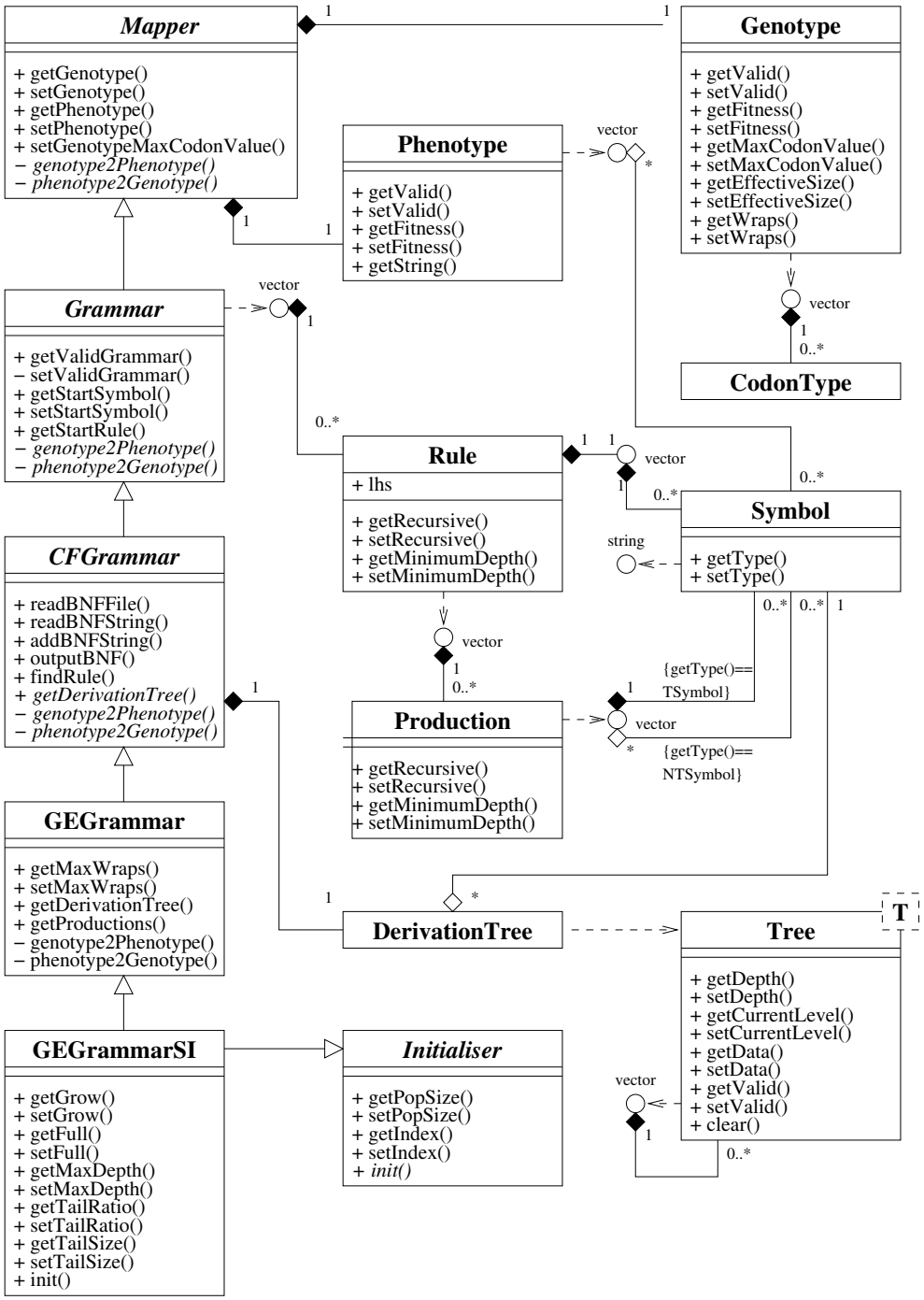
There is also some (limited) support for data structures native to specific search engines (see Chapter 5 [Search Engines], page 39), which can be configured when libGE is installed (see Section 1.2 [Installation], page 1). When using those search engines which are directly supported, the instruction `Transform individual to Genotype structure` seen in the previous pseudo-code example will not be needed, as the `Mapper.setGenotype()` method will accept the genome structures used in those search engines.

There is also some limited support for search engines written in other languages; see Chapter 5 [Search Engines], page 39.

3 Programming Interface

3.1 Class Hierarchy

The following diagram depicts the classes defined in libGE, along with their main methods.



3.2 Description

The `Mapper` abstract class defines the basic structure of a mapper. It is composed of one Genotype and one Phenotype objects, and defines the interface for the (private) genotype to phenotype mapping methods `genotype2phenotype` and `phenotype2genotype`. Every time a `Genotype` structure is assigned to it, the `Phenotype` structure is updated, and vice-versa.

The `Genotype` class defines a standard genotypic structure. It implements the interface of the standard STL `vector` class (http://www.sgi.com/tech/stl/), defining a vector of `CodonType` objects.

The `Phenotype` class defines a standard phenotypic structure. It also implements the standard STL `vector` interface, defining a vector of (pointers to) `Symbol` objects (note that a `Phenotype` object is not responsible for the deletion of the `Symbol` objects in its vector).

The `Grammar` abstract class derives from the `Mapper` class, and adds several components and methods specific to implementing a grammar as a mapper. It too implements the STL `vector` interface, defining a vector of `Rule` objects (which compose its grammar).

The `Rule` class defines the rule of a grammar, that is, a left-hand side composed of one or more symbols, and the associated productions (the possible transformations for that set of symbols). It too implements the `vector` interface, defining a vector of `Production` objects. It is also composed of one `vector` of (pointers to) `Symbol` objects, which define its left-hand side, and of which it is responsible for deletion.

The `Production` class defines a possible transformation for a given set of symbols (typically, the left-hand side of a rule). It also implements the `vector` interface, defining a vector of (pointers to) `Symbol` objects. From those, it is responsible for the deletion of `Symbol` objects with `getType()==TSymbol` only.

The `Symbol` class defines a symbol from a given grammar, and implements the STL `string` interface. It can be either a terminal symbol, with `getType()==TSymbol`, or a non-terminal symbol, with `getType()==NTSymbol`.

The `Tree` class implements the basic structure of a n-ary tree. It is a templated class, allowing for the derivation of trees of different sorts of objects.

The `CFGrammar` abstract class derives from the `Grammar` class, and defines a context-free grammar as a mapper. It adds methods specific to this kind of grammar, such as reading Backus-Naur Form text representations. It is also composed of one `Tree` object, which implements the `Tree` interface with (pointers to) `Symbol` objects, and which represents the derivation tree of the mapping process; it is not responsible for the deletion of the `Symbol` objects to which it points to.

The `GEGrammar` class is derived from the `CFGrammar` class, and implements the standard Grammatical Evolution mapping process. It implements the `genotype2phenotype()` and `phenotype2genotype()` methods, as well as the handling of wrapping events.

The `Initialiser` abstract class defines the basic form of an initialisation process. It defines the interface for the method `init()`, which called in sequence initialises all the elements of a population.

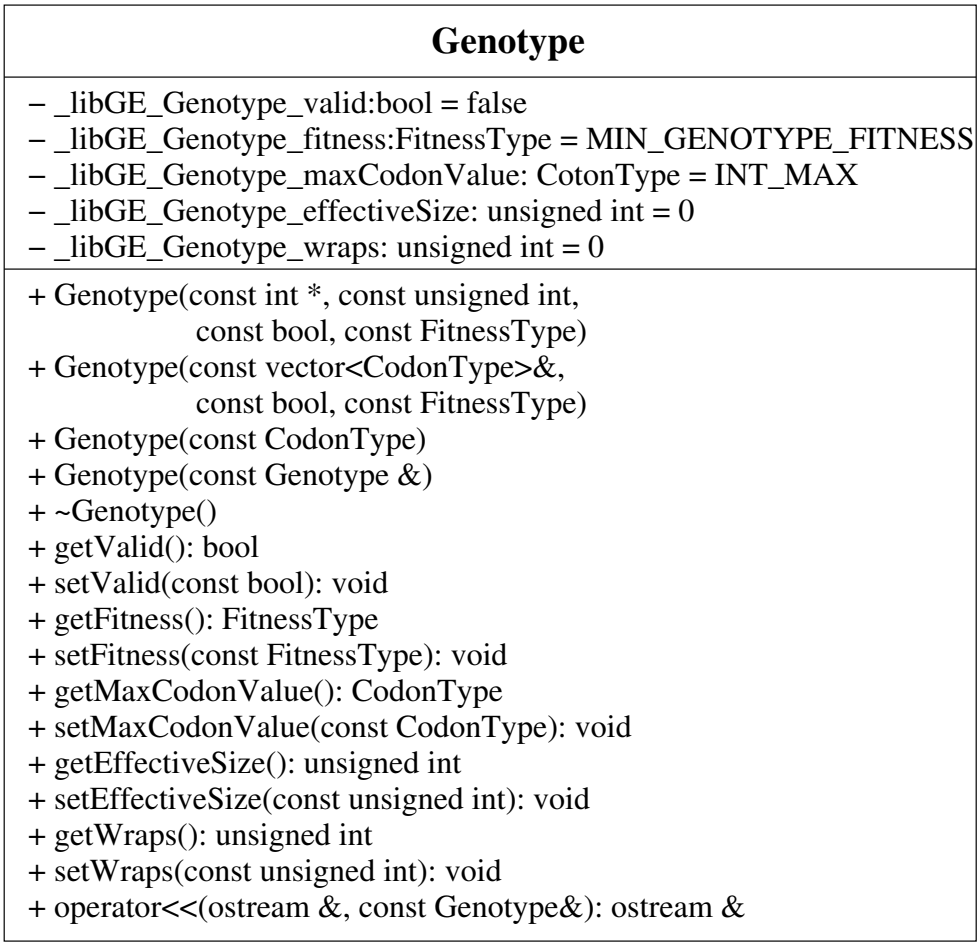
The `GEGrammarSI` class derives from the `GEGrammar` class, and adds to it the implementation of the sensible initialisation process for Grammatical Evolution (*Ryan and Azad, 2003*), by deriving from the `Initialiser` class. It initialises the `Mapper` structures `Genotype` and `Phenotype` through calls to the `init` function.

3.3 Interfaces

These are the interfaces defined for each of the classes used in libGE.

3.3.1 Genotype

This class represents the structure of a genotype, as employed by libGE mappers. It is simply an array of elements, each element corresponding to an allele of an individual. The structures used by search engines to represent genotype strings must be converted to this representation, before they can benefit from libGE's mapping process.



Member function description:

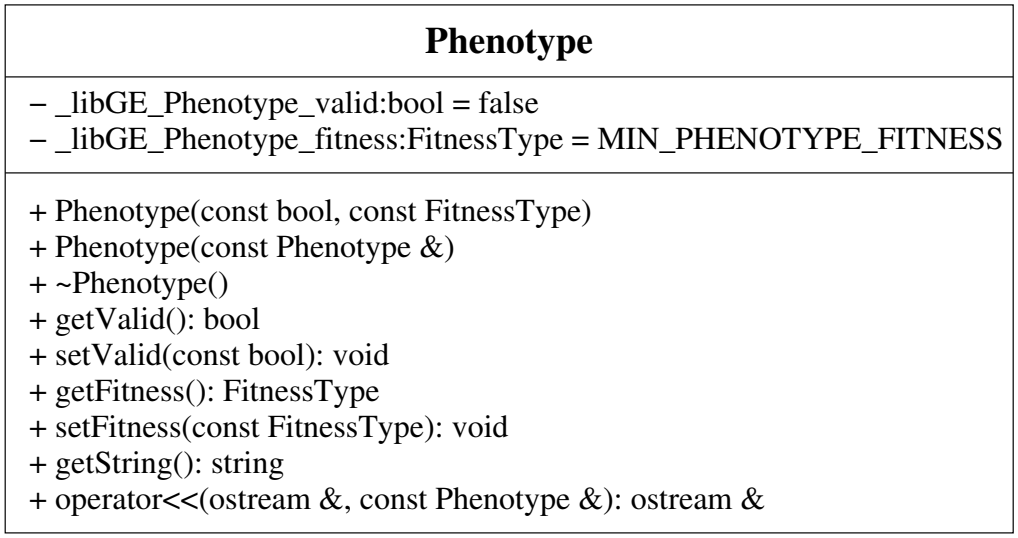
- `Genotype(const int *newArray, const unsigned int newLength, const bool newValid, const FitnessType newFitness)`: Default constructor. Creates a genotype structure of length newLength, using the elements of newArray, and sets its valid field to newValid and its fitness to newFitness. Non-specified arguments are given default values, as specified in the function prototype.
- `Genotype(const vector<CodonType> &newVector, const bool newValid, const FitnessType newFitness)`: Constructor. Creates a genotype structure using

the elements of newVector, and sets its valid field to newValid and its fitness to newFitness. Non-specified arguments are given default values, as specified in the function prototype.

- `Genotype(const CodonType maxCodonValue)`: Constructor. Creates an empty genotype structure, and sets maxCodonValue to the value specified as argument.
- `Genotype(const Genotype &copy):vector<CodonType>(copy)`: Copy constructor.
- `~Genotype()`: Destructor.
- `bool getValid() const`: Returns the current valid field.
- `void setValid(const bool newValid)`: Set a new value for the valid field.
- `FitnessType getFitness() const`: Returns the current fitness score.
- `void setFitness(const FitnessType newFitness)`: Sets a new fitness score.
- `CodonType getMaxCodonValue() const`: Returns the maximum containable value in a codon.
- `void setMaxCodonValue(const CodonType newMaxCodonValue)`: Sets the maximum containable value in a codon.
- `unsigned int getEffectiveSize() const`: Returns effective length of genotype.
- `void setEffectiveSize(const unsigned int newEffectiveSize)`: Sets effective length of genotype.
- `unsigned int getWraps() const`: Returns number of wrapping events.
- `void setWraps(const unsigned int newWraps)`: Sets number of wrapping events.
- `ostream &operator<<(ostream &stream, const Genotype &g)`: Pretty print the contents of the genotype string.

3.3.2 Phenotype

This class represents the structure of a phenotype, as employed by libGE mappers. As in nearly all cases the output of the GE mapping process will be a text generated through the use of a grammar, it made sense to implement this structure as being an array of characters. This structure will most likely have to be converted to the output required by the evaluator used to score phenotype structures.

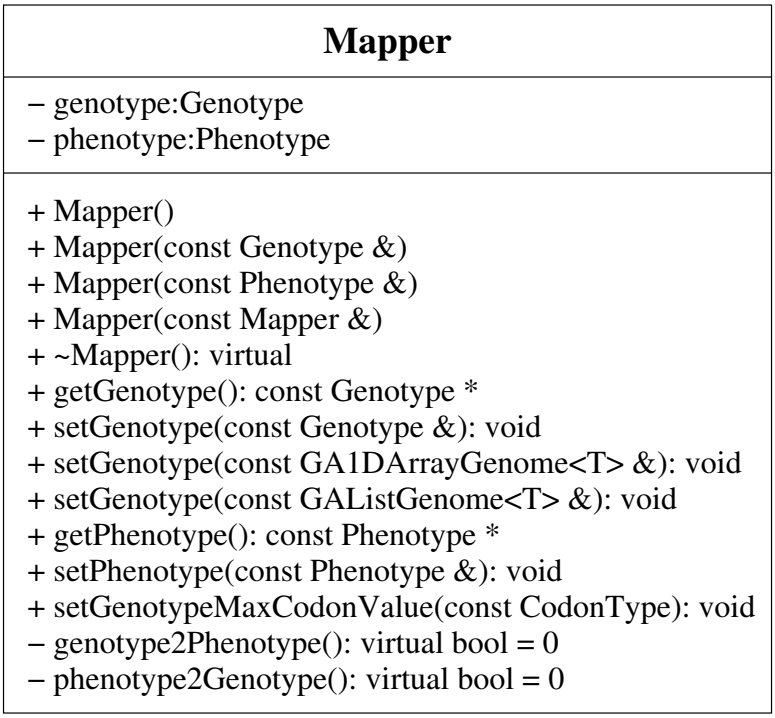


Member function description:

- `Phenotype(const bool newValid, const FitnessType newFitness)`: Default constructor. Creates a phenotype structure, and sets its valid field to newValid and its fitness to newFitness. Non-specified arguments are given default values, as specified in the function prototype.
- `Phenotype(const Phenotype &copy)`: Copy constructor.
- `~Phenotype()`: Destructor.
- `bool getValid() const`: Returns the current valid field.
- `void setValid(const bool newValid)`: Sets a new value for the valid field.
- `FitnessType getFitness() const`: Returns the current fitness score.
- `void setFitness(const FitnessType newFitness)`: Sets a new fitness score.
- `string getString() const`: Returns string containing phenotype symbols.
- `ostream &operator<<(ostream &stream, const Phenotype &ph)`: Print the contents of the phenotype.

3.3.3 Mapper

This is an abstract class that cannot be instantiated; it defines the most basic characteristics and functionality of a mapper. It is always composed of both a Genotype and Phenotype structures, and contains a (private) genotype2phenotype method, that is called every time a new genotype is set, and a (private) phenotype2genotype method, that is called every time a new phenotype is set.



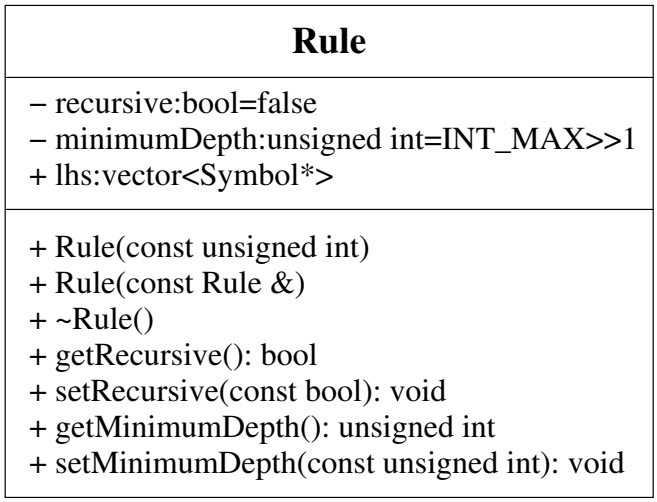
Member function description:

- `Mapper()`: Default constructor.
- `Mapper(const Genotype &newGenotype)`: Constructor with Genotype structure; set Genotype to newGenotype.
- `Mapper(const Phenotype &newPhenotype)`: Constructor with Phenotype structure; set Phenotype to newPhenotype.
- `Mapper(const Mapper& copy)`: Copy constructor.
- `~Mapper()`: Destructor.
- `Genotype const * getGenotype() const`: Returns a pointer to the genotype structure of this mapper.
- `void setGenotype(const Genotype &newGenotype)`: Sets this mapper's genotype structure to be a copy of the argument genotype structure, and calls the genotype2Phenotype private method.
- `void setGenotype(const GA1DArrayGenome<T> &genome)`: Sets this mapper's genotype structure to be a copy of the argument GA1DArrayGenome structure, and calls the genotype2Phenotype private method.

- `void setGenotype(const GAListGenome<T> &genome)`: Sets this mapper's genotype structure to be a copy of the argument GAListGenome structure, and calls the genotype2Phenotype private method.
- `Phenotype const * getPhenotype() const`: Returns a pointer to the phenotype structure of this mapper.
- `void setPhenotype(const Phenotype &newPhenotype)`: Sets this mapper's phenotype structure to be a copy of the argument phenotype structure, and calls the phenotype2Genotype private method.
- `void setGenotypeMaxCodonValue(const CodonType newMaxCodonValue)`: Sets the maximum codon value of the genotype structure.

3.3.4 Rule

This class implements a rule of a grammar. It is used by classes such as Grammar to specify grammars. It is composed of a left hand side, containing Symbol elements, and a right hand side, containing Production elements.

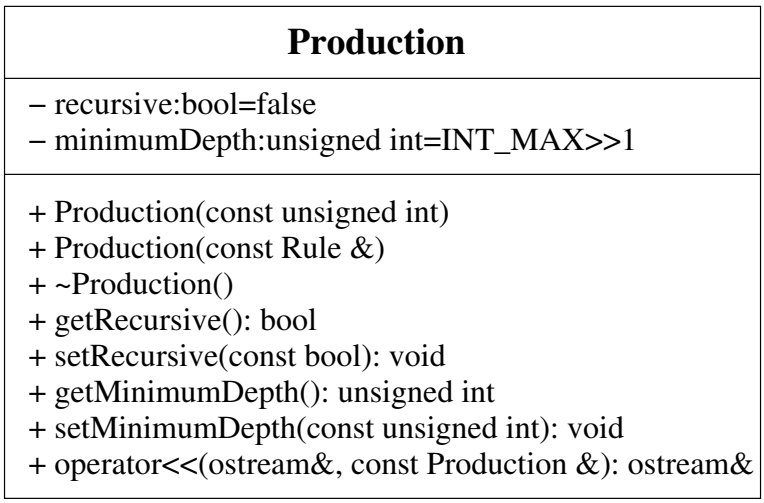


Member function description:

- `Rule(const unsigned int newLength)`: Creates a new rule with newLength elements.
- `Rule(const Rule &copy)`: Copy constructor.
- `~Rule()`: Destructor; call redefined operator clear().
- `void clear()`: Delete all productions, and all symbols stored in lhs.
- `bool getRecursive() const`: Return the recursive nature of this rule.
- `void setRecursive(const bool newRecursive)`: Update the recursive nature of this rule.
- `unsigned int getMinimumDepth() const`: Return the minimum mapping depth of this rule.
- `void setMinimumDepth(const unsigned int newMinimumDepth)`: Update the minimum mapping depth of this Rule.

3.3.5 Production

This class implements the production of a grammar. It is used by the Rule class to specify the various productions which make up a rule.



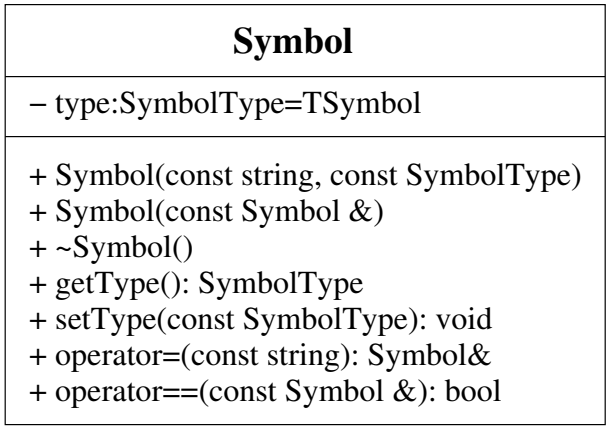
Member function description:

- `Production(const unsigned int newLength)`: Creates a new Production with newLength elements.
- `Production(const Production &copy)`: Copy constructor; copy all symbols.
- `~Production()`: Destructor; call redefined operator clear().
- `void clear()`: Delete all terminal symbols and all symbol references.
- `bool getRecursive() const`: Return the recursive nature of this production.
- `void setRecursive(const bool newRecursive)`: Update the recursive nature of this production.
- `unsigned int getMinimumDepth() const`: Return the minimum mapping depth of this production.
- `void setMinimumDepth(const unsigned int newMinimumDepth)`: Update the minimum mapping depth of this production.
- `ostream &operator<<(ostream &stream, Production const &p)`: Pretty print the contents of the production.

3.3.6 Symbol

This class implements a symbol of a grammar, be it a non-terminal or terminal symbol. It is used both by the Rule class, to specify its left side non-terminal symbols, and by the Production class, to specify its terminal and non-terminal symbols.

It is also used by the Phenotype class, to specify phenotypic symbols. It can hold any data type, but all data types will be treated as text, so for example no arithmetic operations are possible.

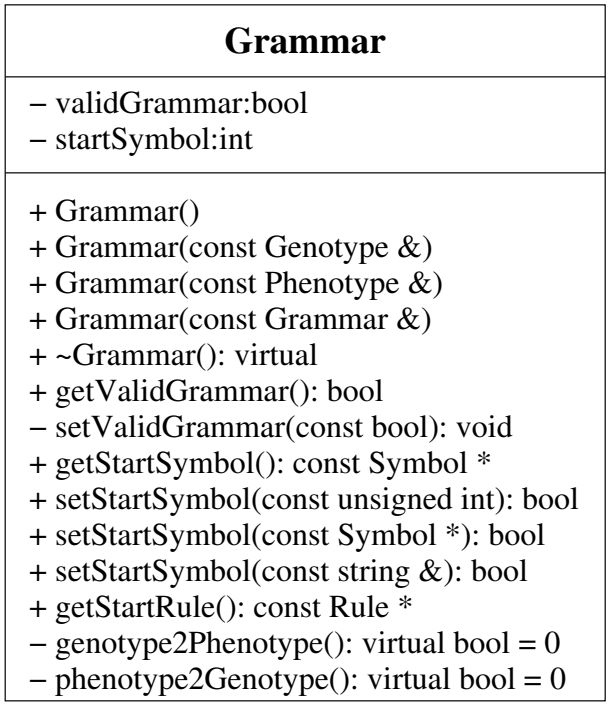


Member function description:

- `Symbol(const string newArray, SymbolType newType)`: Create a new Symbol object as a copy of newArray and with type newType, if specified; otherwise the default values specified in the function prototype are used.
- `Symbol(const Symbol &copy)`: Copy constructor.
- `~Symbol()`: Destructor.
- `SymbolType getType() const`: Return the type of the symbol (NTSymbol or TSymbol).
- `void setType(const SymbolType newType)`: Set the type of the symbol.
- `Symbol &operator=(const string newArray)`: Copy newArray.
- `bool operator==(const Symbol &newSymbol)`: Symbol comparison operator.

3.3.7 Grammar

This is an abstract class that cannot be instantiated; it defines the basic characteristics and functionality of a Grammar mapper. It is composed of a vector of Rule objects, defining a grammar, and of fields defining the maximum number of allowed wrapping events and the validity of the current grammar.



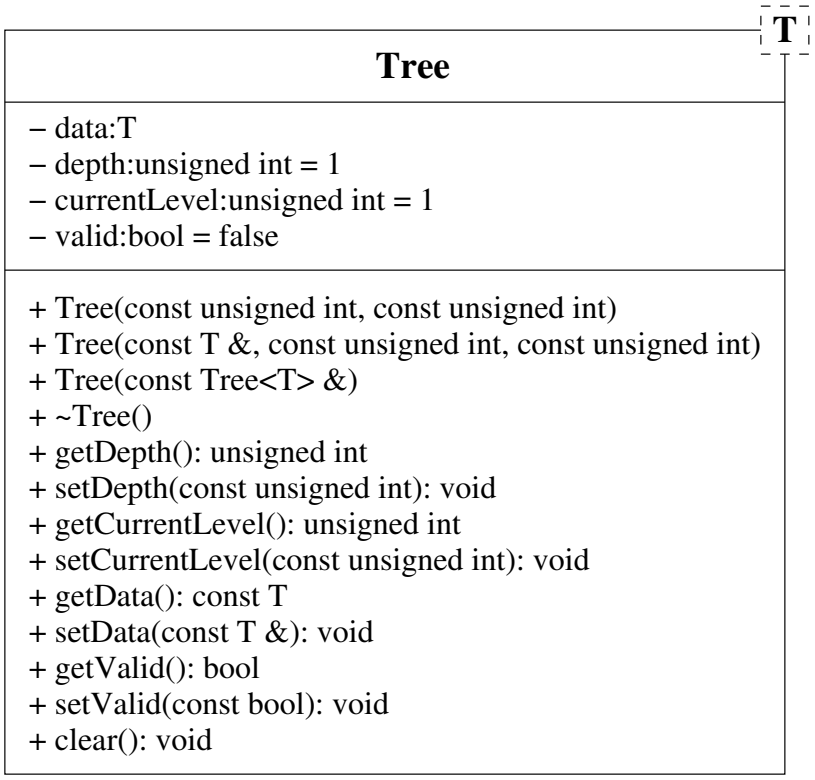
Member function description:

- `Grammar():Mapper()`: Default constructor.
- `Grammar(const Genotype &newGenotype)`: Constructor setting the genotype structure of this mapper to newGenotype.
- `Grammar(const Phenotype &newPhenotype)`: Constructor setting the phenotype structure of this mapper to newPhenotype.
- `Grammar(const Grammar& copy)`: Copy constructor.
- `~Grammar()`: Destructor: delete all Productions first, and then all rules.
- `bool getValidGrammar() const`: Return the validity of the current grammar.
- `void setValidGrammar(const bool newValidGrammar)`: Set the validity of the grammar.
- `const Symbol* getStartSymbol() const`: Return pointer to current start symbol.
- `bool setStartSymbol(const unsigned int)`: Change start symbol by index on Vector of rules.
- `bool setStartSymbol(const Symbol*)`: Change start symbol by symbol pointer.
- `bool setStartSymbol(const string&)`: Change start symbol by string.

- `const Rule* getStartRule() const`: Return pointer to current start rule.

3.3.8 Tree

This is a templated class that implements an n-ary tree.

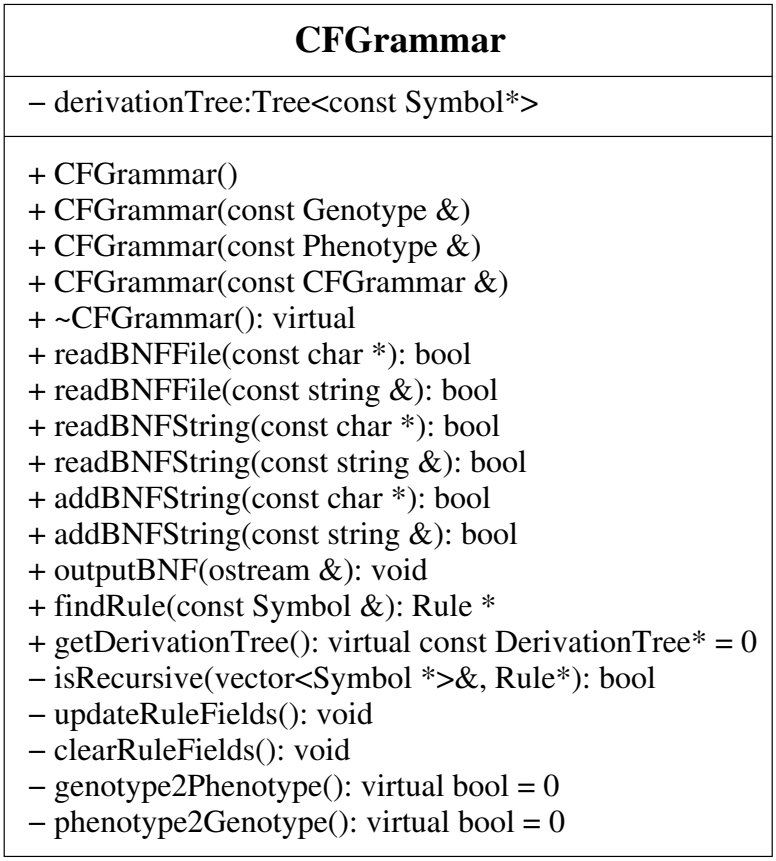


Member function description:

- `Tree(const unsigned int newDepth,const unsigned int newCurrentLevel)`: Constructor; initialises depth and current level.
- `Tree(const T &newData,const unsigned int newCurrentLevel,const unsigned int newDepth)`: Constructor; initialises data to match arguments.
- `Tree(const Tree<T> &copy)`: Copy constructor.
- `~Tree()`: Destructor; clear all sub-trees.
- `unsigned int getDepth() const`: Return depth from this node until the deepest leaf.
- `void setDepth(const unsigned int newDepth)`: Set new depth of this node.
- `unsigned int getCurrentLevel() const`: Return level of this node.
- `void setCurrentLevel(const unsigned int newCurrentLevel)`: Set new current level of this node.
- `const T getData() const`: Return data stored in this node.
- `void setData(const T &newData)`: Set data to store in this node.
- `bool getValid() const`: Return validity of this node.
- `void setValid(const bool newValid)`: Set validity of this node.
- `void clear()`: Remove all elements of this tree.

3.3.9 CFGrammar

This class implements the methods associated with the manipulation of context-free grammars. It implements input and output methods for this type of grammar, along with manipulation and analysis methods.



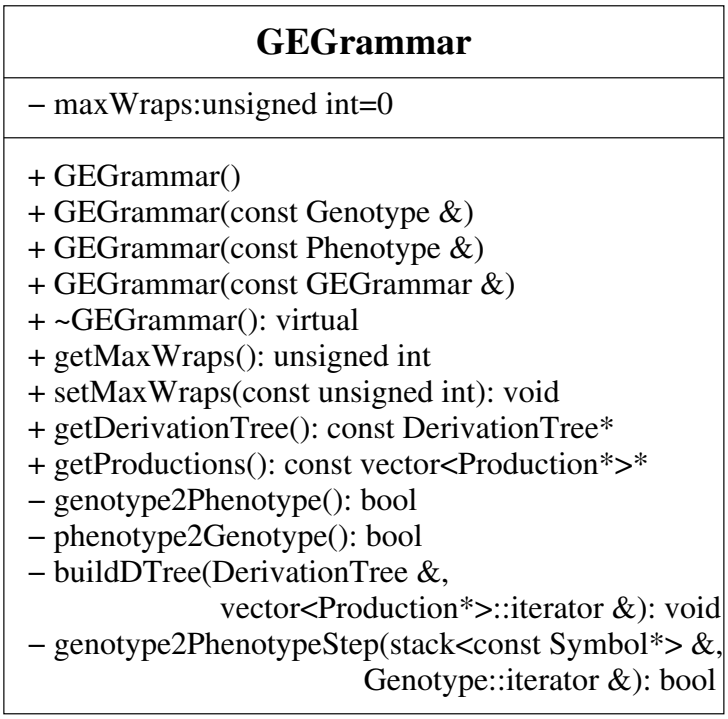
Member function description:

- `CFGrammar():Grammar()`: Default constructor.
- `CFGrammar(const Genotype &newGenotype)`: Constructor setting the genotype structure of this mapper to newGenotype.
- `CFGrammar(const Phenotype &newPhenotype)`: Constructor setting the phenotype structure of this mapper to newPhenotype.
- `CFGrammar(const CFGrammar& copy)`: Copy Constructor.
- `~CFGrammar()`: Destructor.
- `bool readBNFFile(const char *filename)`: Opens the file whose name is passed as an argument, reads its contents onto a character string, and calls readBNFString.
- `bool readBNFFile(const string &filename)`: Interface to the `readBNFFile(const char *)` method.

- `bool readBNFString(const char *stream)`: Reads in the BNF grammar specified by its argument text. Returns true if loading of grammar was successful, false otherwise.
- `bool readBNFString(const string &stream)`: Interface to the `readBNFString(const char *)` method.
- `bool addBNFString(const char *stream)`: Reads in the BNF grammar-part specified by its argument text, and adds it to the current grammar. Returns true if loading of new grammar part was successful, false otherwise.
- `bool addBNFString(const string &stream)`: Interface to the `addBNFString(const char *)` method.
- `void outputBNF(ostream& stream)`: Pretty print the current BNF grammar.
- `Rule* findRule(const Symbol &nonterminal)`: Returns the address of the rule defining the argument non-terminal symbol, if it exists; otherwise returns NULL.
- `const DerivationTree* getDerivationTree()`: Builds the the current derivation tree, and returns its address.
- `bool isRecursive(vector<Symbol*> &visitedRules, Rule *currentRule)`: Returns the calculated recursive nature of the Rule passed as argument, and updates its minimum mapping depth (minimumDepth)
- `void updateRuleFields()`: Update recursive and minimumDepth fields for every Rule and Production in grammar.
- `void clearRuleFields()`: Update recursive and minimumDepth fields for every Rule and Production in grammar.

3.3.10 GEGrammar

This class implements the standard GE mapping process. It implements the virtual methods genotype2phenotype and phenotype2genotype, and the standard GE wrapping operator.



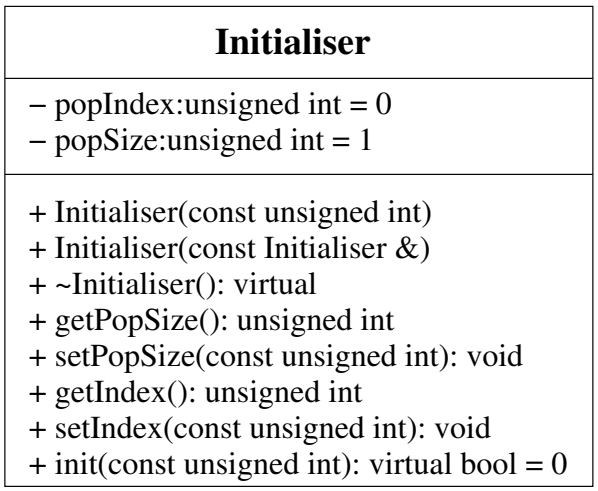
Member function description:

- `GEGrammar():CFGrammar()`: Default constructor.
- `GEGrammar(const Genotype &newGenotype)`: Constructor setting the genotype structure of this mapper to newGenotype.
- `GEGrammar(const Phenotype &newPhenotype)`: Constructor setting the phenotype structure of this mapper to newPhenotype.
- `GEGrammar(const GEGrammar& copy)`: Copy Constructor.
- `~GEGrammar()`: Destructor.
- `unsigned int getMaxWraps() const`: Return number of maximum allowed wrapping events.
- `void setMaxWraps(const unsigned int newMaxWraps)`: Set the new number of maximum allowed wrapping events.
- `const DerivationTree* getDerivationTree()`: Builds the the current derivation tree, and returns its address; if derivation tree is impossible to build, returns NULL.
- `const vector<Production*>* getProductions()`: Returns a vector of all productions used during the mapping process.
- `void buildDTree(DerivationTree&, vector<Production*>::iterator&)`: Builds the derivation tree, based on the productions vector. Arguments are current tree node, and iterator on productions vector.

- `bool genotype2Phenotype()`: Updates the contents of the phenotype structure, based on the current genotype and the current grammar, and according to the standard GE mapping process. Returns true upon a successful mapping, and false otherwise, and also updates the valid field of the phenotype.
- `bool phenotype2Genotype()`: Updates the contents of the genotype structure, based on the current phenotype and the current grammar, and according to a mapping process corresponding to the inverse of the standard GE mapping process. Returns true upon a successful inverse mapping, and false otherwise.
- `bool genotype2PhenotypeStep(stack<const Symbol*> &nonterminals, Genotype::iterator &genoIt)`: Performs one step of the mapping process, that is, maps the next non-terminal symbol on the nonterminals stack passed as argument, using the codon at the position pointed by genoIt.

3.3.11 Initialiser

This is an abstract class that cannot be instantiated; it defines the most basic characteristics and functionality of an initialiser. An initialiser contains a population size and a population index. It defines the virtual method init(), which applies the initialisation process.

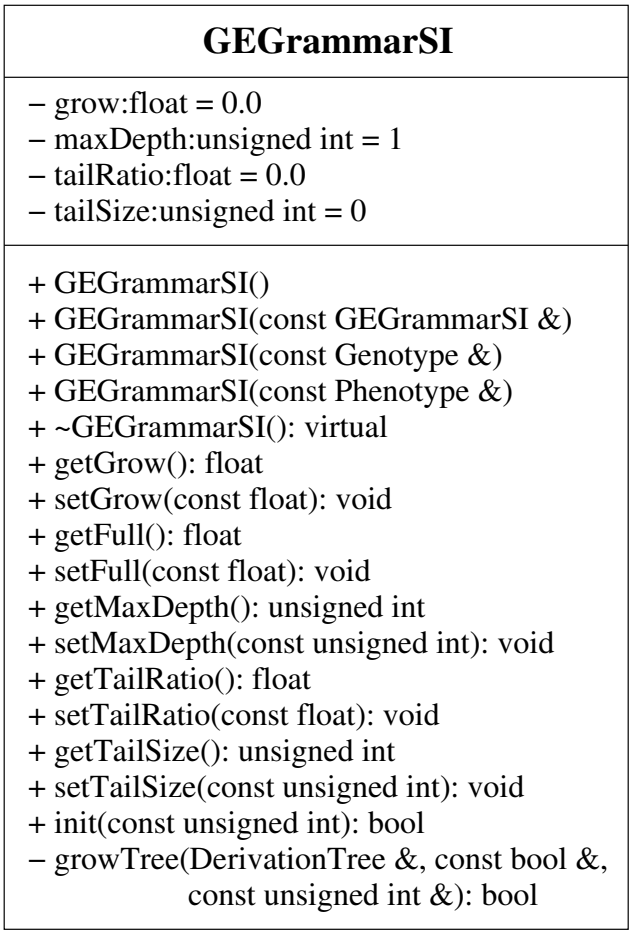


Member function description:

- `Initialiser(const unsigned int newPopSize)`: Default constructor. Create vector of random seeds.
- `Initialiser(const Initialiser&)`: Copy constructor.
- `~Initialiser()`: Destructor.
- `unsigned int getPopSize() const`: Returns current population size of initialiser.
- `void setPopSize(const unsigned int newPopSize)`: Sets population size of initialiser.
- `unsigned int getIndex() const`: Returns current index of initialiser.
- `void setIndex(const unsigned int newSeedsPos)`: Sets population index of initialiser.

3.3.12 GEGrammarSI

This class implements the standard GE ramped half-and-half initialisation routine, also known as sensible initialisation. It implements the virtual methods from the class Initialiser.



Member function description:

- `GEGrammarSI::GEGrammarSI()`: Default constructor.
- `GEGrammarSI::GEGrammarSI(const Genotype& newGenotype)`: Constructor setting the genotype structure of this mapper to newGenotype.
- `GEGrammarSI::GEGrammarSI(const Phenotype& newPhenotype)`: Constructor setting the phenotype structure of this mapper to newPhenotype.
- `GEGrammarSI::GEGrammarSI(const GEGrammarSI& copy)`: Copy Constructor.
- `GEGrammarSI::~GEGrammarSI()`: Destructor.
- `float GEGrammarSI::getGrow() const`: Return the grow percentage set for this initialiser.
- `void GEGrammarSI::setGrow(const float newGrow)`: Update the grow percentage for this initialiser.

- `float GEGrammarSI::getFull() const`: Return the full percentage set for this initialiser.
- `void GEGrammarSI::setFull(const float newFull)`: Update the full percentage for this initialiser.
- `unsigned int GEGrammarSI::getMaxDepth() const`: Return the maxDepth parameter set for this initialiser.
- `void GEGrammarSI::setMaxDepth(const unsigned int newMaxDepth)`: Update the maxDepth parameter for this initialiser.
- `float GEGrammarSI::getTailRatio() const`: Return the tail size percentage set for this initialiser.
- `void GEGrammarSI::setTailRatio(const float newTailRatio)`: Update the tail size percentage for this initialiser, and set tailSize to 0.
- `unsigned int GEGrammarSI::getTailSize() const`: Return the tail size set for this initialiser.
- `void GEGrammarSI::setTailSize(const unsigned int newTailSize)`: Update the tail size for this initialiser, and set tailRatio to 0.
- `bool GEGrammarSI::growTree(DerivationTree &tree,const bool &growMethod,const int &maximumDepth)`: Grow the derivation tree according to the grow or full method, up to the maximumDepth specified.
- `bool GEGrammarSI::init(const unsigned int index)`: Initialise the Genotype and Phenotype structures, according to the sensible initialisation technique for GE. If index is not set (or if it is set to UINT_MAX), initialise the structures as part of a series of calls to init(); if it is set, initialise the structures as being the index-th member (out of popSize) of the population. The next call to this routine will initialise the structures as being the next individual of a population of popSize individuals. There is an exception to this rule: * If a specific index is set, then the structures are initialised as being the index-th individual of a population of popSize, and the next call to this routine will initialise the index-th+1 individual (unless a specific index is set again).

3.4 Designing Your Own Mappers

This section describes how to extend libGE's capabilities by defining your own mappers. The best way to customise the behaviour of a mapper is to derive a new class. Depending on the type of mapper to be developed, one of the existing classes should be derived; for example, the `Mapper` class is the most general (for any kind of mapping process), whereas the `Grammar` class is the most adequate for mapping processes involving any kind of grammars, and finally the `GEGrammar` class is the adequate choice for mapping processes very similar to the standard Grammatical Evolution approach.

For example, to create a mapper that reads a BNF grammar but implements a slightly different mapping process to that of standard GE, you should derive your class from the `CFGrammar` class, as this class already implements a method to read BNF grammars. Your class hierarchy would look something like this:

```
class MyMapper : public CFGrammar
```

Supposing the structure to store a grammar suits your mapping process, the only methods you will have to implement are the virtual methods from `CFGrammar`, that is, the methods `getDerivationTree`, `genotype2Phenotype` and `phenotype2Genotype`. These would look like this:

```
const DerivationTree* MyMapper::getDerivationTree(){ ... your code ...}
bool MyMapper::genotype2Phenotype(){ ... your code ...}
bool MyMapper::phenotype2Genotype(){ ... your code ...}
```

4 Grammars

Grammatical Evolution is based mainly on context-free grammars, specified using the Backus-Naur Form (BNF) format, and therefore the class structures used in libGE are based around this principle. However, the library was designed keeping in mind possible extensions to GE, and is therefore extensible to other grammar types and formats.

4.1 Format of Grammars

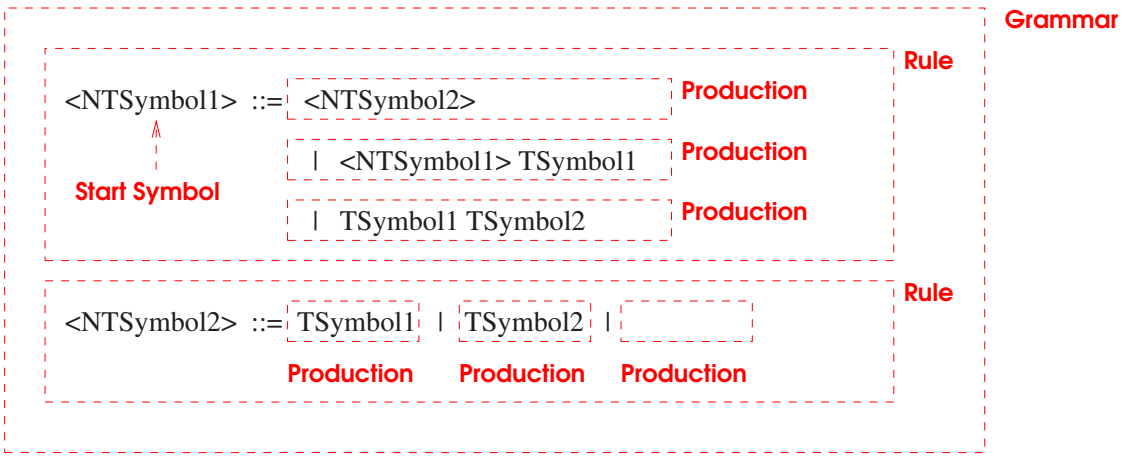
There are several formats in which a grammar can be specified. Traditionally, the format of choice when specifying a grammar is BNF, and this is the format of choice for libGE; although not the easiest to parse, it is easy to write and understand by the user. All grammars should be written in ASCII format.

4.1.1 BNF Grammars

A BNF grammar is represented by a tuple $\{N, T, P, S\}$, where T is a set of *Terminal* symbols, i.e., items that can appear in legal sentences of the grammar, and N is a set of *Non-Terminal* symbols, which are temporary items used in the generation of terminals. P is a set of *Productions* that map the non-terminal symbols to a sequence of terminal (or non-terminal) symbols, and S is a *Start Symbol*, from which all legal sentences must be generated.

When describing the syntax of BNF grammars accepted by libGE, the following definitions are used:

- a `Production` is a specific transformation associated with a non-terminal symbol, and is a set of terminal and non-terminal symbols;
- a `Rule` is the set of all productions associated with a non-terminal symbol;
- a `Grammar` is the set of all rules specified.



4.1.1.1 Rule

A rule consists of a non-terminal symbol (`<NTSymbol>`), followed by the `::=` token, and by a sequence of one or more productions, separated by the `|` token. Newlines can be used to separate productions, provided that they occur only before the `|` tokens; newlines preceded

by a \ character are ignored. Spaces are allowed (but not mandatory) before and after each of the symbols `<NTSymbol>`, `::=` and `|`, and before and after each production, acting as token separators. Rules must always be specified in separate lines.

4.1.1.2 Production

A production consists of a sequence of zero or more non-terminal ('`<NTSymbol>`') or terminal ('`TSymbol`') symbols. A sequence of one or more spaces and/or tabular characters between symbols will result in a single space character being placed in between those two symbols in sentences generated with the grammar, while no space between symbols results in those symbols following each other with no separator character, in generated sentences (i.e. those symbols will be concatenated in the resulting expression). Empty productions are accepted, as shown in the previous example.

4.1.1.3 Non-Terminal Symbols

A non-terminal symbol is a sequence of one or more characters, enclosed within angle brackets (`<...>`). Most ASCII characters are allowed, with the exception of escape sequences.

4.1.1.4 Terminal Symbols

A terminal symbol is a sequence of one or more ASCII characters, with the exception of the space and newline characters, and the characters '<', '>', '\', '"' and '|' (unless they are preceded by a '\' character).

A terminal symbol can also be a sequence of any ASCII characters enclosed within quotes (`"..."`), with the exception of newline, and the character '"' (unless they are preceded by a '\' character).

Additionally, all standard ASCII escape sequences are accepted (with the exception of the `\xnnn` sequence), and are translated into their corresponding characters on sentences generated using the grammar. So if your evaluator needs the output `\n`, then use `\\n` in your grammar.

4.1.1.5 Start Symbol

The start symbol, by default, is the first symbol defined in a grammar (in the example provided, the `<NTSymbol1>` symbol). However, libGE provides methods to specify which non-terminal symbol to use as a start symbol, when the default behaviour is not what is intended (see Chapter 3 [Programming Interface], page 9).

4.1.2 Extended BNF Grammars

Extended BNF grammars are not supported at the moment by libGE.

4.1.3 XML Grammars

Grammars in XML format are not supported at the moment by libGE.

4.2 Type of Grammars

4.2.1 Context-Free Grammars

The original Grammatical Evolution system was designed to work with *Context-Free Grammars* (CFG), that is, grammars in which the choice of a production to map a specific non-terminal symbol is always the same throughout the mapping process, regardless of the context in which that symbol appears, and does not depend on external factors. In other words, a specific codon value from the genotype string will always choose the same production for a specific non-terminal symbol.

4.2.1.1 Errors In Context-Free Grammars

- Endless recursions. Endless recursions are caused by non-terminal symbols that recursively map onto themselves. Consider the following example:

  ```
  <S> ::= <A> | <B>
  ```

  ```
  <A> ::= a
  ```

  ```
  <B> ::= <B> b
  ```

  In this grammar, if the non-terminal symbol `<B>` is chosen, then a sequence of terminal symbols only will never be generated. In cases when that might happen, the mapping methods in libGE will signal that the resulting `Phenotype` structure is false.

- Useless symbols. Useless symbols are non-terminal symbols which are present in the grammar, but are unreachable from the start symbol. Consider the following example:

  ```
  <S> ::= a | b
  ```

  ```
  <A> ::= a
  ```

  In this grammar, the non-terminal symbol `<A>` is never reached, as there is no way to reach it from a mapping string starting with the symbol `<S>`. These symbols are disregarded by libGE, and cause no errors to the mapping process.

- Undefined symbols. Undefined symbols are non-terminal symbols which are present in one or more productions, but have no defining rule. Consider the following example:

  ```
  <S> ::= <A> | <B>
  ```

  ```
  <A> ::= a
  ```

  In this grammar, the non-terminal symbol `<B>` is undefined, because there is no rule defining what that symbol can be mapped to. In libGE, grammars containing undefined symbols are accepted, but whenever one of those symbols is encountered during the mapping process, it is included directly in the resulting `Phenotype` structure, and that structure is flagged as being false.

4.2.1.2 Regular Grammars

Regular Grammars are a subset of CFG, in which all productions are composed of zero or one terminal symbols, followed by zero or one non-terminal symbols. The following is an example of a regular grammar:

```
<A> ::= a <B>
<B> ::= b
```

Regular grammars describe exactly all regular languages and are in that sense equivalent with finite state automata and regular expressions. They can be right regular grammars, as in the example above, or left regular grammars, as in the example below:

```
<A> ::= <B> a
<B> ::= b
```

As they are a subset of CFG, regular grammars are supported by libGE.

4.2.1.3 Closed Grammars

There is a specific type of CFG, often called a *Closed Grammar*. These are grammars which contain only one non-terminal symbol, and therefore are composed of one rule only. The following is an example of a closed grammar:

```
<Expr> ::= <Expr> + x
         | <Expr> - x
         |  x
```

4.2.2 Context-Sensitive Grammars

Context-sensitive grammars are not supported at the moment by libGE.

4.2.3 Attribute Grammars

Attribute grammars are not supported at the moment by libGE.

4.3 libGE Extensions

To improve user control over the mapping process, libGE introduces a set of commands that can be incorporated into grammars. These take the form of special non-terminal symbols, which encode specific instructions to libGE's mapping process. Whenever one of these symbols is encountered during the mapping process, the associated command/instruction is executed.

The case may arise when a user has specified non-terminal symbols in a grammar which bear the same name as the extension symbols introduced by libGE. To deal with such cases, the following policy is employed:

- If one of the extension symbols is present in the grammar, but does not have a defining rule, then the associated extension command/instruction is executed;
- If one of the extension symbols is present in the grammar, and it has a defining rule, then the associated extension command/instruction is ignored, and the symbol is replaced with one of its productions, as per the standard mapping process.

4.3.1 <GECodonValue>

The `<GECodonValue>` symbol has been introduced to allow the user to insert codon values directly from the genotype onto the resulting phenotype. Consider the following grammar:

```
<Expr> ::= <Expr> + <GECodonValue>
         | <Expr> - <GECodonValue>
         |  <GECodonValue>
```

This grammar will generate mathematical expressions that will add or subtract integer values, which are extracted from the genotype structure. Note that every time a `<GECodonValue>` symbol is replaced with a codon value, that codon is considered "used", so the next step in the mapping process will use the following codon.

The range of values generated by `<GECodonValue>` depends on the data type of the codon used. The values extracted can be restricted to specific ranges, by using the following notation:

```
<GECodonValue[-low][+high]>
```

where `-low` and `+high` are optional value boundaries. So for example, `<GECodonValue-10>` will extract the current codon and restrict it to be a value of 10 or higher, `<GECodonValue+20>` will restrict the value to be of 20 or lower, and `<GECodonValue-10+20>` will restrict it to be between 10 and 20 (inclusive).

4.4 Examples of Grammars

4.4.1 Text examples

The following grammar defines a sequence of one or more vowels:

```
<Seq>   ::= <Vowel> | <Seq> <Vowel>
<Vowel> ::= A | E | I | O | U
```

The following grammar defines a useless but somewhat funny sentence:

```
<Sentence>   ::= <Article> <Nounphrase> <Verb> <Article> <Nounphrase>
<Article>    ::= the | a
<Nounphrase> ::= <Noun> | <Adjectives> <Noun>
<Noun>       ::= cat | dog | mouse
<Adjectives> ::= <Adjective> | <Adjectives> <Adjective>
<Adjective>  ::= smiling | big | happy | fake
<Verb>       ::= ate | hit
```

4.4.2 Math examples

The following grammar defines a simple arithmetic expression, working with integer constants, which are constructed by concatenation:

```
<Expr> ::= <Int>
         | <Expr><Oper><Int>
<Int>  ::= <Int><Digit>
         | <Digit>
<Digit>::= 0|1|2|3|4|5|6|7|8|9
<Oper> ::= +
         | -
         | *
         | /
```

The following closed grammar defines a mathematical expression, working with several operators, in which integer constants are created by extracting codons from the genotype string:

```
<Expr> ::= (<Expr>)
         | sin(<Expr>)
         | cos(<Expr>)
         | <Expr>+<GECodonValue>
         | <Expr>-<GECodonValue>
         | <Expr>*<GECodonValue>
         | <Expr>/<GECodonValue>
         | <GECodonValue>
         | (-<GECodonValue>)
```

4.4.3 Code examples

The following grammar defines a C function that returns a mathematical expression:

```
<Fn> ::= double function(double x)\n{\
         \t return <Expr>;\n\
         }\n
<Expr> ::= <Item>
         | <Expr><Oper><Item>
<Item> ::= 1.0
         | x
<Oper> ::= +
         | -
         | *
         | /
```

The following grammar defines a ready-to-compile C++ program, which returns the number of tries a user needed to guess a constant, evolved with number concatenation:

```
<Prog>::="#include<iostream>\n\nusing namespace std;\n\n"<MainFunction>
<MainFunction>::="int main(){\n"<MainVars>
<MainVars> ::= "int guess,tries=0,value="<Value>";\n"<MainLoop>
<Value> ::= <Value><Digit> | <Digit>
<Digit> ::= 0|1|2|3|4|5|6|7|8|9
<MainLoop> ::= "do{\n\tcin >> guess;\n\
        if(guess>value)\n\
                cout << \"Lower.\\n\";\n\
        else if(guess<value)\n\
                cout << \"Higher.\\n\";\n\
        tries++;\n\
}while(guess!=value);\n"<EndCode>
<EndCode> ::= "cout << \"Guessed in \" << tries << \" tries.\\n\";\n\
return tries;\n}\n"
```

5 Search Engines

In this chapter you will find a brief introduction to the search engines which have so far been successfully tested with libGE. Most of these are C++ packages, but some are written in other languages. You will also find a quick guide to using your own search engine.

Note: check the FAQ section (see Appendix A [FAQ], page 127) for possible incompatibilities between these search engines and some evaluators.

5.1 IlliGAL sga-c

- http://www-illigal.ge.uiuc.edu/index.php3

The IlliGAL sga-c code is a C language port of the original simple GA code from David Goldberg's book (*Goldberg,1989*). It provides a few extensions to the original code, and is well documented in a technical report (*Smith,1994*), which is included with the download package.

5.1.1 License

The sga-c code is licensed free of charge, and therefore with no warranty of any kind.

5.1.2 Version

The version tested with libGE is v1.1.

5.1.3 Installation

Simply decompress the package; it will create an '`sga-c`' directory. It should work on most C compilers simply by running `make`; on specific architecture, you might need to change settings at the start of the '`sga.h`' file. There is no system-wide installation available.

5.1.4 Main features

The code provides a fast and effective implementation of a fixed-length genetic algorithm, albeit severely limited. It is intended to be a simple program for first-time GA experimentation. Its main features are:

- Fast and memory efficient: uses bit-level representation of chromosomes, and bitmask implementation of the crossover and mutation operators.
- Provides roulette-wheel, stochastic remainder and tournament selection routines.

5.1.5 Usage

The code accepts parameters either through an input file, or by `stdin` input. If you want to use different selection routines (the default choice is roulette-wheel), you will need to change the '`Makefile`'.

The fitness function is contained in a file called '`app.c`'. Most of the functions in this file are explained in the example '`app.c`' file. The function containing the problem specification is called '`objfunc()`'. It is responsible for setting the fitness score of the `struct individual` chromosome passed as argument.

5.1.6 Examples

Two examples are provided with the code, in files '`app1.c`' (which is a copy of '`app.c`') and '`app2.c`'. These are explained in the technical report provided.

5.1.7 Documentation

As the package is quite minimalistic, it does not need an exhaustive documentation. An explanation of each file and its functionality exists in the technical report provided. No website or mailing-list exist for the project.

5.1.8 Using with libGE

As this package is written in C, a specific *wrapper* is needed to allow it to work with libGE. The code for this wrapper is included in the files '`GE_ILLIGALSGA.h`' and '`GE_ILLIGALSGA.cpp`'. It maintains a static `GEGrammar` mapper in memory, and provides the following functionality:

- `int GE_ILLIGALSGA_getGenotype(int **)`: allocates memory for an integer array, stores the current genotype in that array, and returns the address of the allocated memory in the argument pointer, as well as its size as the return value. The user is responsible for freeing the allocated memory.
- `int GE_ILLIGALSGA_getPhenotype(char **)`: allocates memory for a NULL-terminated character array, stores the current phenotype in that array, and returns the address of the allocated memory in the argument pointer, as well as its size as the return value. The user is responsible for freeing the allocated memory.
- `int GE_ILLIGALSGA_validPhenotype()`: returns the validity of the phenotype contained in the static mapper.
- `void GE_ILLIGALSGA_setMaxWraps(int)`: sets the number of allowed wrapping events for the static mapper.
- `int GE_ILLIGALSGA_readBNFFile(char *)`: interface to the `readBNFFile(char *)` method in `CFGrammar`, for the static mapper.
- `int GE_ILLIGALSGA_readBNFString(char *)`: interface to the `readBNFString(char *)` method in `CFGrammar`, for the static mapper.
- `int GE_ILLIGALSGA_outputBNF()`: interface to the `outputBNF(ostream&)` method in `CFGrammar`, for the static mapper, with `cout` as the output stream.
- `int GE_ILLIGALSGA_translate(unsigned *, int, int *, int, int, int)`: accepts an array of `unsigned`, representing an sga-c chromosome, and translates it to an array of integers, using the specified number of bits per gene, and limiting the integer values to the range specified.
- `void GE_ILLIGALSGA_Mapper(struct individual *, int, int)`: accepts an sga-c chromosome, transforms it into a libGE `Genotype` structure, and assigns it to the static mapper.

Each example in the '`EXAMPLES`' directory contains files which are to be copied into the '`sga-c`' directory, overwriting some of the existing default files.

5.2 GALib

- http://lancet.mit.edu/ga/

The GALib is a C++ library of genetic algorithm components. It has been around for a long time, and is a well established library. It includes tools for implementing genetic algorithms using practically any representation and genetic operators, including parallel computing extensions, and even graphical extensions.

5.2.1 License

The GALib is distributed under a BDS-style license, which allows for the redistribution and use (commercial or otherwise), in either source code or binary form, provided that its copyright notice and disclaimer are included.

5.2.2 Version

The latest version tested with libGE is 2.4.6.

5.2.3 Installation

The GALib has been tested with many different architectures, and should be easy to configure and install. After download, uncompress the package. Inside the created directory there will be a '`makevars`' file, edit it to match your system configuration. Then use the standard `make` and `make install` commands (you might want to check the contents of the '`Makefile`' as well).

Note: version 2.4.6, when installing, does not copy the file '`std_stream.h`' to the destination headers directory (along with files like '`ga.h`', etc). This is a bug; you must do it yourself. You can find the file in the '`ga/`' sub-directory, inside the directory created when you first uncompressed the distribution package.

5.2.4 Main features

The code boots an array of different representations and genetic operators ready to be used; it relies on templates to allow the implementation of these representations with any data type. Major features are:

- Many different algorithms available: simple GA, steady state GA, incremental GA, deme GA.
- Many scaling and selection schemes: linear scaling, power law scaling, etc (for scaling), and rank selection, roulette-wheel, tournament, etc (for selection).
- Many representations available: lists, trees, binary strings, arrays up to three dimensions, etc.
- Although it is quite exhaustive, the code is still remarkably fast and easy to use.

5.2.5 Usage

The best way to learn how to use the GALib is to try out some of the examples provided. The documentation has an overview section that might help to get familiarised with the code. In its simplest form, a GALib source code consists of a main function, where the type of GA is chosen (e.g. steady-state), its parameters are set, and the evolution cycle is called, and a function containing the problem to be solved. Once more control is required, the code becomes slightly more complex, but is still quite easy to use and understand.

5.2.6 Examples

The GALib package provides 27 example files, covering all major aspects of the code.

5.2.7 Documentation

The documentation is well structured and complete. It is available online at the website, and also as a pdf document for download.

5.2.8 Using with libGE

Being a C++ library, compiling the GALib with libGE is quite trivial. The main task at hand is to transform the genetic structure you are using into a libGE `Genotype` object. To implement the standard GE mapping, you should choose one of the linear variable-length structures available in GALib, specifically `GA1DArrayGenome<T>` or `GAListGenome<T>`, with the type chosen being an `int` or `unsigned int`. If you do choose to use one of these, libGE has already incorporated support for them, so you can use a call to the `setGenotype` from the `Mapper` class in libGE, to automatically assign a GALib genome to a mapper. Otherwise, you will have to transform that genome into a `Genotype` object, prior to assigning it to the mapper you are using.

Here is some pseudo-code showing how to use GALib and libGE, when using supported structures from the former:

```
Objective function (GAGenome)

    Cast GAGenome into structure chosen (e.g. GAListGenome);

    Assign it to mapper used;

    Recover phenotype;

    If phenotype is valid, recover fitness;

    Return fitness value.

End of function
```

Some of the operators provided might not follow strictly the original GE implementation. To see how to write operators in GALib for GE, check some of the examples provided (see Chapter 7 [Examples], page 57).

5.3 EO

- http://eodev.sourceforge.net/

The EO (Evolving Objects) library is an extremely complete evolutionary computation library, written in C++. It relies heavily on templates, so that if a specific kind of evolutionary computation technique isn't directly supported, it should be easy to build it using the existing abstract or concrete classes.

5.3.1 License

The EO library is released under the GNU Lesser General Public License.

5.3.2 Version

The latest version tested with libGE is 0.9.4.

5.3.3 Installation

At the time of writing this, EO is not updated frequently, so the best choice is to use the latest CVS version. Once the code has been downloaded, first use the `./autogen.sh` script (which calls `./configure`) to configure the installation; then just run `make` (and `make install` if you wish a system-wide installation).

Note: you can't have a global installation of both EO and GALib, the reason being that both libraries create a '`libga.a`' file. It is therefore recommended that you install at least one of the libraries (possibly even both) locally.

5.3.4 Main features

EO is probably one of the most complete evolutionary computation libraries available. It is designed so that the implementation of practically any evolutionary algorithm is possible, and at the same time the resulting algorithm will be easy to use for the end user. Its main features are:

- Extremely complete code: literally hundreds of classes are available.
- Fully customisable: heavy usage of templates means the code can be adapted to suit practically all search algorithms.
- Complete technical reference: usage of `doxygen` on all source files results in a full (albeit rather unpersonalised) technical documentation.
- Code more, compile less: EO takes quite a while to compile (mainly due to its heavy usage of templates), but the features provided (including a powerful command line parser for arguments, as well as the ability of saving partial runs and restarting them later) require no recompilations for parameter tuning.

5.3.5 Usage

EO is not the easiest evolutionary package to use, probably because of being so complete. If one chooses to use the template files available (check the Tutorial, Lesson five), then the code can be quite easy to implement, but not easily tunable. For a fully tunable, practically ready to use base code, the best approach is to start with the code available in Lesson four of the Tutorial and change it.

Using EO requires some good knowledge of templates in C++, as well as a good general knowledge of evolutionary computation; even simple things like implementing a standard GA solving a onemax problem can seem like a daunting task at the start. Emphasis seems to have been put on simplicity for the end user, and not the programmer; writing an algorithm with EO is a hard task, but the end result is an extremely simple to use and tune algorithm for the end user.

5.3.6 Examples

Not many examples are provided with EO. Probably the best source of examples is the Tutorial, with its five lessons.

5.3.7 Documentation

There is a Tutorial available on-line and for download, which is the best way to get familiarised with EO. Also, a reference document is provided, generated automatically by `doxygen`, as well as some publications and presentations available on the website.

5.3.8 Using with libGE

As EO does not have any in-built data structure (a variable-length string of bits or integers) that directly satisfies the requirements for GE, a specific genotype structure has to be implemented, as well as genetic operators to work on that structure. Check some of the examples provided on how to do this (see Chapter 7 [Examples], page 57), as well as the EO tutorial.

Probably the best way to implement a GE genotype is to create a class that uses a linear structure, such as a `vector`, and to derive that class from the `EO<T>` class (mandatory when using EO). There are a few mandatory member functions for this class, such as the `printOn()` and `readFrom()` methods, which need to be implemented. Operators also need to be implemented on their own separate classes (see Chapter 7 [Examples], page 57).

After that, you only need to implement your evaluation function, again on a class of its own (deriving from `eoEvalFunc<T>`). Use the required initialisation procedures (such as reading a grammar, etc) on the constructor. Then implement a `operator()`, called a *functor*, which implements the actual fitness function: it requires the translation of your genome structure into a libGE `Genotype` structure and assigning it to your mapper (using the method `setGenotype()`), which will then provide a phenotype (through the method `getPhenotype()`) to evaluate.

The actual mapper to use can be located in the main file, for example. Check the examples provided for more information (see Chapter 7 [Examples], page 57).

5.4 Using your own Search Engine

If you do not use any of the search engines listed in this section, then libGE has not been (officially) tested with that search engine as of yet. In that case, all you have to provide is a way to transform the genotypic structures used by the search engine onto `Genotype` structures from libGE. If using the C++ language, you will find that one of the several constructors available for the `Genotype` class might be suitable for that transformation.

As for the genetic operators, if you would like to implement a strict GE implementation, you will might have to implement your own, to ensure that bit-level mutation is used, and crossover at codon boundary.

6 Evaluators

A typical usage of libGE involves choosing a search engine and an appropriate mapping process, and applying them to a problem of choice. In many cases, this problem might be external to the algorithm; a typical example would be a real world application, where an interface is required with a physical simulator, or an actual piece of hardware executing the evolved code. This chapter illustrates how to interface libGE with an external evaluator, and gives examples of how to use the evaluators which have so far been successfully tested with libGE.

6.1 GCC

- http://gcc.gnu.org/

When evolving code using Grammatical Evolution, an obvious choice is to use a compiler to execute the evolved code, and capture back its fitness (that is, the goodness of the evolved code). While using GCC (the GNU C Compiler) might seem a bit of overkill, it might be the only solution at times. Also, interfacing with GCC or with any other compiler is a similar process, so the guidelines on how to use GCC as an evaluator can be easily generalised to other compilers.

6.1.1 License

The GCC compiler is distributed under the GNU General Public License.

6.1.2 Version

The latest version tested with libGE is 4.0.2.

6.1.3 Installation

The GCC compiler follows the standard GNU installation process, so after download, uncompress the package. Inside the created directory, run the `./configure` script, followed by `make` and `make install`.

6.1.4 Main features

The GCC compiler is full of features, far too many to mention. It is the compiler of choice for practically all Linux distributions, and can also be used under different operating systems, such as BDS variants, MacOS, Windows, etc. In addition, it supports multiple languages, such as C, C++, Objective-C, Assembler, Fortran, Ada and Treelang.

6.1.5 Usage

The simplest usage of the GCC compiler is to call it using the name of the file containing the code to be compiled as argument: `gcc code.c`. A multitude of switches are available (several hundred), to control code optimisation, output executable name, include file and library locations, etc.

6.1.6 Examples

No examples as such are provided by GCC, although its test suite is available.

6.1.7 Documentation

The manual page provided is extremely complete, if somewhat hard to read. The website also contains a FAQ list, as well as web pointers to GCC documentation and tutorials. In addition, there are countless books and documents about GCC, both available online and to buy.

6.1.8 Using with libGE

Whenever an individual is ready for evaluation, its evolved code must be written to a file. This file might also need some header and footer code, depending on the grammar used. For example, if the code evolved is simply a mathematical expression, such as

```
y=3*5+2*3*sin(5)
```

then it will need to be surrounded by code which will make it a valid C program:

```
#include<math.h> int main(){ return 3*5+2*3*sin(5);}
```

In order to capture the return value of the function, several options are available. A simple approach is to write the value into a file, transforming the code into:

```
#include<stdio.h>
#include<math.h>
int main(){ printf(%f\n",3*5+2*3*sin(5));}
```

The output of the program, once compiled, can then be captured into a file. A system call would then be needed to compile and execute the code; so after asserting the validity of the phenotype string, and writing all the C code around it, it should be written to a file (e.g. "phenotype.c"), and a system call must be made to compile it, and a second call to execute the program. For example:

```
system("gcc -pipe individual.c");
system("a.out > result");
```

The first line compiles the code, using the GCC flag `-pipe` to speed up somewhat the compilation process. The second line then executes the compiled code, retrieving its output into a file called '`result`'; the objective function can then read the contents of that file to receive the output of the phenotype code. These two system calls can be merged into one:

```
system("gcc -pipe individual.c; a.out > result");
```

Note that system calls (and the compilation process) are very expensive processes, and therefore the use of a compiler like GCC as an evaluator should only be considered in cases where there is no viable alternative. However, in certain cases, such as when using a generational replacement in a genetic algorithm, GCC can be much faster, as all individuals can be written into one single file and be compiled at the same time.

6.2 S-Lang

- http://www.s-lang.org/

S-Lang is a multi-platform programmer's library designed to allow a developer to create robust multi-platform software. It provides facilities required by interactive applications such as display/screen management, keyboard input, keymaps, and so on. The most noteworthy feature of the library is the slang interpreter, that may be easily embedded into a program to make it extensible.

6.2.1 License

The S-Lang library is distributed under the GNU General Public License.

6.2.2 Version

The latest version tested with libGE is 2.0.4.

6.2.3 Installation

The S-Lang library also follows the standard GNU installation process, so after download, uncompress the package. Inside the created directory, run the `./configure` script, followed by `make` and `make install`.

6.2.4 Main features

The S-Lang is a feature-rich environment. It can be seen as made up of two main parts:

- The S-Lang language. This is a scripting language, with a syntax close to C/C++. It has many interesting features from a GE point of view, such as:
  - It is an interpreted language;
  - It is machine-independent;
  - It allows for dynamic typing;
  - It allows for on-the-fly function declarations.
- The S-Lang library. This is a library that allows the interpretation of S-Lang programs from within other programs. It has a powerful interface which allows for the execution of S-Lang statements through the program in which it is embedded, including:
  - IO statements (file and screen);
  - Math statements;
  - Variable interfacing.

6.2.5 Usage

Typically, a piece of code is written in S-Lang, and an interpreter is then called to execute this code. The interpreter can be the S-Lang shell `slsh`, included with the installation package, or the S-Lang library. If the latter is used, then specific library functions are required to initialise the interpreter, load code, execute code, and establish variable interfaces.

6.2.6 Examples

The package contains a small set of example programs written in S-Lang. There are no obvious examples of how to interface the S-Lang interpreter with a program written in another language, such as C/C++.

6.2.7 Documentation

The documentation is fairly complete, if somewhat hard to follow. It is available on the project's website, and is composed of four main parts:

- A Guide to the S-Lang Language;
- S-Lang Library Programmer's Guide;
- Intrinsic Function Reference;
- C Library Reference.

6.2.8 Using with libGE

To use S-Lang with libGE, the first step is to include S-Lang's header (which will depend on the location of the S-Lang library in your system):

```
#include<slang.h>
```

The next step is to initialise the S-Lang library; assuming S-Lang is being used to evaluate mathematical expressions, then code similar to the following would have to be executed:

```
if ((-1 == SLang_init_slang()) /* basic interpreter functions */
|| (-1 == SLang_init_slmath())) /* math intrinsics */
exit(0);
```

Next, an interface needs to be established between the S-Lang interpreter and the program, by associating a local variable (such as `fitness`) to a S-Lang variable (such as `fit`):

```
if (-1==SLadd_intrinsic_variable("fit",&(fitness),SLANG_DOUBLE_TYPE, 0)){
exit(0);
```

Finally, the mathematical code can be loaded and executed inside the S-Lang interpreter:

```
if (-1 == SLang_load_string ("fit=3*5+2*3*sin(5)"))
exit(0);
```

If the code is successful, then the variable `fitness`, which is interfaced with the S-Lang variable `fit`, will contain the result of the mathematical expression.

Note that since the S-Lang library used the math library, your compilation string should look something like this (depending on your compiler, and on the location of includes and libraries):

```
gcc test.c -lslang -lm
```

6.3 TinyCC

- http://tinycc.org/

TinyCC is a small but really fast C compiler. It includes an assembler and linker, so there is no need to use external programs for that. Also, it comes bundled with libtcc, a library that provides a backend to execute dynamically generated code.

6.3.1 License

TinyCC is distributed under the GNU Lesser General Public License.

6.3.2 Version

The latest version tested with libGE is 0.9.23.

6.3.3 Installation

The project uses the standard `./configure`, `make` and `make install` sequence, even though the configure script is not GNU compliant (i.e., it is not generated by `autoconf` and `automake`). This means that a command like `make install prefix=/your/path` will **not** work; use `./configure --prefix=/your/path` instead, prior to compilation.

6.3.4 Main features

Speed is the main feature of TinyCC. The website claims gains of speed of up to 9x when compared with GCC. The libtcc part is particularly useful for GE, as it provides a backend to compile and execute evolved code. Both the ANSI C and ISO C99 standards are supported, as well as most of the GNUC extensions. Additionally, it also provides a memory and bound checker. Finally, it has started support for the Windows platform as well.

6.3.5 Usage

To compile a C program, just invoke the compiler with the name of the file as an argument: `tcc code.c`. TinyCC can also be used to interpret shell scripts written in C language. Additionally, it can also be invoked from within a program, through libtcc.

6.3.6 Examples

The package comes bundled with five small examples of C programs and scripts; one example of how to use libtcc is also provided.

6.3.7 Documentation

The documentation is available both in the website and in the distribution package, and consists of a single (long) HTML file. A man page is also available and installed. The documentation for the usage of libtcc is practically non-existent (although the example file given is pretty self-explanatory), and its usage requires parsing the '`libtcc.h`' file for info on the methods available.

6.3.8 Using with libGE

Usage of libtcc is pretty straightforward. The first thing to do is to include the '`libtcc.h`' file (whose location depends of where it was installed):

```
#include<libtcc.h>
```

Next, a `TCCState` needs to be created and initialised:

```
TCCState *s=tcc_new();
```

The next step consists in setting the output type to memory, that is, to set the code to be compiled and ran in memory:

```
tcc_set_output_type(s,TCC_OUTPUT_MEMORY);
```

A string containing the code is then compiled:

```
tcc_compile_string(s,"double f=3+5*2;");
```

The code inside the state then needs to be *relocated*:

```
tcc_relocate(s);
```

After these steps, the code is ready and compiled inside the `TCCState`, and an interface can then be established. This is done by extracting the address of the variable `f` from within the state (through a `unsigned long` variable), and capturing it with a pointer of the same type (`double`):

```
unsigned long val;
double *pf;
if(tcc_get_symbol(s,&val,"f")) exit(0);
pf=(double*)(val);
```

If the compilation and interface were successful, the result should be in `*pf`. The `TCCState` can then finally be deleted:

```
tcc_delete(s);
```

Note that libtcc is a dynamic library, so your compilation string should look something like this (depending on your compiler, and on the location of includes and libraries):

```
gcc test.c -ltcc -ldl
```

Usage of lictcc gets substantially more complicated as more features are required. Please refer to the tcc documentation and to the examples bundled with libGE (see Chapter 7 [Examples], page 57).

6.4 Lua

- http://www.lua.org/

Lua is a small language that aims to be embeddable and extendable by providing *meta-mechanisms* for implementing features. So while it does not directly provide OOP features, it allows for such concepts to extend the language. It can both be extended by C functions as well as being embedded into C. Lua also supports anonymous functions similar to lambda functions in LISP and dialects. Lua is dnamically typed.

6.4.1 License

Until Lua 4.0, Lua used its own license, which was very close to the zlib license and others, but not quite the same. Starting with Lua 5.0, Lua is licensed under the terms of the MIT license; this is a very unrestricted license, whose sole purpose is to provide the code "as is", with no restrictions to modify or distribute it, but illibating the developpers of any responsbilities.

6.4.2 Version

The latest version tested with libGE is 5.0.2.

6.4.3 Installation

Lua is written in pure ANSI C, so any ANSI C compliant compiler should be able to compile it. It contains a `configure` script, which is **not** created by `autoconf`; its purpose is solely to give information about possible compilation options. A simple `make` (and optional `make install`) should compile Lua.

6.4.4 Main features

Lua is an extension programming language designed to support general procedural programming with data description facilities. It also offers good support for object-oriented programming, functional programming, and data-driven programming. Lua is intended to be used as a powerful, light-weight configuration language for any program that needs one. Lua is implemented as a library, written in clean C (that is, in the common subset of ANSI C and C++).

Being an extension language, Lua works embedded in a host program. This program can invoke functions to execute a piece of Lua code, can write and read Lua variables, and can register C functions to be called by Lua code.

The Lua distribution includes a stand-alone embedding program, `lua`, that uses the Lua library to offer a complete Lua interpreter.

6.4.5 Usage

Working with Lua might not be immediately straightforward, because of its extensive use of a stack, which can hold compiled code, results from function calls, and is also used for communication with the host program.

A Lua program can be interpreted by the included `lua` interpreter; if using the C API, then a Lua state has to be created, and a Lua program can then be loaded into it and executed.

6.4.6 Examples

A series of Lua source code examples are included in the '`test`' directory of the distribution package. Additionally, many examples are given in the various documents available.

6.4.7 Documentation

The documentation available for Lua is extensive. There is a user manual available in HTML, both in the website and in the distribution package; this manual is also offered in PS and PDF formats. The website also offers a FAQ list, a set of seminar slides, a series of Lua Technical Notes, a book (available both in print and online), and all the academic papers written about (or using) Lua. Finally, the http://lua-users.org/ website offers a wiki, including a tutorial.

6.4.8 Using with libGE

The first thing that needs to be done is to include the Lua include files (whose location depends on your system's installation):

```
#include <lua.h>
#include <lualib.h>
#include <lauxlib.h>
```

Note that if you're using Lua from within a C++ program, then the include files should be surrounded by a "C" wrapper:

```
extern "C" {
#include <lua.h>
#include <lualib.h>
#include <lauxlib.h>
}
```

Next, a `lua_State` needs to be created and initialised:

```
lua_State *l=lua_open();
```

If you are using mathematical functions, the `lua_State` needs to be loaded with its mathematical library:

```
luaopen_math(l);
```

Assume the program evolved with GE is contained in a string:

```
char *program="f=3*5+2*3*math.sin(5)";
```

The next step consists in compiling the code, through a call to `luaL_loadbuffer()`. If the compilation is successful, the call returns zero and the resulting code chunck is pushed on the stack. A call to `lua_pcall()` then loads the code chunck from the stack, and runs it in memory, in protected mode; again, if there are no errors, zero is returned. If there was an error with either of these functions, an error message is pushed into the stack, which can be retrieved by the `lua_tostring()` function:

```
error = luaL_loadbuffer(l, program, strlen(program), "line") ||
lua_pcall(l, 0, 0, 0);
if(error){
fprintf(stderr, "%s", lua_tostring(l, -1));
```

```
lua_pop(l, 1); /* pop error message from the stack */
}
```

Assuming the compilation was successful, a call to `lua_getglobal()` is then required, to place the value contained in variable `f` on top of the stack. Once that is done, the `lua_tonumber()` function can retrieve the value of that variable from the top of the stack (index `-1`, and place it on a (previously declared) variable (`fit`):

```
lua_getglobal(l, "f");
fit=lua_tonumber(l,-1);
```

Note that Lua uses two libraries (the actual Lua code routines, and the C API routines), so you have to link to both; furthermore, if you are using its mathematical instructions, it needs to link to the math library. So your compilation string should look something like this (depending on your compiler, and on the location of includes and libraries):

```
gcc test.c -llua -llualib -lm
```

6.5 Using your own Evaluator

To use your own evaluator (such as an interpreter, a simulator, a piece of code, etc), you need to create an interface with it. The phenotype structures created with libGE can be handled as strings or as a sequence of symbols; you should choose the representation that best meets your needs.

7 Examples

This chapter describes the examples included in the '`EXAMPLES`' directory. Most of these examples are standard applications of evolutionary programming, such as the Santa Fe Ant Trail problem or Symbolic Regression problems. Each problem includes one or more examples of usage of different search engines and evaluators.

7.1 Santa Fe Ant Trail Problem

The Santa Fe Ant Trail is a standard problem in the area of Genetic Programming, and Grammatical Evolution has been applied to it on several occasions. The objective is to find a computer program to control an artificial ant, so that it can find all 89 pieces of food located on a 32 by 32 toroidal grid (i.e. a grid with no borders: top and bottom are connected, as are its left and right sides). The ant can turn left, turn right, move one square forward, and may also look ahead one square in the direction it is facing, to determine if that square contains a piece of food. All actions, with the exception of looking ahead for food, take one time step to execute. The ant starts in the top left-hand corner of the grid, facing the first piece of food on the trail.

7.1.1 Grammar

The grammar used is equal for most evaluators, and is contained in a file called '`grammar.bnf`'. Specific grammars exist, for evaluators that have different languages (e.g. '`grammar-lua.bnf`').

7.1.2 Ant Trail

The '`santafe.trl`' file describes the Santa Fe Ant Trail. Modify this file only if you wish to try out different trails.

7.1.3 GE_ILLIGALSGA

This example, contained in the directory '`EXAMPLES/SantaFeAntTrail/GE_ILLIGALSGA`', shows how to modify the code of the sga-c code from the IlliGAL laboratory to use libGE with the Santa Fe Ant Trail problem. In order to use this problem, you will need to replace the appropriate files in your distribution of the sga-c code with the files provided. The following sections describe those files.

7.1.3.1 'app.c'

This file contains the source code required to run the Santa Fe Ant Trail problem with the sga-c code, using GCC as a compiler for the evolved phenotype strings. Only the functions `app_init` and `objfunc` need code for this problem.

The function `app_init` initialises the libGE interface and mapper. It starts by setting the maximum number of wrapping events allowed, through a call to the wrapper function `GE_ILLIGALSGA_setMaxWraps`, and proceeds by reading the file containing the BNF grammar, through a call to the wrapper function `GE_ILLIGALSGA_readBNF` (see Section 5.1 [IlliGAL sga-c], page 39).

The function `objfunc` contains the code to map the genotypic structure passed as an argument onto a phenotypic structure, and the evaluation of the mapped structure. It starts by assigning the `struct individual` passed as an argument to this function as the genotype of the mapper used, through a call to the wrapper function `GE_ILLIGALSGA_Mapper`, and checks for the validity of the resulting phenotype, through a call to the wrapper function `GE_ILLIGALSGA_validPhenotype`. If valid, then a file is created, containing the phenotype code and some header and footer code, required to compile that phenotype. Once the file is created, it is compiled and executed through a system call, and the result of that execution, which was redirected to a file called '`result`', is then read and assigned to the critter structure.

7.1.3.2 'Makefile'

The '`Makefile`' distributed with this example shows how to link the sga-c code to the libGE library.

Few changes are required to the original Makefile, distributed with the sga-c code. The `LDLIBS` variable must include *`-lGE`*; both the '`GEant.c`' and '`GEtrail.c`' files must be compiled (with a C compiler); and the linking of all object files should be done with a C++ compiler (in the file provided, the `g++` compiler is used).

7.1.4 GE_MITGALIB

The example using MIT's GALib uses a `GAListGenome<unsigned char>` as the GALib genotype structure, with a steady-state genetic algorithm. Here follows a user's guide to the example, and a description of its files.

7.1.4.1 GALib Example User Guide

This example of GALib code uses the standard `GAParameterList` way of passing arguments to the program, to specify the parameters of the run. The GE parameters were encoded in a similar way, by a simple parsing of the command-line.

Parameters are passed to the executable in the following fashion:

```
./executable <param_name> <value>
```

The following are the parameters accepted by GALib, and their default values:

- popsize: population size (default is 100);
- ngen: population size (default is 10);
- pcross: crossover probability (default is 0.9);
- pmut: mutation probability (default is 0.01);
- prepl: steady-state replacement strategy (default is 1.0);
- sfreq: how often to record scores (generation interval) (default is 1);
- ffreq: how often to dump scores to file (generation interval) (default is 1);
- sfile: name of score data file (default is "sf-output.dat").

All other parameters are set to the GALib defaults (check the GALib documentation for more details). To this list, the following GE specific parameters were added:

- seed: random seed to be used both by GALib and libGE (default is random);
- grammar: grammar file to use (default is "grammar.bnf");
- wrap: number of wrapping events (default is 0);
- sensible: use sensible initialisation instead of random (default is 0);
- min: minimum genotype size for random initialisation (default is 15);
- max: maximum genotype size for random initialisation (default is 25);
- grow: grow rate for sensible initialisation (default is 0.5);
- maxDepth: maximum tree depth for sensible initialisation (default is 10);
- tailSize: tail size for sensible initialisation (default is 0);
- tailRatio: tail ratio for sensible initialisation (default is 0.0).

For example, to run the `GELUA` implementation, using the grammar file "grammar-lua.bnf", with a population size of 500 individuals, for 50 generations, using random seed 1, and leaving all other parameters to standard values, the command to execute is:

```
./GELUA grammar grammar-lua.bnf popsize 500 ngen 50 seed 1
```

7.1.4.2 'main.cpp'

This file drives the evolutionary process. It starts with the declaration of all functions required, and the definition of a `GEGrammarSI` mapper, as well as global integers containing size information for random initialisation, and a counter for the number of objective function calls.

After displaying the version of libGE, it starts by checking the command-line for all arguments; first the parameters related to libGE are looked for, and the corresponding variables are set; afterwards, a `GAParameterList` object is created, to look for all GALib specific parameters (such as population size, number of generations, etc).

Next comes the creation of a genome structure; in this example, the genome is a `GEListGenome`, defined on the files '`GEListGenome.h`' and '`GEListGenome.cpp`'. The fitness function, called `objfunc`, is associated with the genome.

The genetic operators to use are specified next. A specific initialisation function (`initFuncRandom()` or `initFuncSI()`), contained in the file '`initfunc.cpp`', is associated with the genome. This is followed by the choice of crossover to use, which is either the the standard one-point crossover or an effective length version, and to use a specific mutation function (`PointMutator()`), all defined in the file '`GEListGenome.h`'.

A call to the function `app_init()` follows, which loads the code to place before and after each evolved phenotype prior to evaluation, and initialises the libGE mapper; this function is contained in the files '`santafe-*.cpp`' (with '`*`' replaced with the name of the evaluator used).

Next comes the creation of the genetic algorithm `ga`, which uses a steady-state replacement strategy; the `GAParameterList` containing all parameters is associated with it. The mapper is also initialised, using either default values or the values obtained when parsing the command-line arguments. Finally, the statistics to record are selected, a custom statistics file is created, and a call to the `initialize()` method is made, to apply all settings to the genetic algorithm, including the random seed.

An evolutionary cycle is then performed. This can either be done directly with `ga.evolve()`, or on a step by step cycle, as shown in the example. Also note how two statistics files are created, one using the GALib standard outputs, and a second one, illustrating the creation of a personalised statistics file.

Finally, once the evolution is finished, the statistical data is dumped onto the screen, as is the best individual of the population (through a call to the function `print_individual()`, defined in the files '`santafe-*.cpp`').

7.1.4.3 'initfunc.cpp'

This file contains the two specific initialisation functions designed for this example, `initFuncRandom()` and `initFuncSI()`. The first function is called to randomly initialise each genome from the population. It starts by destroying any data possibly contained on an individual, and chooses a number between `minSize` and `maxSize` (declared and initialised in '`main.cpp`') for its size. It then initialises each gene to be a number between 0 and 255, starting with the head of the list.

The second function (`initFuncSI()`) can be called to use libGE's sensible initialisation routines to initialise each genome from the population. It starts by destroying any data possibly contained on an individual, and then uses the `init()` method from the mapper

to apply the sensible initialisation routines. If successful, each codon from the mapper's `Genotype` structure is then copied onto the genome.

7.1.4.4 'GEListGenome.h'

This file declares the `GEListGenome` class, which is the genome used in this example. It derives from GALib's `GAListGenome` class, using the type `unsigned char` as its elements, as an `unsigned char` is usually composed of eight bits, which is the standard number of bits per gene used in Grammatical Evolution.

This class extends the `GAListGenome` by allowing the genome to record its effective length, that is, the portion of the genome that was actually used during the mapping process. It also provides two methods, `effCrossover()` and `pointMutator()`, which implement a crossover restricted to the effective lengths of the genomes and GE's standard bit-level mutation, respectively. The implementation of all these methods is in the file '`GEListGenome.cpp`'.

7.1.4.5 'santafe-gcc.cpp'

This file contains the functions directly related to the evaluation of each individual, made specifically for usage with the GCC compiler. It starts by defining two strings, `SFstart` and `SFend`, to contain buffer text for compilation, and a reference to the `GEGrammarSI` mapper, defined in the '`main.cpp`' file.

The `app_init` function, called from the '`main.cpp`' file, is responsible for reading in buffer code to place before and after each individual, before they are passed to the compiler for evaluation. It also initialises the libGE mapper, setting the maximum number of wrap events, and loading the grammar file.

The `objfunc` function is responsible for evaluating an individual (passed as an argument). The `setGenotype()` method from the mapper is used to create a `Genotype` structure within the mapper, from the genome passed as argument. The resulting phenotype is then checked, and if valid, then a C file containing the individual is created; this file will contain the starting buffer code, the phenotype code, and the end buffer code. It is then compiled and linked with the '`GEant.o`' and '`GEtrail.o`' files, containing an interpreter for Santa Fe Ant Trail programs, and executed, all through a system call. The resulting program saves the output of the program onto a '`result`' file, which is then read, and the contents of that file are assigned as the fitness score of the current individual. Finally, the effective size of the individual is returned back to the genome.

Finally, the `print_individual` function receives a genome as argument, performs its mapping to a phenotype structure, and prints it onto the screen, along with measurements of the genotype, generated phenotype, and the mapping process itself.

7.1.4.6 'santafe-slang.cpp'

This file contains the functions related to the evaluation of each individual, made specifically for evaluation with the S-Lang library. As with the previous file, it makes reference to the `GEGrammarSI` mapper, defined in the '`main.cpp`' file.

The `app_init` function, called from the '`main.cpp`' file, is responsible for initialising the libGE mapper, setting the maximum number of wrap events, and loading the grammar file. It also initialises the S-Lang interface, loading both the '`GEant.sl`' and '`GEtrail.sl`' files,

which are S-Lang versions of the same interpreter for the Santa Fe Ant Trail problem (as used with the GCC compiler). It also calls the `ReadTrailGEtrail` function, defined within those files, to load the '`santafe.trl`' file. Finally, an interface with the S-Lang interpreter is created, through a call to the function `SLadd_intrinsic_variable`, to link the variable `fitness` to the S-Lang variable `Fitness_variable`.

The `objfunc` function is again responsible for the evaluation of a genome passed as an argument. The `setGenotype()` method from the mapper is used to create a `Genotype` structure within the mapper, from the genome passed as argument. The resulting phenotype is then checked, and if valid, then a S-Lang function containing the individual is created, and wrote onto the string `buffer`, containing some buffer start and ending code, and the phenotype code. The function contained in the buffer is then executed with the S-Lang interpreter, and the resulting fitness is used as the fitness score of the genome. Finally, the effective size of the individual is returned back to the genome.

Finally, the `print_individual` function receives a genome as argument, performs its mapping to a phenotype structure, and prints it onto the screen, along with measurements of the genotype, generated phenotype, and the mapping process itself.

7.1.4.7 'santafe-tcc.cpp'

This file contains the functions directly related to the evaluation of each individual, made specifically for usage with the TCC compiler. It starts by defining two strings, `SFstart` and `SFend`, to contain buffer text for compilation, and a reference to the `GEGrammarSI` mapper, defined in the '`main.cpp`' file.

The `app_init` function, called from the '`main.cpp`' file, is responsible for reading in buffer code to place before and after each individual, before they are passed to a `TCCState` for evaluation. It also initialises the libGE mapper, setting the maximum number of wrap events, and loading the grammar file.

The `objfunc` function is responsible for evaluating an individual (passed as an argument). The `setGenotype()` method from the mapper is used to create a `Genotype` structure within the mapper, from the genome passed as argument. The resulting phenotype is then checked, and if valid, then a `TCCState` is created with `tcc_new()`, its output type is set to memory (`tcc_set_output_type()`), and the file '`GEantfuncs.o`' is added to it (`tcc_add_file()`), which contains the object code for the functions called from the phenotype code (`left()`, `right()`, etc). A buffer is then created, containing the phenotype string and the start end end buffer code, and it is compiled inside the `TCCState` (with `tcc_compile_string()`). If the compilation is successful, the `main()` function is run by calling `tcc_run()`, and its return value is assigned as the fitness score of the current individual. Finally, the effective size of the individual is returned back to the genome.

Finally, the `print_individual` function receives a genome as argument, performs its mapping to a phenotype structure, and prints it onto the screen, along with measurements of the genotype, generated phenotype, and the mapping process itself.

7.1.4.8 'santafe-lua.cpp'

This file contains the functions related to the evaluation of each individual, made specifically for evaluation with the Lua evaluator. It includes all the required Lua include files, wrapped with a `extern "C"` definition, as Lua is a C library and this example is written in C++. As

with the previous files, this file makes reference to the `GEGrammarSI` mapper, defined in the '`main.cpp`' file. A global `lua_State` is also created at the top of the file.

The `app_init` function, called from the '`main.cpp`' file, is responsible for initialising the libGE mapper, setting the maximum number of wrap events, and loading the grammar file. It also creates the `lua_State`, initialising its base, I/O and string libraries. It then proceeds to load both the '`GEant.lua`' and '`GEtrail.lua`' files (through a call to the `luaL_loadfile()` function), which are Lua versions of the interpreter for the Santa Fe Ant Trail problem (as used with all the other evaluators), and executes them (through a call to `lua_pcall()`). Finally, it executes the `ReadTrailGEtrail` function (with the function `luaL_loadbuffer`), defined within those files, to load the '`santafe.trl`' file.

The `objfunc` function is once again responsible for the evaluation of a genome passed as an argument. The `setGenotype()` method from the mapper is used to create a `Genotype` structure within the mapper, from the genome passed as argument. The resulting phenotype is then checked, and if valid, then the string `buffer` is loaded with the phenotype code and some start and ending code, and passed to the `lua_State` for evaluation. If successful, the `lua_getglobal()` function is used to push into the Lua stack the value of the `_picked_up` variable, and the `fitness` variable is used to receive that value from the stack (through the use of the `lua_tonumber()` function). Finally, the effective size of the individual is returned back to the genome.

Finally, the `print_individual` function receives a genome as argument, performs its mapping to a phenotype structure, and prints it onto the screen, along with measurements of the genotype, generated phenotype, and the mapping process itself.

7.1.4.9 '`Makefile`'

The '`Makefile`' contained with the example allows you to create a program using a choice of evaluators for your phenotype strings. Use `make` to create all executables, or `make GE*` to create a specific evaluator implementation (where `*` is any of `GCC`, `SLANG`, `TCC` or `LUA`). At the top of the file there are definitions for the location of the libraries, make sure you adjust these to suit your system.

Some of the evaluators require the pre-compilation of the '`GEant.c`' and '`GEtrail.c`' files; this is done by the '`Makefile`', when the relevant target implementations are chosen.

7.1.5 GE_EO

The example using the EO library uses a custom genotype structure, as EO does not have direct support for variable-length integer-based structures. Here follows a user's guide to this implementation, and a description of its files.

7.1.5.1 EO Example User Guide

The EO library has a powerful command-line parser, which is fully-configurable. This example uses that parser to specify both the standard GA parameters, and also the GE specific parameters.

Parameters are passed to the executable in the following fashion:

```
./executable -param_short_name[=value]
```

or

```
./executable --param_long_name[=value]
```

The following are the parameters accepted by this example, and their default values:

- -h, –help: prints help message (default is 0);
- –stopOnUnknownParam: stop if unknown parameter entered (default is 1);
- -C, –pCross: probability of crossover (default is 0.9);
- -E, –effCross: use effective crossover (default is 0);
- -M, –pMut: probability of mutation (default is 1);
- -b, –pMutPerBit: probability of bit-flip mutation (default is 0.01);
- -s, –sensibleInit: use sensible initialisation (default is 0);
- -x, –minSize: minimum size for random initialisation (default is 15);
- -X, –maxSize: maximum size for random initialisation (default is 25);
- -r, –grow: grow ratio for SI (default is 0.5);
- -d, –maxDepth: maximum depth for SI (default is 10);
- -T, –tailSize: tail size for SI (default is 0);
- -t, –tailRatio: tail ratio for SI (default is 0);
- -g, –grammarFile: grammar file (default is "grammar.bnf");
- -w, –wrappingEvents: number of wrapping events (default is 0);
- -L, –Load: save file to restart run (no default);
- -S, –Status: status file (default is "./<name_of_executable>.status");
- -R, –seed: sandom number seed (default is random);
- -P, –popSize: population size (default is 100);
- -G, –maxGen: maximum number of generations (default is 10).

For example, to run the `GELUA` implementation, using the grammar file "grammar-lua.bnf", with a population size of 500 individuals, for 50 generations, using random seed 1, and leaving all other parameters to standard values, the command to execute is:

```
./GELUA -g=grammar-lua.bnf -P=500 -G=50 -R=1
```

7.1.5.2 'main.cpp'

This file contains the `main()` function, and is the central point of control over the evolutionary process. It begins by including all required EO include files, followed by a declaration of the fitness type to be used, the type of each individual, and finally the declaration (and required include) of the libGE mapper to use.

The function `main_function()` follows. It starts by defining all parameters available to the system in a `eoParser` object, along with code related to it (such as printing help messages).

The objective function is declared next, along with an objective function call counter, `eoEvalFuncCounter`. Finally, a population of individuals is declared.

An `eoState` is then declared, on which the random generator and population are registered, and which either loads a previous existing population from disk (if the run is a continuation of a previous run), or calls the individual initialiser to create a new population. In the latter case, each individual is also evaluated.

The definition of roulette-wheel selection and steady-state replacement (GE style) follow, along with the definition of the genetic operators to use (which are encapsulated into a `eoTransform` object). Finally, a series of continuators, checkpoints, statistics objects and monitors are defined, and associated together.

At the end of the `main_function()`, the actual evolutionary algorithm is then declared (an `eoEasyEA` object), and associated with the checkpoints, evaluation function, selection procedure, genetic operators, and replacement mechanism. The evolutionary cycle consists merely of associating it to the previously declared population.

The actual `main()` function consists only of a protected call to `main_function()`, to handle exceptions.

7.1.5.3 'eoGE.h'

This file declares the `eoGE` class, defining a genome. The structure chosen is a vector of `unsigned char` elements, as these are typically represented by 8 bits, which is the standard GE representation of a codon.

The methods `printOn()` and `readFrom()` output or read a genome, respectively, from the specified stream. The method `getEffectiveSize()` merely returns the recorded effective size of this individual.

7.1.5.4 'eoGEInit.h'

This file declares the `eoGEInit` class, defining the initialisation procedure used to create individuals. The constructor reads, processes and stores the required parameters, to either select a random initialisation, or a sensible initialisation.

In the case of a random initialisation, the method `operator()` calls the method `randomInit()`. This method creates individuals between the sizes `minSize` and `maxSize`, containing random `unsigned char` numbers.

If the sensible initialisation is selected, the mapper's `init()` method is called, and if successful, the contents of its `Genotype` structure are used to initialise the `_genotype` object passed as an argument.

7.1.5.5 'eoGEMutation.h'

This file declares the `eoGEMutation` class, which implements a point mutation operator, as used in standard GE. The mutation is applied to genotypes passed as arguments to the `operator()` method (the probability of mutation is specified through a call to the constructor).

7.1.5.6 'eoGEQuadCrossover.h'

This file declares the `eoGEQuadCrossover` class, which implements the crossover operator to use during the evolutionary cycle. Its constructor receives a boolean argument stating whether to use a standard 1-point crossover, or an effective crossover.

If a standard 1-point crossover is to be used, a random cut point is selected within each individual; if the effective version is to be used, each cut point must lie within the effective part of each individual (that is, the part of its genome which has been read to create a phenotype). After choosing the cut points, the second half of each individual is then swapped.

7.1.5.7 'eoGEEvalFuncGCC.h'

This file declares the `eoGEEvalFunc` class, which contains the fitness function to use on each individual, implemented using the GCC evaluator. Its constructor receives as arguments the name of the file containing the grammar, and the number of wrapping events; these are used to initialise the mapper passed as reference. The private strings `SFstart` and `SFend` are then loaded with buffer code, to be placed before and after each individual, before they are passed to the compiler for evaluation.

The method `operator()`, receiving an individual as argument, updates the fitness score of that individual. If the individual is considered *invalid* (which, in EO terms, means it has been modified and requires re-evaluation), a vector of integers is created with its contents, and used to initialise the `Genotype` structure of the libGE mapper used. If the resulting `Phenotype` structure is not valid, a fitness of 0 is assigned; otherwise, the generated code is wrapped with the start and end buffer code and written to a file; this file is then compiled and linked to the '`GEant.o`' and '`GEtrail.o`' files, containing an interpreter for Santa Fe Ant Trail programs, and finally it is executed, through a system call. The resulting program saves the output of the program onto a '`result`' file, which is then read, and the contents of that file are assigned as the fitness score of the current individual.

7.1.5.8 'eoGEEvalFuncSlang.h'

This file declares the `eoGEEvalFunc` class, which contains the fitness function to use on each individual, implemented using the S-Lang evaluator. Its constructor receives as arguments the name of the file containing the grammar, and the number of wrapping events; these are used to initialise the mapper passed as reference. The constructor also initialises the S-Lang interface, to be used to run the evaluation of each individual; it loads the files '`GEant.sl`' and '`GEtrail.sl`', containing definitions specific to the Santa Fe Ant Trail problem, and executes the code `ReadTrailGEtrail("santafe.trl")`, which is run on the S-Lang interpreter, and which loads a trail definition into memory. Finally, an interface with the S-Lang interpreter is created, through a call to the function `SLadd_intrinsic_variable`, to link the private variable `fit` to the S-Lang variable `Fitness_variable`.

The method `operator()`, receiving an individual as argument, updates the fitness score of that individual. If the individual is considered *invalid* (which, in EO terms, means it has been modified and requires re-evaluation), a vector of integers is created with its contents, and used to initialise the `Genotype` structure of the libGE mapper used. If the resulting `Phenotype` structure is not valid, a fitness of 0 is assigned; otherwise, the generated code is wrapped with some buffer code, and then passed onto the S-Lang interpreter, which returns the fitness of that code through the `fit` variable, which is finally assigned to the individual.

7.1.5.9 'eoGEEvalFuncTCC.h'

This file declares the `eoGEEvalFunc` class, which contains the fitness function to use on each individual, implemented using the TCC evaluator. Its constructor receives as arguments the name of the file containing the grammar, and the number of wrapping events; these are used to initialise the mapper passed as reference. The private strings `SFstart` and `SFend` are then loaded with buffer code, to be placed before and after each individual, before they are passed to the compiler for evaluation.

The method `operator()`, receiving an individual as argument, updates the fitness score of that individual. If the individual is considered *invalid* (which, in EO terms, means it has been modified and requires re-evaluation), a vector of integers is created with its contents, and used to initialise the `Genotype` structure of the libGE mapper used. If the resulting `Phenotype` structure is not valid, a fitness of 0 is assigned; otherwise, a `TCCState` is created with `tcc_new()`, its output type is set to memory (`tcc_set_output_type()`), and the file '`GEantfuncs.o`' is added to it (`tcc_add_file()`), which contains the object code for the functions called from the phenotype code (`left()`, `right()`, etc). A buffer is then created, containing the phenotype string and the start end end buffer code, and it is compiled inside the `TCCState` (with `tcc_compile_string()`). If the compilation is successful, the `main()` function is run by calling `tcc_run()`, and its return value is assigned as the fitness score of the current individual.

7.1.5.10 'eoGEEvalFuncLua.h'

This file declares the `eoGEEvalFunc` class, which contains the fitness function to use on each individual, implemented using the Lua evaluator. It includes all the required Lua include files, wrapped with an `extern "C"` definition, as Lua is a C library and this example is written in C++.

The constructor receives as arguments the name of the file containing the grammar, and the number of wrapping events; these are used to initialise the mapper passed as reference. It also creates the `lua_State`, initialising its base, I/O and string libraries. It then proceeds to load both the '`GEant.lua`' and '`GEtrail.lua`' files (through a call to the `luaL_loadfile()` function), which are Lua versions of the interpreter for the Santa Fe Ant Trail problem (as used with all the other evaluators), and executes them (through a call to `lua_pcall()`). Finally, it executes the `ReadTrailGEtrail` function (with the function `luaL_loadbuffer`), defined within those files, to load the '`santafe.trl`' file.

The method `operator()`, receiving an individual as argument, updates the fitness score of that individual. If the individual is considered *invalid* (which, in EO terms, means it has been modified and requires re-evaluation), a vector of integers is created with its contents, and used to initialise the `Genotype` structure of the libGE mapper used. If the resulting `Phenotype` structure is not valid, a fitness of 0 is assigned; otherwise, the string `buffer` is

loaded with the phenotype code and some start and ending code, and passed to the `lua_State` for evaluation. If successful, the `lua_getglobal()` function is used to push into the Lua stack the value of the `_picked_up` variable, and the `fit` private variable variable is used to receive that value from the stack (through the use of the `lua_tonumber()` function).

7.1.5.11 'Makefile'

The '`Makefile`' contained with the example allows you to create a program using a choice of evaluators for your phenotype strings. Use `make` to create all executables, or `make GE*` to create a specific evaluator implementation (where `*` is any of `GCC`, `SLANG`, `TCC` or `LUA`). At the top of the file there are definitions for the location of the libraries, make sure you adjust these to suit your system.

The '`Makefile`' uses the `-include` compiler flag to use only the relevant include file for your choice of evaluator; if your compiler does not accept that flag, you should replace it with the appropriate flag.

Finally, some of the evaluators require the pre-compilation of the '`GEant.c`' and '`GEtrail.c`' files; this is done by the '`Makefile`', when the relevant target implementations are chosen.

7.1.6 Santa Fe Ant Trail Performance

This section looks at the performance obtained with most of the search engines tested, for this specific problem. All search engines used a similar experimental setup: steady-state replacement, population size of 500 individuals, 50 generations, probability of crossover of 0.9, and probability of bit-wise mutation of 0.01. Furthermore, each experiment was performed with four different setups:

- Using a standard GE approach;
- Using effective crossover;
- Using sensible initialisation;
- Using a combination of the last two.

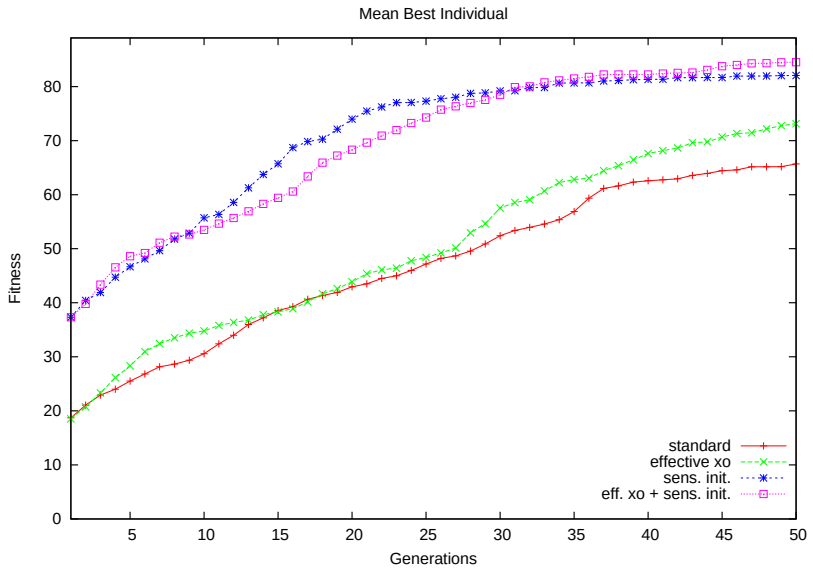
The results obtained show that the last setup is clearly the best. Although the results varied somewhat between search engines, they tend to be equivalent. Interestingly, EO performance tended to benefit more from the use of effective crossover, which was not the case with GALib.

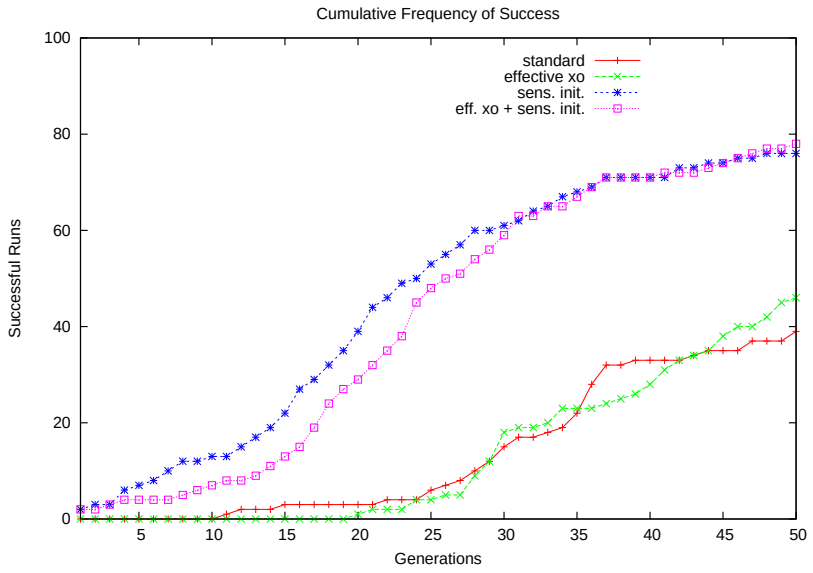
As for using which evaluator to use, the results obtained were identical with all the evaluators tested. The following list gives an approximation of the execution times to expect from each of the evaluators tested:

- GCC: average of 52:00.00 per run;
- SLANG: average of 1:57.51 per run;
- Lua: average of 1:17.22 per run;
- TCC: average of 0:45.01 per run.

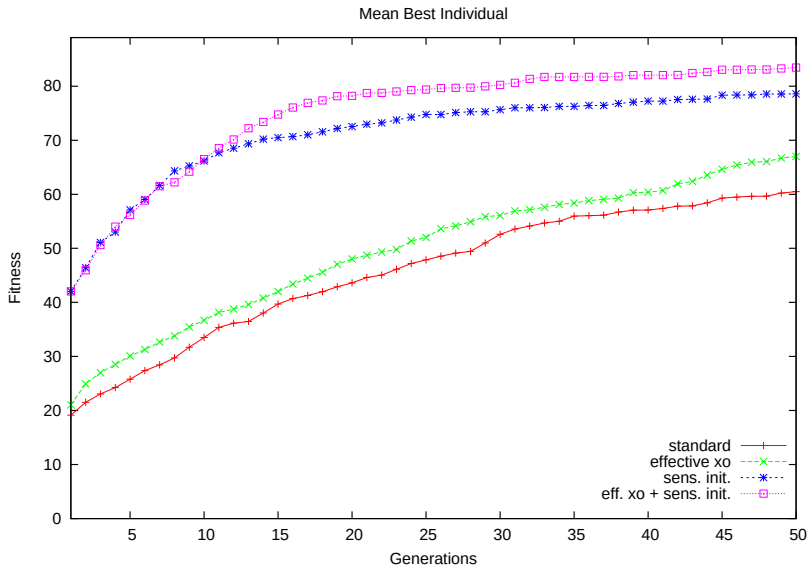
For reference, these results were obtained on a PC with a Pentium IV processor running at 1.50GHz, 512MB of memory, using Linux.

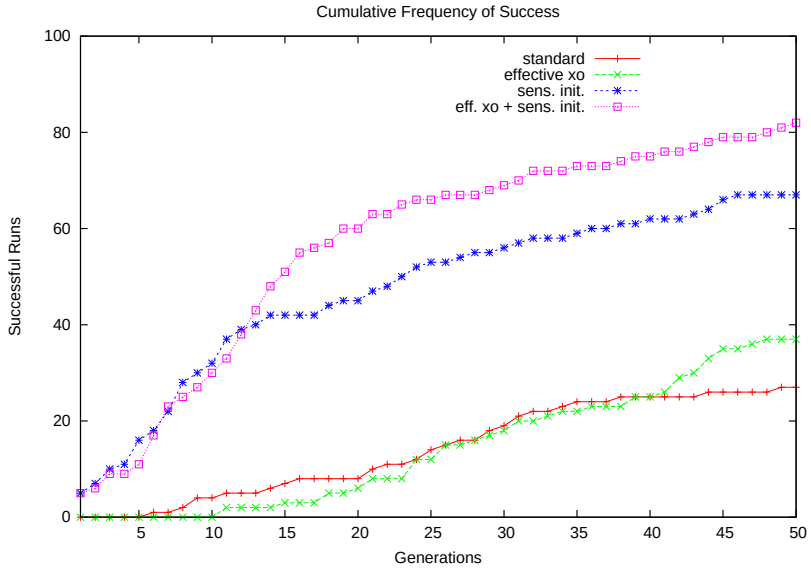
7.1.6.1 GALib performance





7.1.6.2 EO performance





7.2 Cart Centering Problem

The cart centering problem is a control optimisation problem in which an object of fixed mass, often referred to as a wheeled cart, must achieve a state of rest at the center of a one-dimensional track in minimal time. The state of the cart at any time can be described by two variables: its velocity along the track v; and its position relative to the center of the track x. The state of the cart can be changed by the control variable u. This value represents the direction in which a force f of constant magnitude is applied to the cart. The control variable is limited to either +1 or -1, with a positive value resulting in the application of the force in the positive direction, and a negative value resulting in the application of the force in the negative direction. The force is referred to as the *bang-bang force* and depending on the current state of the cart, has the effect of either accelerating or decelerating its velocity on the track. The problem posed is therefore the generation of a control function which achieves a state of rest at the center of the track in minimal time.

In this implementation, the effect of the control variable on the state of the cart is simulated by re-evaluating the control expression after a specific interval, called a time-step, until a maximum time limit elapses, or the cart achieves the target state.

The change in state due to the bang-bang force after each time-step is calculated using the Euler approximations of higher order differential equations. The following describes the steps taken to calculate the change in state using this method.

First, the following equation for Newton's second law of motion is applied:

$a(t) = F(t)/m$

in which $a(t)$ is the acceleration of the cart due to the bang-bang force at the current time-step t, $F(t)$ is the magnitude of the force applied at the current time-step, and m is the mass of the cart.

Once the acceleration is known, we can calculate the velocity part of the new state of the cart at the next time-step $(t + \tau)$ using the following equation.

$v(t + \tau) = v(t) + \tau a(t)$

The new position of the cart can then be calculated using:

$x(t + \tau) = x(t) + \tau v(t)$

The value of the time-step τ can contribute significantly to the amount of error produced by these equations. As a result, this value is usually kept small $(0.0 < \tau <= 0.2)$.

The values $x(t + \tau)$ and $v(t + \tau)$ describe the state of the cart after the force is applied. The Euclidean distance between this state and the target state can be calculated using the following equation.

$\epsilon = \sqrt{(x_{curr} - x_{target})^2 + (v_{curr} - v_{target})^2}$

in which x_{curr} and v_{curr} represent the current state of the cart, and x_{target} and v_{target} represent the target state. In typical implementations, the target position is the origin and the target velocity is 0.0. The cart is said to have achieved the target state when $epsilon = 0.0$.

In most implementations of this problem however, the distance between the current and target state is compared with a maximum threshold value. If the distance is less than this value, then the cart is said to have achieved the target state. The optimal solution then becomes that which reduces the distance between current target states to the threshold

distance in minimal time. The number of time-steps is also limited to a maximum simulation time t_{max}.

7.2.1 The Optimal Solution

The optimal solution to the cart centering problem must achieve the target cart state from any initial state in minimal time. The solution shown in the next set of equations is proven to be optimal for the cart centering problem and was successfully evolved using Genetic Programming (GP) by Koza. This equations show how the value of the control variable is calculated from the current state of the cart.

$u(t) = +1, -x(t) > v(t)^2\ Sign\ v(t)$

$u(t) = -1 - x(t) < v(t)^2\ Sign\ v(t)$

The *Sign* function returns +1.0 if the argument is a positive integer and −1.0 if the argument is less than or equal to zero.

Note: in their paper on the use of Grammatical Evolution (GE) for this problem, Azad et al. found many solutions which appeared to be more successful than this (Azad et al. 2002).

7.2.2 Grammar

Four BNF grammars are supplied with the cart centering example in the libGE distribution. These grammars differ in the number of non-terminals and the number of production rules used to structure the language of the problem. An example grammar is shown below:

```
<expr>  ::= (<expr> <op> <expr>)
         | DIV(<expr>, <expr>)
         | ABS(<expr>)
         | GT(<expr>, <expr>)
         | <var>
<op>    ::= + | - | *
<var>   ::= X | V | (-1.0)
```

In this grammar, the terminal `DIV` is a protected version of the standard division arithmetic operator in which division by zero is prevented, the terminal `ABS` performs the same functions as the `fabs` function found in the C language, and the terminals `X` and `V` describe the current state of the cart. The remaining non-terminals `+`, `-`, `*`, `>` are arithmetic operators. Floating point data types are used for all operations and variables. This and other example grammars for the cart centering problem can be found in the directory: '`EXAMPLES/CartCentering/grammars`'

7.2.3 '`GEcart.c`'

The file '`GEcart.c`' provides a number of variable and function declarations that are required to evaluate GALib individuals using the GCC, TCC, and libTCC backends. This section documents the use of each of these functions and variables in the cart centering problem. The following list describes each variable and its purpose.

- `X`: the current position of the cart. This variable is specified in the grammar as a terminal.

- `V`: the current velocity of the cart. This variable is specified in the grammar as a terminal.
- `currTime`: the current time.
- `numFitCase`: the number of fitness cases to be tested.
- `startValuesArray`: a 2-D array containing the initial position and velocity values for each of the fitness cases. Initial values for twenty fitness cases are provided as an example in this file.
- `numFitCase`: the index of the current fitness case being tested.
- `sumFitCaseTimes`: the total time required by all completed fitness cases.
- `hitCount`: the number of fitness cases for which a solution was found that centered the cart before the maximum time elapsed.
- `maxTime`: the maximum time allowed to center the cart.
- `overMaxTimePenalty`: the time penalty added to the `sumFitCaseTimes` variable if the cart is not centered before the maximum time elapses.
- `maxHitEuclError`: the maximum euclidean distance allowed between the cart's state and the target state (0,0) for a hit to be registered.
- `currEuclError`: the euclidean distance between the cart's current state and the target state (0, 0).
- `timeStep`: the amount of time to advance at each simulation step.
- `posForceVelChange`: the effect on the velocity of the cart due to a positive bang-bang force.
- `negForceVelChange`: the effect on the velocity of the cart due to a negative bang-bang force.
- `velocityChange`: the effect on velocity of the last force applied to the cart i.e. stores either the value of `posForceVelChange` or `negForceVelChange`.

The function `initCart` must be called before simulation starts. It initialises the variables relating to fitness cases and invokes the `resetCart` function.

The `resetCart` function sets the value of the variables `X` and `V` to the initial values specified in the current fitness case (see variable `startValuesArray`). The initial euclidean distance is also calculated. This function must also be called before simulation starts.

The function `stepSim` steps the simulation by one `timeStep`. The effect of applying the force to the cart is simulated by adding the value in the variable `velocityChange` to the current velocity variable `V`. The current position `X` is also recalculated based on the new velocity. The current time variable (`currTime`) is then updated and the euclidean distance between the current and target states of the cart is calculated and stored in `currEuclError`. If the distance is less than `maxHitEuclError`, the cart has achieved the target state and the simulation stops. The current time is added to the total time required for all previous fitness cases (`sumFitCaseTimes`) and the hit counter `hitCount` is incremented by one. The fitness case index `currFitCase` is also incremented by one and the function `resetCart` is then called to the set the initial values of the cart's state for the next fitness case.

If the distance is greater than `maxHitEuclError`, the current time is compared with the maximum allowed time `maxTime`. If it is less than the maximum time, the function simply exits. If the maximum time has elapsed however, the value `overMaxTimePenalty` is added

to the current fitness case total `sumFitCaseTimes`. The fitness case index `currFitCase` is then incremented by one and the function `resetCart` is called.

The binary function `DIV` is a protected form of the division operator which returns a value of 1.0 if the denominator is 0.0. The binary function `GT` compares the two arguments and returns 1.0 if the first argument is greater than the second or -1.0 otherwise. The unary function `ABS` returns the absolute value of the argument e.g. $ABS(-3.5) = 3.5$. These three functions are specified in the grammar as terminals.

7.2.4 'cartcenterstart.c'

The file '`cartcenterstart.c`' contains the C code that is prepended to a GE phenotype at the evaluation stage and contains a number of references to the functions and variables declared in '`GEcart.c`'. It also contains the start of the `main` function which is called when the compiled program is executed. This function calls the `initCart` and `resetCart` function described in the previous section.

The function then constructs a while loop to execute the simulation for each fitness case. A fitness case in this example is a pair of initial position and velocity values i.e. a fitness case defines the initial state of the cart.

The body of the loop only contains the start of an `if` statement in order to allow the insertion of the GE phenotype expression at compile time. Once the phenotype is inserted, further C code is required to form a valid C program. That code is described in Section 7.2.5 [EXAMPLES/CartCentering/cartcenterend.c], page 75.

7.2.5 'cartcenterend.c'

The code in the file '`cartcenterend.c`' completes the `if` statement started in the file '`cartcenterstart.c`' (see Section 7.2.4 [EXAMPLES/CartCentering/cartcenterstart.c], page 75). If the value of the phenotype expression is greater than 0.0, then a positive force is applied to the cart. If the value is less than or equal to 0.0, then a negative force is applied. Forces are applied by assigning the value `posForceVelChange` or `negForceVelChange` to the variable `velocityChange`. The function `stepSim` is then called to advance the simulation by one time step.

Once the while loop condition evaluates to false, the total time taken to center the cart for all the fitness cases (`sumFitCaseTimes`) and the number of hits achieved (`hitCount`) are printed to standard output.

7.2.6 Example wrapped C phenotype

The following code segment shows Koza's optimum solution prepended and appended with the start and end code described previously.

```
int main(void)
{
  initCart();
  resetCart();
  while (currFitCase < numFitCase)
  {
    if (GT(((-1.0) * X), (V * (ABS(V)))) > 0.0)
```

```
    {
      velocityChange = posForceVelChange;
    }
    else
    {
      velocityChange = negForceVelChange;
    }

    stepSim();
  }

  printf("%f %d\n",
         sumFitCaseTimes,
         hitCount);

  return (0);
}
```

7.2.7 GE_ILLIGALSGA

The ILLIGAL SGA implementation of the cart centering problem using libGE can be found in '`EXAMPLES/CartCentering/GE_ILLIGALSGA`'. In order to use this problem, you will need to replace the appropriate files in your distribution of the sga-c code with the files provided. The following sections describe the various files provided in the ILLIGAL SGA cart centering example.

7.2.7.1 'app.c'

This file contains the source code required to run the cart centering problem with the sga-c code using the GCC C compiler for the evolved phenotype strings. This file contains definitions for the functions `app_init` and `objfunc` which are required by sga-c. A new type called `Real` is declared at the top of this file as a substitute for the type `float`. The function `objfunc` uses this type to store the result of the phenotype execution. The `typedef` is provided here as a facility in case greater precision is required.

The function `app_init` is responsible for initialising the libGE mapper, load the BNF grammar, and prepare any resources required by the function `objfunc`. It starts by setting the maximum number of wrapping events allowed, through a call to the wrapper function `GE_ILLIGALSGA_setMaxWraps` It then loads the grammar file into the mapper using the function `GE_ILLIGALSGA_readBNF`. If this operation fails, an error is printed to standard error output and the program exits.

The function then loads the contents of two files into character arrays. The files '`cartcenterstart.c`' and '`cartcenterend.c`' contain the C code that is prepended and appended to the phenotype code to construct a valid and compilable C program. The content is loaded into the character arrays `CCstart` and `CCend`. If either of these operations fail, an error message is printed and the program exits. These files are discussed in more detail in Section 7.2.4 [EXAMPLES/CartCentering/cartcenterstart.c], page 75 and Section 7.2.5 [EXAMPLES/CartCentering/cartcenterend.c], page 75.

The function `objfunc` contains the code to map the genotypic structure passed as an argument onto a phenotypic structure and and to evaluate that phenotype using the GCC compiler. It starts by assigning the `struct individual` passed as an argument to the GE mapper using the function `GE_ILLIGALSGA_Mapper`. The validity of the phenotype generated by the mapper is then accessed by calling the function `GE_ILLIGALSGA_validPhenotype`. If the phenotype is not valid, a fitness of 0 is assigned to the individual and the function exits. If the phenotype is valid, then evaluation can continue.

Evaluation of the phenotype is carried out by creating a plain text file called '`individual.c`' and writing the phenotype code wrapped with the start and end code read into the `CCstart` and `CCend` char buffers in the function `app_init`. Once the concatenated program code has been written to the file it is closed and compiled using by invoking a system call to the C compiler.

The actual compiler used is dependent on the definition of a preprocessor macro. If the macro `GECART_USE_TCC_COMP` is defined, then the TinyCC compiler `tcc` is used to compile the file. If this macro is not defined however, the GNU compiler `gcc` is used. The system call comprises the name of the compiler program, the name of the file containing the phenotype and wrapper code (in this case '`individual.c`'), and a directive to link the resulting executable with the math library '`m`' and the object file `GEcart.o`. The source file of this object, called '`GEcart.c`' contains functions and variables that are required by the phenotype code in order to perform the cart centering simulation. For more information on this file, see Section 7.2.3 [EXAMPLES/CartCentering/GEcart.c], page 73. The system call also invokes the execution of the compiled program and redirects all standard output to a file called `result`. If any of these operations fail, the program prints an error message and exits.

If the previous operations were successful, the file `result` should contain the total time required to center the cart for the fitness cases defined in '`GEcart.c`'. The fitness value is read from the file and assigned to the variable `fitness`. The value is then inverted because the problem requires that the total time taken to center and stop the cart be minimised. Inverting the value allows a fitness-maximising GA to interpret the fitness correctly. The value is then assigned to the fitness field in the `struct individual` passed as an argument to the function.

7.2.7.2 'Makefile'

The file '`Makefile`' distributed with this example shows how to link the sga-c code to the libGE library.

Few changes are required to the original Makefile, distributed with the sga-c code. The `LDLIBS` variable must include *`-lGE`*; both the '`GEant.c`' and '`GEtrail.c`' files must be compiled (with a C compiler); and the linking of all object files should be done with a C++ compiler (in the file provided, the `g++` compiler is used).

7.2.8 GE_MITGALIB

The example using MIT's GALib library can be found in '`EXAMPLES/CartCentering/GE_MITGALIB`'. The following sections contain a user's guide and a description of the files in this example.

7.2.8.1 GALib Example User Guide

This example uses the standard `GAParameterList` for passing arguments to specify the parameters of the run. The paramaters specific to GE can also be specified on the command line.

Parameters are passed to the executable in the following fashion:

```
./executable <param_name> <value>
```

The following list shows the parameters accepted by GALib with their default values:

- popsize: population size (default is 100);
- ngen: population size (default is 10);
- pcross: crossover probability (default is 0.9);
- pmut: mutation probability (default is 0.01);
- prepl: steady-state replacement strategy (default is 1.0);
- sfreq: how often to record scores (generation interval) (default is 1);
- ffreq: how often to dump scores to file (generation interval) (default is 1);
- sfile: name of score data file (default is "cc-output.dat");
- setttings_file: the name of a file with GALib configuration parameters;

All other parameters are set to the GALib defaults (check the GALib documentation for more details).

The following parameters specific to GE can also be specified on the command line.

- seed: random seed to be used both by GALib and libGE (default is random);
- grammar: grammar file to use (default is "grammar.bnf");
- wrap: number of wrapping events (default is 0);
- sensible: use sensible initialisation instead of random (default is 0);
- min: minimum genotype size for random initialisation (default is 15);
- max: maximum genotype size for random initialisation (default is 25);
- grow: grow rate for sensible initialisation (default is 0.5);
- maxDepth: maximum tree depth for sensible initialisation (default is 10);
- tailSize: tail size for sensible initialisation (default is 0);
- tailRatio: tail ratio for sensible initialisation (default is 0.0);
- effective: use effective crossover instead of one point crossover (default is 0);

For example, to run the `GELUA` implementation, using the grammar file "grammar-lua.bnf", with a population size of 500 individuals, for 50 generations, using random seed 1234, and leaving all other parameters to standard values, the command to execute is:

```
GELUA grammar grammar-lua.bnf popsize 500 ngen 50 seed 1234
```

7.2.8.2 'main.cpp'

This file drives the GALib evolutionary process. It starts with the declaration of all functions required to set up and perform the GALib evolution, a `GEGrammarSI` genotype-to-phenotype mapper, global integers specifying the minimum and maximum size information for random initialisation, and a counter for the number of objective function calls made by GALib.

The most important element of this file however is the function `main`. After displaying the current version of libGE, the function creates the variables for the parameters specific to GE e.g. the location of the grammar file, the maximum number of wrapping events allowed, the use of sensible initialisation, etc. Additional variables are created to store the location of a GALib parameters file and the random seed to use. Each of the arguments passed to the program on the command line are then read and values are assigned to the variables when matched.

A `GAParameterList` is then created to specify defaults for a number of GALib parameters including population size, number of generations, crossover and mutation probabilities, etc. If a settings file was specified on the command line, parameters are read from this file by calling the `GAParameterList::read` member function.

Once the parameters have been read, the genome structure is created. In this example, a class called `GEListGenome` is provided (in files '`GEListGenome.h`' and '`GEListGenome.cpp`') which derives from the GALib class `GAListGenome`.

The function `main` then specifies the genetic operators to be employed during the evolution process. The most important of all operators, the fitness operator, is associated with the genome using the member function `GAListGenome::evaluator`.

A number of fitness function implementations are provided in the cart centering example for GALib which use a different backend platform to evaluate the GE phenotype. Each of the evaluation backends are implemented in files named '`cartcenter-*.cpp`' with '`*`' replaced by the name of the backend e.g. '`cartcenter-slang.cpp`'. The fitness functions in these files, which are all called `objfunc`, implement the function interface required by GALib i.e. they must take a single argument of a GAGenome and return a fitness value. The particular backend fitness function that called at execution time is determined during compilation process. It is therefore sufficient at this point to associate the function name `objfunc` with the genome using the function `GAListGenome::evaluator`. The compilation process is described in more detail later in this section.

The initialisation operator is specified next using the `GAGenome::initialiser` member function. This operator is executed once to generate the initial population (also known as generation 0) of the evolution. In this example, the random initialiser (`initFuncRandom()` is the default operator used. Alternatively, sensible initialisation of the population can be specified by supplying the function `initFuncSI()` to the `GAGenome::initialiser` member function. Both of these initialisation functions are declared in the file '`initfunc.h`' and defined in the file '`initfunc.cpp`',

Two crossover operators are also available in this example. The default operator, implemented in the function `GAListGenome::OnePointCrossover`, is part of GALib and provides standard crossover functionality. The other operator, implemented in the function `GEListGenome::effCrossover` limits the crossover point to locations within the effective region of the genotype. This operator and the mutation operator `GEListGenome::pointMutator` are declared in the '`GEListGenome.h`' and defined in

'GEListGenome.cpp'. The crossover and mutation operators are associated with the GALib genome using the GAGenome::crossover and GAGenome::mutation member functions respectively.

Once all operators have been specified, the main function calls the function app_init(). This function, implemented for each of the backend evaluators supplied in this example, is responsible for initialising the GE mapper, loading the BNF grammar, and preparing the backend for the task of evaluating GE phenotypes. These function implementations are described in further detail in the next section.

The main function then creates an instance of a GALib GAStedyStateGA which is a steady-state genetic algorithm. The parameters of the GA are set by passing the GAParameterList instance to the parameters member function. The GE mapper is also initialised, using either the default values or the values obtained when parsing the command-line arguments. The statistics recording facility of the GA is then customised. Finally, the function main calls the initialize() member function to initialise the GA and prepare it for evolution.

The evolutionary cycle is controlled by a while loop which keeps calling the function step while the function done returns false. After each generation, a number of statistics are printed to the file stream.

Finally, once the evolution is finished, the statistical data is dumped onto the screen, as is the best individual of the population (through a call to the function print_individual(), defined in the files 'cart-*.cpp').

7.2.8.3 'initfunc.cpp'

This file contains the two specific initialisation functions designed for this example, initFuncRandom() and initFuncSI(). The first function randomly initialises each genome in the population. It starts by destroying any data possibly contained in an individual and then chooses a number between minSize and maxSize (declared and initialised in 'main.cpp') for its size. It then initialises each gene to a number between 0 and 255, starting with the head of the list.

The second function (initFuncSI()) can be called to use libGE's sensible initialisation routines to initialise each genome in the population. It starts by destroying any data possibly contained in an individual and then uses the init() method from the mapper to apply the sensible initialisation routines. If successful, each codon from the mapper's Genotype structure is then copied into the genome.

7.2.8.4 'GEListGenome.h'

This file declares the GEListGenome class, which is the genome used in the cart centering example. It derives from GALib's GAListGenome class, using the type unsigned char as its elements, as an unsigned char is usually composed of eight bits, which is the standard number of bits per gene used in Grammatical Evolution.

This class extends the GAListGenome by allowing the genome to record its effective length, that is, the portion of the genome that was actually used during the mapping process. It also provides two methods, effCrossover() and pointMutator(), which implement a crossover restricted to the effective lengths of the genomes and GE's standard bit-level mutation, respectively. The class also provides a number of functions required for

correct operation with GALib including a copy constructor and the `clone`, `copy`, and `equal` functions. The implementation of all these functions is in the file '`GEListGenome.cpp`'.

7.2.8.5 'cartcenter-gcc.cpp'

The file '`cartcenter-gcc.cpp`' contains the functions directly related to the evaluation of each GALib individual, made specifically for usage with the GCC compiler. The following section describes the various files and functions that are part of the C backend for the cart centering example. This section also describes how the default compiler, GCC, can be exchanged for another C compiler called TCC (from the TinyCC distribution). . The `app_init` function, called from the function `main`, is responsible for initialising the GE mapper and setting up the C evaluation backend.

The `GEGrammarSI mapper` object, which is declared in the '`main.cpp`', is initialised by specifying the maximum number of mapping wraps and loading the BNF grammar using the member function `readBNFFile`. Both the number of wrapping events and the path to the grammar file are specified in parameters to `app_init`. If the grammar fails to load, an error message is printed to the standard error stream (`std::cerr`) and the program exits.

The next task in this function is to load the code that will be placed before and after the phenotype which is contained in the files '`cartcenterstart.c`' and '`cartcenterend.c`'. The function loads the contents of these files into the `CCstart` and `CCend` buffers (character arrays). These buffers are then used in the `objfunc` function for every individual that is evaluated using the C compiler backend. For more information about these files, see Section 7.2.4 [EXAMPLES/CartCentering/cartcenterstart.c], page 75, and Section 7.2.5 [EXAMPLES/CartCentering/cartcenterend.c], page 75.

The function `objfunc` provides the definition of the evaluation operator required by GALib. This function takes one argument of type `GAGenome` and returns a floating point value representing the fitness awarded to that `GAGenome` object. In order to evaluate the genome, the GAGenome object has to be cast into its derived type (`GEListGenome`). This allows the extra member functions specified in the class `GEListGenome` to be used in this function.

Once cast, the `GEListGenome` is then assigned to the `mapper` object. A libGE `Phenotype` is then extracted from the same `mapper` object. If the phenotype is invalid, a fitness value of 0.0 is returned and the function exits. If the phenotype is valid, the full executable code needs to be constructed so that the phenotype can be evaluated in the C backend.

The first step in this process is to create a new empty file called '`individual.c`'. The contents of the buffer `CCstart` are written out to this file, followed by the contents of the phenotype, followed by the contents of the buffer `CCend`. The file '`individual.c`' is then closed. An example '`individual.c`' file is shown in Section 7.2.6 [EXAMPLES/CartCentering/WrappedCPhenotype], page 75.

At this point, the file contains a full valid C program which should be compilable by any C compiler. In this example, two different C compilers can be used: GCC and TinyCC. If the macro `GECART_USE_TCC_COMP` was defined when the file '`cartcenter-gcc.cpp`' was compiled, then the TinyCC C compiler (called `tcc`) is used. If the macro was not defined, the GNU C compiler (called `gcc`) is used. The compiler program is called by invoking a `system` call with the name of the compiler program and the name of the file containing the phenotype and wrapper code i.e. '`individual.c`'. The phenotype file is also linked

with the object file produced by the compilation of the file `GEcart.c`. The `system` call also invokes the execution of the compiled program and redirects all standard output to a file called '`result`'. If the program compiles and the program runs successfully, the file '`result`' should contain the total time required to center the cart for the fitness cases defined for the problem.

To extract this time value, the result file is opened and the first floating point value found is assigned to the variable `fitness`. The result file is then closed.

This fitness is then inverted because the problem requires that the total time take to center and stop the cart be minimised. Inverting the value provides GALib with the correct fitness format i.e. GALib is a maximising GA whereas the goal of the cart centering problem is to minimise the total time taken. Before the function returns this value, it sets the effective size of the genotype in the genome object. This value is used by the effective crossover operator.

Finally, the fitness is returned and the function exits.

Finally, the `print_individual` function receives a genome as argument, performs its mapping to a phenotype structure, and prints it onto the screen, along with measurements of the genotype, generated phenotype, and the mapping process itself.

7.2.8.6 'cartcenter-slang.cpp'

This file contains the functions related to the evaluation of each GALib individual using the S-Lang runtime. The following section describes the various files and functions that are part of the S-Lang backend for the cart centering example.

The `app_init` function, called from the `main` function, is responsible for initialising the libGE mapper and initialising the S-Lang interpreter.

The `GEGrammarSI` mapper object used in this function is declared in the file '`main.cpp`'. The number of wrapping events and the path to the grammar are supplied in parameters to this function. The `mapper` object is initialised by specifying the maximum number of mapping wraps and loading the BNF grammar file using the `readBNFFile` member function. If the process of loading the grammar fails, an error message is printed to the standard error stream (`std::cerr`) and the program exits.

The function initialises the S-Lang environment by calling the S-Lang function `SLang_init_slang` and loads the stdio and math runtime libraries by calling `SLang_init_stdio` and `SLang_init_slmath`. If any of these functions fail, the program exits.

Once the S-Lang environment is set up, the file '`GEcart.sl`' is loaded using the function `SLang_load_file`. This file contains the functions and variables required to evaluate a GE phenotype and is only required to be loaded once. In order to extract information from the S-Lang runtime environment, a new C++ variable is added to the environment using the `SLadd_intrinsic_variable` function. This function requires the name of the variable within the S-Lang environment (called `Total_Time` in this example), the address of the variable in the C/C++ runtime (called `fitness` in this example), and the type of data stored in the variable (`SLANG_DOUBLE_TYPE`). If any of the S-Lang functions function fail, the program exits.

The function `objfunc` provides the definition of the evaluation operator required by GALib. This function takes one argument of type `GAGenome` and returns a floating point value representing the fitness awarded to that `GAGenome` object. In order to evaluate the

genome, the GAGenome object has to be cast into its derived type (`GEListGenome`). This allows the extra member functions specified in the class `GEListGenome` to be used in this function.

Once cast, the `GEListGenome` is then assigned to the `mapper` object. A libGE `Phenotype` is then extracted from the same `mapper` object. If the phenotype is invalid, a fitness value of 0.0 is returned and the function exits. If the phenotype is valid, the full executable code needs to be constructed so that the phenotype can be evaluated in the S-Lang backend.

Instead of having to construct the program and write it to a file to be compiled, S-Lang allows the interpretation and execution of the program in memory. This is achieved in this function by creating a string called `buffer` and appending the start, phenotype, and end code to it.

Once the complete program is constructed, the `buffer` string should contain a program that can execute within the S-Lang runtime. The contents of the string are executed in the runtime by calling the function `SLang_load_string`. Once execution has completed, the value in the variable `fitness` should contain the total time taken to center the cart for all the specified fitness cases.

Inverting the value provides GALib with the correct fitness format i.e. GALib is a maximising GA whereas the goal of the cart centering problem is to minimise the total time taken.

Before the function returns this value, it sets the effective size of the genotype in the genome object. This value is used by the effective crossover operator.

Finally, the fitness is returned and the function exits.

Finally, the `print_individual` function receives a genome as argument, performs its mapping to a phenotype structure, and prints it onto the screen, along with measurements of the genotype, generated phenotype, and the mapping process itself.

7.2.8.7 'GEcart.sl'

The file '`GEcart.sl`' provides the variables and functions required to evaluate the GE phenotype for the cart centering problem using the S-Lang runtime. It provides the functions `initCart`. `resetCart`, `stepSim`, and the grammar functions `GT`, `DIV`, and `ABS`. The functionality of the code is the same as that found in the file '`GEcart.c`' and descriptions of the variables and functions can be found in section Section 7.2.3 [EXAMPLES/CartCentering/GEcart.c], page 73.

7.2.8.8 S-Lang start code

The code required to invoke the functions found in '`GEcart.sl`' is much simpler than the code required for the C implementation. Instead of including a number of header files and defining external variables, the functions defined in '`GEcart.sl`' can be called directly. The following code is executed just before the GE phenotype in the `objfunc` function described in section Section 7.2.8.6 [EXAMPLES/CartCentering/GE_MITGALIB/cartcenter-slang.cpp], page 82.

```
initCart();
resetCart();
while(currFitCase < numFitCase) {
  if (
```

The code segment above initialises the cart problem, resets the cart to initial position and velocity and creates a loop to iterate while the current fitness case (`currFitCase`, a variable defined in '`GEcart.sl`') is less than the total number of fitness cases (`numFitCase`, also in '`GEcart.sl`'). An `if` statement is started but the condition part is left empty. This is done to allow the insertion of the GE phenotype code at this point. The remainder of the code is explained in the next section.

7.2.8.9 S-Lang end code

In the previous section, the code segment shows that an `if` statement is started before the GE phenotype expression is evaluated. The remainder of the condition and the true and false branches are shown in the code segment below.

```
  > 0.0)
  {
    velocityChange = posForceVelChange;
  }
  else
  {
    velocityChange = negForceVelChange;
  }
  stepSim();
}
Total_Time = sumFitCaseTimes;
```

If the value of the GE phenotype is greater than 0.0, the force applied to the cart is positive. If the value is less than or equal to 0.0, the applied force is negative. The application of the force is simulated in this example by assigning the appropriate change in velocity to the variable `velocityChange`. See section '`GEcart.c`' for more information on this process.

Once the condition of the outer loop evaluates to false, the total time required to center the cart for all fitness cases is applied to the variable `Total\_Time`. This statement allows the value to be transferred into the C++ runtime through the variable `fitness`.

7.2.8.10 Example wrapped S-Lang phenotype

A phenotype complete with start and end code is shown in the following code segment as an example.

```
initCart();
resetCart();
while(currFitCase < numFitCase) {
  if (GT(((-1.0) * X), (V * (ABS(V)))) > 0.0)
  {
    velocityChange = posForceVelChange;
  }
  else {
    velocityChange = negForceVelChange;
  }
  stepSim();
}
```

```
Total_Time = sumFitCaseTimes;
```

7.2.8.11 'cartcenter-libtcc.cpp'

The file '`cartcenter-libtcc.cpp`' contains the functions directly related to the evaluation of each GALib individual, made specifically for use with the libTCC library (part of the TinyCC distribution). The cart centering example uses the C API of the libTCC library to interpret, execute, and evaluate the GE phenotype. Evaluation of the phenotype in this way results in significant improvements in execution time over language compilers e.g. GCC, and TCC. The following section describes the various files and functions that are part of the libTCC backend for the cart centering example.

The function `app_init` function, called from the function `main` in the file '`main.cpp`' is responsible for initialising the libGE mapper, loading the BNF grammar file, and setting up the resources required to evaluate the GE phenotypes using the libTCC evaluator in the function `objfunc`.

The `GEGrammarSI mapper` object, which is declared in the '`main.cpp`', is initialised by specifying the maximum number of mapping wraps and loading the BNF grammar using the member function `readBNFFile`. Both the number of wrapping events and the path to the grammar file are specified in parameters to `app_init`. If the grammar fails to load, an error message is printed to the standard error stream (`std::cerr`) and the program exits.

In order to minimise the time taken to extract, evaluate and award a fitness to a GE phenotype, the start and end code that is placed before and after a GE phenotype are loaded into character buffers in this function. These buffers are accessed in the function `objfunc` because they are required for every GALib individual. In this example, the C code in the file '`cartcenterstart.c`' and '`cartcenterend.c`' are read into the `char` arrays `CCstart` and `CCend` respectively. The last task in this function is to load the file '`GEcart.c`' into another character buffer called `GEfunc`. This file contains the definitions of the functions and variables used either directly or indirectly by the GE phenotype code. For more information about these files, see Section 7.2.3 [EXAMPLES/CartCentering/GEcart.c], page 73 and Section 7.2.4 [EXAMPLES/CartCentering/cartcenterstart.c], page 75, and Section 7.2.5 [EXAMPLES/CartCentering/cartcenterend.c], page 75.

The function `objfunc` provides the definition of the evaluation operator required by GALib. This function takes one argument of type `GAGenome` and returns a floating point value representing the fitness awarded to that `GAGenome` object. In order to evaluate the genome, the GAGenome object has to be cast into its derived type (`GEListGenome`). This allows the extra member functions specified in the class `GEListGenome` to be used in this function.

Once cast, the `GEListGenome` is then assigned to the `mapper` object. A libGE `Phenotype` is then extracted from the same `mapper` object. If the phenotype is invalid, a fitness value of 0.0 is returned and the function exits. If the phenotype is valid, the full executable code needs to be constructed so that the phenotype can be evaluated in the libTCC backend.

The first task is to set up a new libTCC state which can compile and execute C code. This is achieved by calling the function `tcc_new`. The function `tcc_set_output_type` is then called to specify that compilation of the C code should take place in main memory to maximise performance. The math library `m` is also added to the libTCC state using the `tcc_add_library` function to allow the use of the square root function `sqrt` in the simulation process.

The next step is to make the various functions and variables that are defined in the code stored in the buffer `GEfunc` available in the libTCC state. This is achieved by calling the function `tcc_compile_string`, and supplying the current libTCC state and the buffer containing the code as arguments. It is important to note that the code in the buffer `GEfunc` is not executed at this stage. The purpose of this call is simply to compile the code and make it available to the phenotype code which will be compiled next.

Before the phenotype can be executed in the libTCC state, it must be wrapped with the start and end code stored in the buffers `CCstart` and `CCend`. The `char` buffer `buffer` is used to store the result of concatenating the `CCstart` buffer, the phenotype code, and the `CCend` buffer. An example of this concatenation is shown in Section 7.2.6 [EXAMPLES/CartCentering/WrappedCPhenotype], page 75. This buffer is then passed to the libTCC state using the function `tcc_compile_string`. If the compilation is successful, the libTCC state contains the executable form of the wrapped phenotype, the supporting functions, and the variables required to evaluate the GE phenotype.

In order to execute the code in memory, the `main` function (declared in '`cartcenterstart.c`') must be explicitly invoked. This function is accessed by first calling the function `tcc_relocate`. This makes all function and variable symbols accessible through the C interface and should be called after every call to `tcc_compile_string`. The function `tcc_get_symbol` is then called with a pointer to the libTCC state, a reference to a variable of type `unsigned long`, and the name of the symbol we require ("main"). If the required symbol is found, the `tcc_get_symbol` function stores its address in the `unsigned long` variable. This variable is then cast into a function pointer of type `int (*p_tccMainFunc)()`. The main function can the be invoked in the libTCC state by calling the function through the pointer i.e. `p_tccMainFunc()`.

The function `tcc_get_symbol` is then used again to retrieve the `sumFitCaseTimes` symbol within the libTCC state. The symbol corresponds to a variable of type `Real` which store the total time required to center the cart for all of the fitness cases. In order to access the value of the symbol, the unsigned `long variable val` must be cast to a pointer to a `Real`.. The type `Real` is defined at the top of the file using the `typedef` directive. In this example, the true type of `Real` is `double`. It is important that the true type of Real matches the type of the variable `sumFitCaseTimes` in the file `GEcart.c`. Once the cast is complete, the actual value of the variable `sumFitCaseTimes` is then retrieved from the libTCC state by dereferencing the local pointer variable. The function then deletes the libTCC state using the function `tcc_delete`.

Inverting the fitness value provides GALib with the correct fitness format i.e. GALib is a maximising GA whereas the goal of the cart centering problem is to minimise the total time taken.

Before the function returns this value, it sets the effective size of the genotype in the genome object. This value is used by the effective crossover operator.

Finally, the fitness is returned and the function exits.

Finally, the `print_individual` function receives a genome as argument, performs its mapping to a phenotype structure, and prints it onto the screen, along with measurements of the genotype, generated phenotype, and the mapping process itself.

7.2.8.12 'cartcenter-lua.cpp'

This file contains the functions related to the evaluation of each GALib individual using the Lua evaluator. The cart centerting example uses the C API of the Lua library to interpret, execute, and evaluate the GE phenotype. Evaluation of the phenotype in this way results in significant improvements in execution time over language compilers e.g. GCC, and TCC. The following section describes the various files and functions that are part of the Lua backend for the cart centering example.

In order to access the Lua C API through the C++ code in 'cartcenter-lua.cpp', the #include <header.h> directives must be enclosed in a special directive called extern "C" e.g.

```
extern "C"
{
#include <lua.h>
#include <lauxlib.h>
#include <lualib.h>
}
```

This tells the compiler that the code in these files is C code and not C++ code. The functions in the file 'cartcenter-lua.cpp' make use of the GE GEGrammarSI mapper, defined in the 'main.cpp' file. A global pointer to a lua_State, called L in this example, is also created at the top of the file.for use by all the functions in the file.

The app_init function, called from the main function, is responsible for initialising the libGE mapper and initialising the Lua interpreter.

The GEGrammarSI mapper object used in this function is declared in the file 'main.cpp'. The number of wrapping events and the path to the grammar are supplied in parameters to this function. The mapper object is initialised by specifying the maximum number of mapping wraps and loading the BNF grammar file using the readBNFFile member function. If the process of loading the grammar fails, an error message is printed to the standard error stream (std::cerr) and the program exits.

The Lua state is opened in this function by a call to the lua API function lua_open. Additional math and io libraries are added to this state using the function luaL_openlibs. Once completed, the Lua state is then ready to interpret and execute Lua code.

In order to minimise the time required for individual evaluations in the function objfunc, the Lua code in 'GEcart.lua' is loaded into the state using the function luaL_dofile. This function loads and executes the Lua code in this file which is discussed in more detail in Section 7.2.8.13 [EXAMPLES/CartCentering/GE_MITGALIB/GEcart.lua], page 88. It is important to note that, because the file only contains variable declarations and function implementations, no code is executed at this time. If an error occurs while loading this file, Lua sets a error message at the top of its call stack. Each element on the Lua stack has an index e.g. an index of -1 represents the element at the top of the stack. This index value is supplied to thelua_tostring to retrieve the error message. The message is then printed to standard error output and the program exits.

The function objfunc provides the definition of the evaluation operator required by GALib. This function takes one argument of type GAGenome and returns a floating point value representing the fitness awarded to that GAGenome object. In order to evaluate the genome, the GAGenome object has to be cast into its derived type (GEListGenome). This

allows the extra member functions specified in the class `GEListGenome` to be used in this function.

Once cast, the `GEListGenome` is then assigned to the `mapper` object. A libGE `Phenotype` is then extracted from the same `mapper` object. If the phenotype is invalid, a fitness value of 0.0 is returned and the function exits. If the phenotype is valid, the full executable code needs to be constructed so that the phenotype can be evaluated in the Lua backend.

Instead of writing the phenotype code to a new file, the code is constructed in memory by appending the required start code, phenotype code, and end code to a `string` object (The start and end code are described in Section 7.2.8.14 [EXAMPLES/CartCentering/GE_MITGALIB/LuaStartCode], page 88 and Section 7.2.8.15 [EXAMPLES/CartCentering/GE_MITGALIB/LuaEndCode], page 89).

This code is executed in the Lua runtime by calling the function `luaL_dostring` and passing it a pointer to the Lua state and the buffer containing the code to be executed. If execution fails, the error message is retrieved from the Lua stack and printed to standard error output and the program exits.

If the compilation and execution is successful, the function `lua_getglobal` is called to retrieve the Lua variable `sumFitCaseTimes`. This function pushes the variable on to the top of the stack. The value is checked and retrieved from the stack using the functions `lua_isnumber` and `lua_tonumber`. The value is then assigned to the `fitness` variable.

Inverting the fitness value provides GALib with the correct fitness format i.e. GALib is a maximising GA whereas the goal of the cart centering problem is to minimise the total time taken.

Before the function returns this value, it sets the effective size of the genotype in the genome object. This value is used by the effective crossover operator.

Finally, the fitness is returned and the function exits.

Finally, the `print_individual` function receives a genome as argument, performs its mapping to a phenotype structure, and prints it onto the screen, along with measurements of the genotype, generated phenotype, and the mapping process itself.

7.2.8.13 'GEcart.lua'

The file '`GEcart.lua`' defines the variables and functions required to evaluate the GE phenotype for the cart centering problem using the Lua runtime. It provides the functions `initCart`. `resetCart`, `stepSim`, and the grammar functions `GT`, `DIV`, and `ABS`. The functionality of the code is the same as that found in the file '`GEcart.c`' and descriptions of the variables and functions can be found in Section 7.2.3 [EXAMPLES/CartCentering/GEcart.c], page 73.

7.2.8.14 Lua start code

In order to execute the phenotype in the Lua state, code must be prepended and appended to the phenotype. The following code segment shows the code that is prepended to the phenotype. It contains calls to the `initCart` and `resetCart` functions defined in '`GEcart.lua`'. It also sets up the while loop to continue evaluating the phenotype expression while the current fitness case `currFitCase` is less than the total number of fitness cases `numFitCase`. It also contains the first part of the if statement that will test the value of the phenotype expression.

```
initCart();
resetCart();
while currFitCase < numFitCase do
  if
```

7.2.8.15 Lua end code

The following code is appended to the phenotype to finish the condition of the if statement and provide execution paths for both the true and false cases. If the value of the phenotype expression is greater than 0.0, then a positive force is applied to the cart. If the value is less than or equal to 0.0, then a negative force is applied. Forces are applied by assigning the value `posForceVelChange` or `negForceVelChange` to the variable `velocityChange`. The simulation is then stepped by one time step by calling the function `stepSim`. Once the while loop condition evaluates to false, the total time taken to center the cart for all the fitness cases (`sumFitCaseTimes`) and the number of hits achieved (`hitCount`) are printed to standard output.

```
  > 0.0 then
    velocityChange = posForceVelChange;
  else
    velocityChange = negForceVelChange;
  end
  stepSim();
end
```

7.2.8.16 Example wrapped Lua phenotype

A phenotype complete with start and end code is shown below as an example.

```
initCart();
resetCart();
while currFitCase < numFitCase do
  if GT(((-1.0) * X), (V * (ABS(V)))) > 0.0) > 0.0 then
    velocityChange = posForceVelChange;
  else
    velocityChange = negForceVelChange;
  end
  stepSim();
end
```

7.2.8.17 'Makefile'

The file '`Makefile`' contained in the example allows you to create a program using a choice of evaluators for your phenotype strings. Use `make` to create all executables, or `make GE*` to create a specific evaluator implementation (where `*` is any of `GCC`, `SLANG`, `TCC`, `LIBTCC`, or `LUA`). At the top of the file there are definitions for the location of the GALib, GE, and evaluator specific header and library files. If you have installed any of these in non-standard locations, make sure the values are adjusted to suit your system.

Note: Some of the evaluators require the pre-compilation of the '`GEcart.c`'. This is automatically done by the '`Makefile`' when the relevant evaluators are chosen.

7.2.9 GE_EO

The example using the Evolutionary Objects (EO) library can be found in '`EXAMPLES/CartCentering/GE_EO`'. The following sections contain a user's guide and a description of the files in this example.

7.2.9.1 EO Example User Guide

The EO library has a powerful command-line parser, which is fully-configurable. This example uses that parser to specify both the standard GA parameters, and also the GE specific parameters.

Parameters are passed to the executable in the following fashion:

```
./executable -param_short_name[=value]
```

or

```
./executable --param_long_name[=value]
```

The following are the parameters accepted by this example, and their default values:

- -h, –help: prints help message (default is 0);
- –stopOnUnknownParam: stop if unknown parameter entered (default is 1);
- -C, –pCross: probability of crossover (default is 0.9);
- -E, –effCross: use effective crossover (default is 0);
- -M, –pMut: probability of mutation (default is 1);
- -b, –pMutPerBit: probability of bit-flip mutation (default is 0.01);
- -s, –sensibleInit: use sensible initialisation (default is 0);
- -x, –minSize: minimum size for random initialisation (default is 15);
- -X, –maxSize: maximum size for random initialisation (default is 25);
- -r, –grow: grow ratio for SI (default is 0.5);
- -d, –maxDepth: maximum depth for SI (default is 10);
- -T, –tailSize: tail size for SI (default is 0);
- -t, –tailRatio: tail ratio for SI (default is 0);
- -g, –grammarFile: grammar file (default is "grammar.bnf");
- -w, –wrappingEvents: number of wrapping events (default is 0);
- -L, –Load: save file to restart run (no default);
- -S, –Status: status file (default is "./<name_of_executable>.status");
- -R, –seed: random number seed (default is random);
- -P, –popSize: population size (default is 100);
- -G, –maxGen: maximum number of generations (default is 10).

For example, to run the `GELUA` implementation, using the grammar file "grammar.bnf", with a population size of 500 individuals, for 50 generations, using random seed 1, and leaving all other parameters to standard values, the command to execute is:

```
GELUA -g=grammar.bnf -P=500 -G=50 -R=1
```

7.2.9.2 'main.cpp'

This file contains the `main()` function, and is the central point of control over the evolutionary process. It begins by including all required EO include files, followed by a declaration of the fitness type to be used, and the type of each individual. The typedef directive declares the type `Indi` as a substitute name for the individual type `eoGE<MyFitT>`. The class eoGE is declared in the file '`eoGE.h`' and is discussed in the next section. This example uses a custom genotype structure because EO does not have direct support for variable-length integer-based structures. The file '`main.cpp`' also declares the libGE `GEGrammarSI` mapper.

The function `main_function()` starts by defining all parameters available to the system using an instance of the `eoParser` class. The name of the parameter, the associated help messages, and the default value are supplied to the `eoParser::createParam` member function.

The fitness operator is then declared by creating an instance of the template class `eoGEEvalFunc`. The template in this case concerns the type of individual to be evaluated (called `Indi` in this example). The operator object is supplied with the libGE mapper object and the path of the BNF grammar file to use.

A number of fitness operator classes are provided in the cart centering example for EO which use a differend backend platform to evaluate the GE phenotype. Each of the evaluation backends are implemented in files named '`eoGEEvalFunc-*.cpp`' with '`*`' replaced by the name of the backend e.g. '`eoGEEvalFunc-slang.cpp`'. All the fitness operator classes are called `eoGEEvalFunc` and each implements the interface required by EO i.e. they implement an operator function `()` which takes an eo individual, evaluates it, and then assigns a fitness value. The particular backend fitness operator class created at execution time is determined during compilation. It is therefore sufficient at this point to create an instance of the class using the name `eoGEEvalFunc`. The compilation process is described in more detail later in this section.

The function `main_function` then declares an objective function call counter and a `eoPOP` object to store the population of `Indi` individuals.

The random seed generator and the `Indi` population are then registered with a new instance of the class `eoState`. This class is responsible for initialising and evaluating the population in the case of a new run, or loading the population from disk if continuing a previous run.

The definition of roulette-wheel selection and steady-state replacement (GE style) follow, along with the definition of the genetic operators to use (which are encapsulated into a `eoTransform` object). Finally, a series of continuators, checkpoints, statistics objects and monitors are defined and associated together. For more information on these settings, see the EO documentation.

At the end of the function `main_function`, the actual evolutionary algorithm object is declared. This `eoEasyEA` instance is associated with the defined checkpoints, evaluation function, selection procedure, genetic operators, and replacement mechanism. The evolutionary cycle consists merely of associating the EA object with the current population object `pop`.

The function `main`, declared at the bottom of the file '`main.cpp`' consists of a protected call to the function `main_function` in a try/catch block to handle any execution exceptions.

7.2.9.3 'eoGE.h'

The file 'eoGE.h' defines the class eoGE, which represents a genome. The eoGE genome consists of a vector of unsigned char elements. This type was chosen because it is typically represented by 8 bits, which is the standard size of GE codons.

The methods printOn() and readFrom() output or read a genome using a standard input or output stream respectively. The method getEffectiveSize() merely returns the recorded effective size of the genotype of this individual.

7.2.9.4 'eoGEInit.h'

The file 'eoGEInit.h' defines the class eoGEInit which contains member functions for population initialisation. The constructor of the class reads, processes, and stores the related EO parameters to select a random or a sensible method for population initialisation.

In the case of a random initialisation, the method operator() calls the method randomInit(). This method creates individuals between the sizes minSize and maxSize containing random unsigned char numbers.

If sensible initialisation is selected, the mapper's init() method is called, and if successful, the contents of its Genotype structure are used to initialise the _genotype object passed as an argument.

7.2.9.5 'eoGEMutation.h'

The file 'eoGEMutation.h' declares the class eoGEMutation, which implements a point mutation operator, as used in standard GE. The mutation is applied to genotypes as arguments to the operator() method (the probability of mutation is supplied in an argument to the constructor).

7.2.9.6 'eoGEQuadCrossover.h'

The file 'eoGEQuadCrossover.h' declares the class eoGEQuadCrossover, which implements the crossover operator to use during the evolutionary cycle. Its constructor receives a boolean argument stating whether to use a standard 1-point crossover or an effective crossover method.

If standard 1-point crossover is specified, a random cut point is selected within each individual. If effective crossover is selected, each cut point must lie within the effective part of each individual (that is, the part of its genome which has been read to create a phenotype). After choosing the cut points, the second half of each individual is then swapped.

7.2.9.7 'eoGEEvalFunc-gcc_tcc.h'

The file 'eoGEEvalFunc-gcc_tcc.cpp' contains the functions directly related to the evaluation of each EO individual using a C compiler such as GCC. The following section describes the various files and functions that are part of the GCC backend for the cart centering example. This section also describes how GCC can be exchanged for another C compiler called TCC (from the TinyCC distribution).

The class eoGEEvalFunc is a template class which allows the individual type EOT to be specified when an instance of the class is created. The constructor is responsible for initialising the GE mapper and setting up the C evaluator backend. The mapper object

is initialised by specifying the maximum number of mapping wraps and loading the BNF grammar file using the `readBNFFile` member function. Each of these values are supplied as parameters to the constructor function. If the process of loading the grammar fails, an error message is printed to the standard error stream (`std::cerr`) and the program exits.

The next task in this function is to load the code that will be placed before and after the phenotype which is contained in the files '`cartcenterstart.c`' and '`cartcenterend.c`'. The function loads the contents of these files into the `CCstart` and `CCend` buffers (character arrays). These buffers are then used in the member `operator()` for every individual that is evaluated using the C compiler backend. For more information about these files, Section 7.2.4 [EXAMPLES/CartCentering/cartcenterstart.c], page 75, and Section 7.2.5 [EXAMPLES/CartCentering/cartcenterend.c], page 75.

The member `operator()` provides the definition of the operator required by EO for individual evaluation. It takes one argument of type `EOT` which represents an EO individual. The function first checks that this individual has not been evaluated previously by calling the member function `EOT::invalid`. If the individual is invalid, then it needs to be evaluated and assigned a fitness. If the individual is valid, the function exits.

In order to evaluate this individual, the mapper must be supplied with a valid libGE structure. This is constructed by generating an array of integers and copying each element from the EO individual. This array is then supplied to the constructor of the `Genotype` object. This is then assigned to the `mapper` object using the member function `setGenotype`. Once this is complete, a libGE `Phenotype` is extracted from the `mapper` object. If the phenotype is invalid, a fitness value of 0.0 is returned and the function exits. If the phenotype is valid, the full executable code needs to be constructed so that the phenotype can be evaluated in the C backend.

The first step in this process is to create a new empty file called '`individual.c`'. The contents of the buffer `CCstart` are written out to this file, followed by the contents of the phenotype, followed by the contents of the buffer `CCend`. The file '`individual.c`' is then closed. An example '`individual.c`' file is shown in Section 7.2.6 [EXAMPLES/CartCentering/WrappedCPhenotype], page 75.

At this point, the file contains a full valid C program which should be compilable by any C compiler. In this example, two different C compilers can be used: GCC and TinyCC. If the macro `GECART_USE_TCC_COMP` was defined when the file '`cartcenter-gcc.cpp`' was compiled, then the TinyCC C compiler (called `tcc`) is used. If the macro was not defined, the GNU C compiler (called `gcc`) is used. The compiler program is called by invoking a `system` call with the name of the compiler program and the name of the file containing the phenotype and wrapper code i.e. '`individual.c`'. The phenotype file is also linked with the object file produced by the compilation of the file `GEcart.c`. The `system` call also invokes the execution of the compiled program and redirects all standard output to a file called '`result`'. If the program compiles and the program runs successfully, the file '`result`' should contain the total time required to center the cart for the fitness cases defined for the problem.

To extract this time value, the result file is opened and the first floating point value found is assigned to the variable `fitness`. The result file is then closed.

This fitness is then inverted because the problem requires that the total time take to center and stop the cart be minimised. Inverting the value provides EO with the correct

fitness format i.e. EO is a maximising GA whereas the goal of the cart centering problem is to minimise the total time taken. Finally, the fitness is assigned to the EO individual using the member function `EOT::fitness`. The function then exits.

7.2.9.8 'eoGEEvalFunc-slang.h'

The file '`eoGEEvalFunc-slang.h`' declares the class `eoGEEvalFunc` required by EO for individual evaluation. This implementation uses the S-Lang library.

The class `eoGEEvalFunc` is a template class which allows the individual type `EOT` to be specified when an instance of the class is created. The constructor is responsible for initialising the GE mapper and setting up the GCC or TCC evaluator backend. The `mapper` object is initialised by specifying the maximum number of mapping wraps and loading the BNF grammar file using the `readBNFFile` member function. Each of these values are supplied as parameters to the constructor function. If the process of loading the grammar fails, an error message is printed to the standard error stream (`std::cerr`) and the program exits.

The function initialises the S-Lang environment by calling the S-Lang function `SLang_init_slang` and loads the stdio and math runtime libraries by calling `SLang_init_stdio` and `SLang_init_slmath`. If any of these functions fail, the program exits.

Once the S-Lang environment is set up, the file '`GEcart.sl`' (see Section 7.2.8.7 [EXAMPLES/CartCentering/GE_MITGALIB/GEcart.sl], page 83) is loaded using the function `SLang_load_file`. This file contains the functions and variables required to evaluate a genotype and is only required to be loaded once.

In order to extract information from the S-Lang runtime environment, a new C++ variable is added to the environment using the `SLadd_intrinsic_variable` function. This function requires the name of the variable within the S-Lang environment (called `Total_Time` in this example), the address of the variable in the C/C++ runtime (called `fitness` in this example), and the type of data stored in the variable (`SLANG_DOUBLE_TYPE`). If any of the S-Lang functions function fail, the program exits.

The member `operator()` provides the definition of the operator required by EO for individual evaluation. It takes one argument of type `EOT` which represents an EO individual. The function first checks that this individual has not been evaluated previously by calling the member function `EOT::invalid`. If the individual is invalid, then it needs to be evaluated and assigned a fitness. If the individual is valid, the function exits.

In order to evaluate this individual, the mapper must be supplied with a valid libGE structure. This is constructed by generating an array of integers and copying each element from the EO individual. This array is then supplied to the constructor of the `Genotype` object. This is then assigned to the `mapper` object using the member function `setGenotype`. Once this is complete, a libGE `Phenotype` is extracted from the `mapper` object. If the phenotype is invalid, a fitness value of 0.0 is returned and the function exits. If the phenotype is valid, the full executable code needs to be constructed so that the phenotype can be evaluated in the S-Lang backend.

Instead of having to construct the program and write it to a file to be compiled, S-Lang allows the interpretation and execution of the program in memory. This is achieved in this function by creating a string called `buffer` and appending the start, phenotype, and end code to it.

Once the complete program is constructed, the `buffer` string should contain a program that can execute within the S-Lang runtime. The contents of the string are executed in the runtime by calling the function `SLang_load_string`. Once execution has completed, the value in the variable `fitness` should contain the total time taken to center the cart for all the specified fitness cases.

This fitness is then inverted because the problem requires that the total time take to center and stop the cart be minimised. Inverting the value provides EO with the correct fitness format i.e. EO is a maximising GA whereas the goal of the cart centering problem is to minimise the total time taken. Finally, the fitness is assigned to the EO individual using the member function `EOT::fitness`. The function then exits.

7.2.9.9 'eoGEEvalFunc-libtcc.h'

The file '`eoGEEvalFunc-slang.h`' declares the class `eoGEEvalFunc` required by EO for individual evaluation. This implementation uses the libTCC library provided in the TinyCC distribution.

The class `eoGEEvalFunc` is a template class which allows the individual type `EOT` to be specified when an instance of the class is created. The constructor is responsible for initialising the GE mapper and setting up the libTCC evaluator backend. The `mapper` object is initialised by specifying the maximum number of mapping wraps and loading the BNF grammar file using the `readBNFFile` member function. Each of these values are supplied as parameters to the constructor function. If the process of loading the grammar fails, an error message is printed to the standard error stream (`std::cerr`) and the program exits.

In order to minimise the time taken to extract, evaluate and award a fitness to a GE phenotype, the start and end code that is placed before and after a GE phenotype are loaded into character buffers in this function. These buffers are accessed in the function `objfunc` because they are required for every GALib individual. In this example, the C code in the file '`cartcenterstart.c`' and '`cartcenterend.c`' are read into the `char` arrays `CCstart` and `CCend` respectively. The last task in this function is to load the file '`GEcart.c`' into another character buffer called `GEfunc`. This file contains the definitions of the functions and variables used either directly or indirectly by the GE phenotype code. For more information about these files, see Section 7.2.3 [EXAMPLES/CartCentering/GEcart.c], page 73 and Section 7.2.4 [EXAMPLES/CartCentering/cartcenterstart.c], page 75, and Section 7.2.5 [EXAMPLES/CartCentering/cartcenterend.c], page 75.

The member `operator()` provides the definition of the operator required by EO for individual evaluation. It takes one argument of type `EOT` which represents an EO individual. The function first checks that this individual has not been evaluated previously by calling the member function `EOT::invalid`. If the individual is invalid, then it needs to be evaluated and assigned a fitness. If the individual is valid, the function exits.

In order to evaluate this individual, the mapper must be supplied with a valid libGE structure. This is constructed by generating an array of integers and copying each element from the EO individual. This array is then supplied to the constructor of the `Genotype` object. This is then assigned to the `mapper` object using the member function `setGenotype`. Once this is complete, a libGE `Phenotype` is extracted from the `mapper` object. If the phenotype is invalid, a fitness value of 0.0 is returned and the function exits. If the phenotype

is valid, the full executable code needs to be constructed so that the phenotype can be evaluated in the libTCC backend.

The first task is to set up a new libTCC state which can compile and execute C code. This is achieved by calling the function `tcc_new`. The function `tcc_set_output_type` is then called to specify that compilation of the C code should take place in main memory to maximise performance. The math library `m` is also added to the libTCC state using the `tcc_add_library` function to allow the use of the square root function `sqrt` in the simulation process.

The next step is to make the various functions and variables that are defined in the code stored in the buffer `GEfunc` available in the libTCC state. This is achieved by calling the function `tcc_compile_string`, and supplying the current libTCC state and the buffer containing the code as arguments. It is important to note that the code in the `GEfunc` buffer is not executed at this stage. The purpose of this call is simply to compile the code and make it available to the phenotype code which will be compiled next.

Before the phenotype can be executed in the libTCC state, it must be wrapped with the start and end code stored in the `CCstart` and `CCend` buffers. The `char` buffer `buffer` is used to store the result of concatenating the buffer `CCstart`, the phenotype code, and the buffer `CCend`. An example of this concatenation is shown in Section 7.2.6 [EXAMPLES/CartCentering/WrappedCPhenotype], page 75. This buffer is then passed to the libTCC state using the function `tcc_compile_string`. If the compilation is successful, the libTCC state contains the executable form of the wrapped phenotype, the supporting functions, and the variables required to evaluate the GE phenotype.

In order to execute the code in memory, the `main` function (declared in '`cartcenterstart.c`') must be explicitly invoked. This function is accessed by first calling the function `tcc_relocate`. This makes all function and variable symbols accessible through the C interface and should be called after every call to `tcc_compile_string`. The function `tcc_get_symbol` is then called with a pointer to the libTCC state, a reference to a variable of type `unsigned long`, and the name of the symbol we require ("main"). If the required symbol is found, the `tcc_get_symbol` function stores its address in the `unsigned long` variable. This variable is then cast into a function pointer of type `int (*p_tccMainFunc)()`. The main function can the be invoked in the libTCC state by calling the function through the pointer i.e. `p_tccMainFunc()`.

The function `tcc_get_symbol` is then used again to retrieve the `sumFitCaseTimes` symbol within the libTCC state. The symbol corresponds to a variable of type `Real` which store the total time required to center the cart for all of the fitness cases. In order to access the value of the symbol, the unsigned `long variable val` must be cast to a pointer to a `Real`.. The type `Real` is defined at the top of the file using the `typedef` directive. In this example, the true type of `Real` is `double`. It is important that the true type of Real matches the type of the variable `sumFitCaseTimes` in the file `GEcart.c`. Once the cast is complete, the actual value of the variable `sumFitCaseTimes` is then retrieved from the libTCC state by dereferencing the local pointer variable. The function then deletes the libTCC state using the function `tcc_delete`.

This fitness is then inverted because the problem requires that the total time take to center and stop the cart be minimised. Inverting the value provides EO with the correct fitness format i.e. EO is a maximising GA whereas the goal of the cart centering problem

is to minimise the total time taken. Finally, the fitness is assigned to the EO individual using the member function `EOT::fitness`. The function then exits.

7.2.9.10 'eoGEEvalFunc-lua.h'

The file '`eoGEEvalFunc-lua.h`' declares the class `eoGEEvalFunc` required by EO for individual evaluation. The cart centering example uses the C API of the Lua library to interpret, execute, and evaluate the GE phenotype. Evaluation of the phenotype in this way results in significant improvements in execution time over language compilers e.g. GCC, and TCC. The following section describes the various files and functions that are part of the Lua backend for the cart centering example.

In order to access the Lua C API through the C++ code in '`eoGEEvalFunc-lua.h`', the `#include <header.h>` directives must be enclosed in a special directive called `extern "C"` e.g.

```
extern "C"
{
#include <lua.h>
#include <lauxlib.h>
#include <lualib.h>
}
```

This tells the compiler that the code in these files is C code and not C++ code. The functions in the file '`cartcenter-lua.cpp`' make use of the GE `GEGrammarSI` mapper, defined in the '`main.cpp`' file. A global pointer to a `lua_State`, called `L` in this example, is also created at the top of the file.for use by all the functions in the file.

This class is a template class which allows the individual type `EOT` to be specified when an instance of the class is created. The constructor is responsible for initialising the GE mapper and setting up the libTCC evaluator backend. The `mapper` object is initialised by specifying the maximum number of mapping wraps and loading the BNF grammar file using the `readBNFFile` member function. Each of these values are supplied as parameters to the constructor function. If the process of loading the grammar fails, an error message is printed to the standard error stream (`std::cerr`) and the program exits.

The Lua state is opened in this function by a call to the lua API function `lua_open`. Additional math and io libraries are added to this state using the function `luaL_openlibs`. Once completed, the Lua state is then ready to interpret and execute Lua code.

In order to minimise the time required for individual evaluations in the function `objfunc`, the Lua code in '`GEcart.lua`' is loaded into the state using the function `luaL_dofile`. This function loads and executes the Lua code in this file which is discussed in more detail in Section 7.2.8.13 [EXAMPLES/CartCentering/GE_MITGALIB/GEcart.lua], page 88. It is important to note that, because the file only contains variable declarations and function implementations, no code is executed at this time. If an error occurs while loading this file, Lua sets a error message at the top of its call stack. Each element on the Lua stack has an index e.g. an index of `-1` represents the element at the top of the stack. This index value is supplied to the`lua_tostring` to retrieve the error message. The message is then printed to standard error output and the program exits.

The member `operator()` provides the definition of the operator required by EO for individual evaluation. It takes one argument of type `EOT` which represents an EO individual.

The function first checks that this individual has not been evaluated previously by calling the member function `EOT::invalid`. If the individual is invalid, then it needs to be evaluated and assigned a fitness. If the individual is valid, the function exits.

In order to evaluate this individual, the mapper must be supplied with a valid libGE structure. This is constructed by generating an array of integers and copying each element from the EO individual. This array is then supplied to the constructor of the `Genotype` object. This is then assigned to the `mapper` object using the member function `setGenotype`. Once this is complete, a libGE `Phenotype` is extracted from the `mapper` object. If the phenotype is invalid, a fitness value of 0.0 is returned and the function exits. If the phenotype is valid, the full executable code needs to be constructed so that the phenotype can be evaluated in the Lua backend.

Instead of writing the phenotype code to a new file, the code is constructed in memory by appending the required start code, phenotype code, and end code to a `string` object (The start and end code are described in Section 7.2.8.14 [EXAMPLES/CartCentering/GE_MITGALIB/LuaStartCode], page 88 and Section 7.2.8.15 [EXAMPLES/CartCentering/GE_MITGALIB/LuaEndCode], page 89).

This code is executed in the Lua runtime by calling the function `luaL_dostring` and passing it a pointer to the Lua state and the buffer containing the code to be executed. If execution fails, the error message is retrieved from the Lua stack and printed to standard error output and the program then exits.

If the compilation and execution is successful, the function `lua_getglobal` is called to retrieve the Lua variable `sumFitCaseTimes`. This function pushes the variable on to the top of the stack. The value is checked and retrieved from the stack using the functions `lua_isnumber` and `lua_tonumber`. The value is then assigned to the `fitness` variable.

This fitness is then inverted because the problem requires that the total time take to center and stop the cart be minimised. Inverting the value provides EO with the correct fitness format i.e. EO is a maximising GA whereas the goal of the cart centering problem is to minimise the total time taken. Finally, the fitness is assigned to the EO individual using the member function `EOT::fitness`. The function then exits.

7.2.9.11 'Makefile'

The file '`Makefile`' contained in the example allows you to create a program using a choice of evaluators for your phenotype strings. Use `make` to create all executables, or `make GE*` to create a specific evaluator implementation (where `*` is any of `GCC`, `SLANG`, `TCC`, `LIBTCC`, or `LUA`). At the top of the file there are definitions for the location of the GALib, GE, and evaluator specific header and library files. If you have installed any of these in non-standard locations, make sure the values are adjusted to suit your system.

The file '`Makefile`' uses the `-include` compiler flag to use only the relevant include file for your choice of evaluator; if your compiler does not accept that flag, you should replace it with the appropriate flag.

Finally, some of the evaluators require the pre-compilation of the file '`GEcart.c`'; this is done by the '`Makefile`', when the relevant target implementations are chosen.

7.2.10 Cart Centering Performance

This section looks at the performance obtained with most of the search engines tested, for this specific problem. All search engines used a similar experimental setup: steady-state replacement, population size of 500 individuals, 100 generations, probability of crossover of 0.9, and probability of bit-wise mutation of 0.01. Furthermore, each experiment was performed with four different setups:

- Using a standard GE approach;
- Using effective crossover;
- Using sensible initialisation;
- Using a combination of the last two.

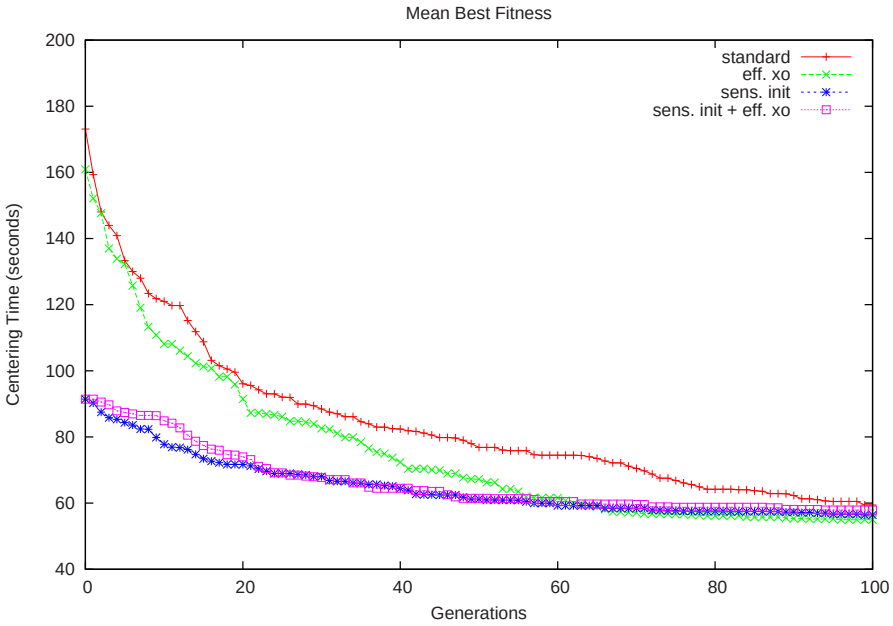
The results show that the standard GE configuration fairs slightly worse than the others and that the use of effective crossover alone produces the fittest individuals.The best centering time achieved over 30 runs using this configuration was approximately 54.9 seconds for 20 fitness cases.

The average execution times of these runs for each of the five backends and for both search engines are shown below.

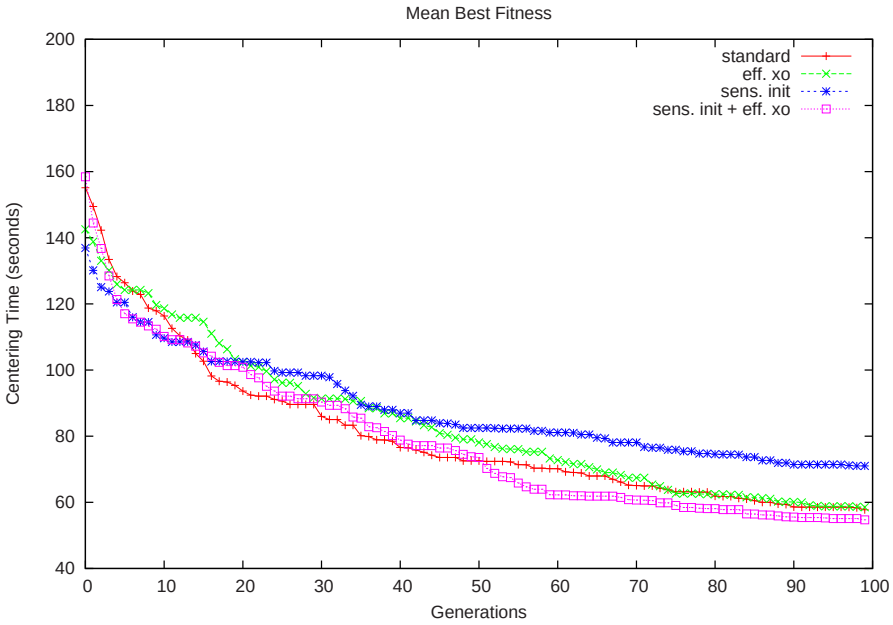
- GCC: 41m52.882s (GALib) 47m10.939s (EO)
- TCC: 7m14.129s (GALib) 6m33.551s (EO)
- LIBTCC: 5m6.340s (GALib) 5m47.404s (EO)
- SLANG: 8m27.626s (GALib) 10m9.699s (EO)
- LUA: 9m25.401s(GALib) 9m54.785s (EO)

For reference, these results were obtained on a PC with a Pentium IV processor running at 2.6GHz, 2048MB of memory, using Linux.

7.2.10.1 GALib performance



7.2.10.2 EO performance



7.3 Intertwined Spirals Problem

The intertwined spirals problem is a symbolic regression problem in which points must classified as belonging to one of two classes. The points of each class are arranged on the x-y plane so that they are intertwined and spiral out from the origin.

Lang and Whitbrock first attempted to solve this problem in 1989 using a neural network. The experiments used 97 coordinates for each class. In 1992, Koza attempted the problem using GP using three terminals x, y and ERC (a random constant). The function set included four arithmetic operators, a decision making function and the trigonometric sine and cosine functions. Koza increased the number of coordinates on each spiral to 194. Given that only two spirals needed to be classified, the output of the system was limited to boolean values by mapping all positive output to +1 and all negative output to -1.

7.3.1 Grammar

The grammar used in the GE implementation of the intertwined spirals problem is as follows.

```
<expr> ::= mymul(<expr>, <expr>) | mysub(<expr>, <expr>)
           | myadd(<expr>, <expr>) | pdiv(<expr>, <expr>)
           | if_cond(<expr>, <expr>, <expr>, <expr>)
           | (<expr>)
           | sin(<expr>) | cos(<expr>) | tan(<expr>) | exp(<expr>)
           | x | y | 1.000 | <const> | -<const>
<const> ::= 0.<digit><digit><digit>
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

7.3.2 'GEspiral.c'

The file '`GEspiral.c`' provides a number of variable and function declarations that are required to evaluate GALib individuals using the GCC, TCC, and libTCC backends. The header file '`GEspiral.h`' provides the declarations of the functions found in this file. This section documents the use of each of these functions and variables in the intertwined spirals problem. The following lists the variables declared in the file.

- `xvals` the array of x components of each of the 194 coordinates.
- `yvals` the array of y components of each of the 194 coordinates.
- `Class` the array of true classification values (i.e. +1 or -1 for each of the 194 coordinates.
- `Evolved` the array of classification values as determined by the GE individual.
- `FintessCases` the number of fitness cases (coordinates). In this example, this value is set to 194.

The function `initVals` populates the coordinate arrays `xvals` and `yvals` with values. The values are arraged in an intertwined spiral formation about the origin. The class of each coordinate is also stored in the corresponding index in the array `class`.

The function `AssignFitness` compares the classification values generated by the GE individual and stored in the array `Evolved` with the true classification values stored in the array `Class`.

The remaining functions in this file `if_cond`, `pdiv`, `myadd`, `mysub`, and `mymul` provide the implementation of the functions defined in the grammar file:

- `if_cond` a binary decision making function.

- `pdiv` a binary protected division operator.
- `myadd` a binary addition operator.
- `mysub` a binary subtraction operator.
- `mymul` a binary multiplication operator.

7.3.3 C Start Code 'spiralstart.c'

The file '`spiralstart.c`' references the variables and functions defined in '`GEspiral.c`' by including the header file '`GEspiral.h`' and using the `extern` keyword for the arrays `xvals`, `yvals`, and `Evolved`.

The file also starts the definition of a `main` function in which the phenotype execution is executed. The function starts with a call to `initVals` (defined in '`GEspiral.c`'. This function initialises the coordinates of the spirals and stores the x and y components in the arrays `xvals` and `yvals` respectively.

The function then construts a for loop to executing the classifier code for each of the coordinates in the spirals. The current x and y coordinates are retrieved from the current position in the arrays `xvals` and `yvals` and assigned to the working variables `x` and `y`. Note that these working variables are references by name in the grammar. The function then starts an assignment operator in which the result of the classification code is assigned to the variable temp. The code to complete the program after the phenotype code is placed at this point is described in the next section.

7.3.4 C End Code 'spiralend.c'

The code in the file '`spiralend.c`' completes the assignment operator from the last line in the file '`spiralstart.c`' and tests if the value assigned to temp is negative, in which case a value of 0 is stored in the current index of the array `Evolved`. or positive, in which case a 1 is stored. These values represent the classification of the x and y coordinates as belonging to either spiral 0 or spiral 1.

Finally, the result of the call to the function `AssignFitness` is printed to standard output. This value will be read by the objective function in the search engine-specific code described in the following sections.

7.3.5 Example wrapped C phenotype

```
#include <stdio.h>
#include <math.h>
#include "GEspiral.h"

extern double xvals[194];
extern double yvals[194];
extern int Evolved[194];
int main()
{
 initVals();
 int i;
 double temp,x,y;
 for(i=0;i<194;i++)
```

```
   {
     x=xvals[i];
     y=yvals[i];
     temp = x + y;
   if(temp < 0)
     Evolved[i]=0;
   else
     Evolved[i]=1;
 }
 printf("%d\n",AssignFitness());
}
```

7.3.6 GE_MITGALIB

This example uses the standard `GAParameterList` for passing arguments to specify the parameters of the run. The paramaters specific to GE can also be specified on the command line.

Parameters are passed to the executable in the following fashion:

```
./executable <param_name> <value>
```

The following list shows the parameters accepted by GALib with their default values:

- popsize: population size (default is 100);
- ngen: population size (default is 10);
- pcross: crossover probability (default is 0.9);
- pmut: mutation probability (default is 0.01);
- prepl: steady-state replacement strategy (default is 1.0);
- sfreq: how often to record scores (generation interval) (default is 1);
- ffreq: how often to dump scores to file (generation interval) (default is 1);
- sfile: name of score data file (default is "is-output.dat").
- setttings_file: the name of a file with GALib configuration parameters;

All other parameters are set to the GALib defaults (check the GALib documentation for more details).

The following parameters specific to GE can also be specified on the command line.

- seed: random seed to be used both by GALib and libGE (default is random);
- grammar: grammar file to use (default is "grammar.bnf");
- wrap: number of wrapping events (default is 0);
- sensible: use sensible initialisation instead of random (default is 0);
- min: minimum genotype size for random initialisation (default is 15);
- max: maximum genotype size for random initialisation (default is 25);
- grow: grow rate for sensible initialisation (default is 0.5);
- maxDepth: maximum tree depth for sensible initialisation (default is 10);
- tailSize: tail size for sensible initialisation (default is 0);
- tailRatio: tail ratio for sensible initialisation (default is 0.0);
- effective: use effective crossover instead of one point crossover (default is 0);

For example, to run the `GELUA` implementation, using the grammar file "grammar-lua.bnf", with a population size of 500 individuals, for 50 generations, using random seed 1234, and leaving all other parameters to standard values, the command to execute is:

```
./GELUA grammar grammar-lua.bnf popsize 500 ngen 50 seed 1234
```

7.3.6.1 ‘`main.cpp`’

This file drives the evolutionary process. It starts with the declaration of all functions required to set up and perform the GALib evolver, the definition of a `GEGrammarSI` mapper, the global integers containing minimum and maximum size information for random initialisation, and a counter for the number of objective function calls.

After displaying the version of libGE, the function creates the variables for the parameters specific to GE e.g. the location of the grammar file, the maximum number of wrapping events allowed, the use of sensible initialisation, etc. Additional variables are created to store the location of a GALib parameters file the random seed to use for GALib. Each of the arguments passed to the program on the command line are then read and values are assigned to the variables when matched.

A `GAParameterList` is then created to specify defaults for a number of GALib parameters including population size, number of generations, crossover and mutation probabilities, etc. If a settings file was specified on the command line, parameters are read from this file by calling the `GAParameterList::read` member function.

Once the parameters have been read, the genome structure is created. In this example, a class called `GEListGenome` is provied (in files '`GEListGenome.h`' and '`GEListGenome.cpp`') which derives from the GALib class `GAListGenome`. The fitness function, called `objfunc`, is also associated with the genome using the member function `GAListGenome::evaluator`.

The genetic operators to use are specified next. In this example, the default initialisation function is the random initialiser (`initFuncRandom()` but sensible initialisation is available in the function `initFuncSI()`. Both of these functions are declared in the file '`initfunc.h`' and defined in the file '`initfunc.cpp`', The initialiser function is associated with the genome using the `GAGenome::initialiser` member function.

Two crossover operators are available in this example. The default operator, implemented in the function `GAListGenome::OnePointCrossover`, is part of GALib and provides standard crossover functionality. The other operator, implemented in the function `GEListGenome::effCrossover` limits the crossover point to locations within the effective region of the genotype. This function is declared in the file '`GEListGenome.h`' and defined in '`GEListGenome.cpp`'. This operator and the mutation operator `GEListGenome::pointMutator` are declared in the '`GEListGenome.h`' and defined in '`GEListGenome.cpp`'. The crossover and mutation operators are associated with the GALib genome using the `GAGenome::crossover` and `GAGenome::mutation` member functions.

A call to the function `app_init()` follows, which loads the code to place before and after each evolved phenotype prior to evaluation, and initialises the libGE mapper; this function is contained in the files '`spiral-*.cpp`' (with '`*`' replaced with the name of the evaluator used).

Next comes the creation of the genetic algorithm `ga`, which uses a steady-state replacement strategy; the `GAParameterList` containing all parameters is associated with it. The mapper is also initialised, using either default values or the values obtained when parsing the command-line arguments. Finally, the recorded statistics are selected, a custom statistics file is created, and a call to the `initialize()` method is made, to apply all settings to the genetic algorithm, including the random seed.

An evolutionary cycle is then performed. This can either be done directly with `ga.evolve()`, or on a step by step cycle, as shown in the example. Also note how two statistics files are created, one using the GALib standard outputs, and a second one, illustrating the creation of a personalised statistics file.

Finally, once the evolution is finished, the statistical data is dumped onto the screen, as is the best individual of the population (through a call to the function `print_individual()`, defined in the files '`spiral-*.cpp`').

7.3.6.2 'initfunc.cpp'

This file contains the two specific initialisation functions designed for this example, `initFuncRandom()` and `initFuncSI()`. The first function randomly initialises each genome in the population. It starts by destroying any data possibly contained in an individual and then chooses a number between `minSize` and `maxSize` (declared and initialised in '`main.cpp`') for its size. It then initialises each gene to a number between 0 and 255, starting with the head of the list.

The second function (`initFuncSI()`) can be called to use libGE's sensible initialisation routines to initialise each genome in the population. It starts by destroying any data possibly contained in an individual and then uses the `init()` method from the mapper to apply the sensible initialisation routines. If successful, each codon from the mapper's `Genotype` structure is then copied into the genome.

7.3.6.3 'GEListGenome.h'

This file declares the `GEListGenome` class, which is the genome used in the intertwined spirals example. It derives from GALib's `GAListGenome` class, using the type `unsigned char` as its elements, as an `unsigned char` is usually composed of eight bits, which is the standard number of bits per gene used in Grammatical Evolution.

This class extends the `GAListGenome` by allowing the genome to record its effective length, that is, the portion of the genome that was actually used during the mapping process. It also provides two methods, `effCrossover()` and `pointMutator()`, which implement a crossover restricted to the effective lengths of the genomes and GE's standard bit-level mutation, respectively. The class also provides a number of functions required for correct operation with GALib including a copy constructor and the `clone`, `copy`, and `equal` functions. The implementation of all these functions is in the file '`GEListGenome.cpp`'.

7.3.6.4 'spiral-gcc.cpp'

The file '`spiral-gcc.cpp`' contains the functions directly related to the evaluation of each GALib individual using a C compiler such as GCC. The following section describes the various files and functions that are part of the C backend for the intertwined spirals example. This section also describes how the default compiler, GCC, can be exchanged for another C compiler called TCC (from the TinyCC distribution).

The `app_init` function, called from the function `main`, is responsible for initialising the GE mapper and setting up the C evaluation backend.

The `GEGrammarSI mapper` object, which is declared in the '`main.cpp`', is initialised by specifying the maximum number of mapping wraps and loading the BNF grammar using the member function `readBNFFile`. Both the number of wrapping events and the path to the grammar file are specified in parameters to `app_init`. If the grammar fails to load, an error message is printed to the standard error stream (`std::cerr`) and the program exits.

The next task in this function is to load the code that will be placed before and after the phenotype which is contained in the files '`spiralstart.c`' and '`spiralend.c`'. The function

loads the contents of these files into the `SPstart` and `SPend` buffers (character arrays). These buffers are then used in the `objfunc` function for every individual that is evaluated using the C compiler backend. For more information about these files, Section 7.3.3 [spiralstart.c], page 102, and Section 7.3.4 [spiralend.c], page 102.

The function `objfunc` provides the definition of the evaluation operator required by GALib. This function takes one argument of type `GAGenome` and returns a floating point value representing the fitness awarded to that `GAGenome` object. In order to evaluate the genome, the GAGenome object has to be cast into its derived type (`GEListGenome`). This allows the extra member functions specified in the class `GEListGenome` to be used in this function.

Once cast, the `GEListGenome` is then assigned to the `mapper` object. A libGE `Phenotype` is then extracted from the same `mapper` object. If the phenotype is invalid, a fitness value of 0.0 is returned and the function exits. If the phenotype is valid, the full executable code needs to be constructed so that the phenotype can be evaluated in the C backend.

The first step in this process is to create a new empty file called '`individual.c`'. The contents of the buffer `SPstart` are written out to this file, followed by the contents of the phenotype, followed by the contents of the buffer `SPend`. The file '`individual.c`' is then closed. An example '`individual.c`' file is shown in Section 7.3.5 [WrappedCPhenotype], page 102.

At this point, the file contains a full valid C program which should be compilable by any C compiler. In this example, two different C compilers can be used: GCC and TinyCC. If the macro `GESPIRAL_USE_TCC_COMP` was defined when the file '`spiral-gcc.cpp`' was compiled, then the TinyCC C compiler (called `tcc`) is used. If the macro was not defined, the GNU C compiler (called `gcc`) is used. The compiler program is called by invoking a `system` call with the name of the compiler program and the name of the file containing the phenotype and wrapper code i.e. '`individual.c`'. The phenotype file is also linked with the object file produced by the compilation of the file `GEspiral.c`. The `system` call also invokes the execution of the compiled program and redirects all standard output to a file called '`result`'. If the program compiles and the program runs successfully, the file '`result`' should contain the number of fitness cases which were classified correctly.

To extact this value, the result file is opened and the first integer value found is assigned to the variable `fitness`. The result file is then closed.

A number of tests are done on the fitness variable to ensure that it is a number. Before the function returns this value, it sets the effective size of the genotype in the genome object. This value is used by the effective crossover operator.

Before the function returns this value, it sets the effective size of the genotype in the genome object. This value is used by the effective crossover operator.

Finally, the fitness is returned and the function exits.

Finally, the `print_individual` function receives a genome as argument, performs its mapping to a phenotype structure, and prints it onto the screen, along with measurements of the genotype, generated phenotype, and the mapping process itself.

7.3.6.5 'spiral-slang.cpp'

This file contains the functions related to the evaluation of each individual, made specifically for use with the S-Lang runtime. The following section describes the various files and functions that are part of the S-Lang backend for the intertwined spirals example.

The `app_init` function, called from the `main` function, is responsible for initialising the libGE mapper and initialising the S-Lang interpreter.

The `GEGrammarSI` mapper object used in this function is declared in the file '`main.cpp`'. The number of wrapping events and the path to the grammar are supplied in parameters to this function. The `mapper` object is initialised by specifying the maximum number of mapping wraps and loading the BNF grammar file using the `readBNFFile` member function. If the process of loading the grammar fails, an error message is printed to the standard error stream (`std::cerr`) and the program exits.

The function initialises the S-Lang environment by calling the S-Lang function `SLang_init_slang` and loads the stdio and math runtime libraries by calling `SLang_init_stdio` and `SLang_init_slmath`. If any of these functions fail, the program exits.

Once the S-Lang environment is set up, the file '`GEspiral.sl`' is loaded using the function `SLang_load_file`. This file contains the functions and variables required to evaluate a GE phenotype and is only required to be loaded once.

The function `objfunc` provides the definition of the evaluation operator required by GALib. This function takes one argument of type `GAGenome` and returns a floating point value representing the fitness awarded to that `GAGenome` object. In order to evaluate the genome, the GAGenome object has to be cast into its derived type (`GEListGenome`). This allows the extra member functions specified in the class `GEListGenome` to be used in this function.

Once cast, the `GEListGenome` is then assigned to the `mapper` object. Once this is complete, a libGE `Phenotype` is then extracted from the same `mapper` object. If the phenotype is invalid, a fitness value of 0.0 is returned and the function exits. If the phenotype is valid, the full executable code needs to be constructed so that the phenotype can be evaluated in the S-Lang backend.

Instead of having to construct the program and write it to a file to be compiled, S-Lang allows the interpretation and execution of the program in memory. This is achieved in this function by creating a string called `buffer` and appending the start, phenotype, and end code to it. Once the complete program is constructed, the `buffer` string should contain a program that can execute within the S-Lang runtime.

In order to extract information from the S-Lang runtime environment, a new C++ variable is added to the environment using the `SLadd_intrinsic_variable` function. This function requires the name of the variable within the S-Lang environment (called `Fitness_Variable` in this example), the address of the variable in the C/C++ runtime (called `fitness` in this example), and the type of data stored in the variable (`SLANG_INT_TYPE`). If any of the S-Lang functions function fail, the program exits

The contents of the string are executed in the runtime by calling the function `SLang_load_string`. Once execution has completed, the value in the variable `fitness` should contain the number of fitness cases which were correctly classified.

Before the function returns this value, it sets the effective size of the genotype in the genome object. This value is used by the effective crossover operator.

Finally, the fitness is returned and the function exits.

Finally, the `print_individual` function receives a genome as argument, performs its mapping to a phenotype structure, and prints it onto the screen, along with measurements of the genotype, generated phenotype, and the mapping process itself.

7.3.6.6 'GEspiral.sl'

The file '`GEspiral.sl`' provides the variables and functions required to evaluate the GE phenotype for the intertwined spirals problem using the S-Lang runtime. It provides the functions `initVals` and `AssignFitness` and the grammar functions `if_cond`, `pdiv`, `myadd`, `mysub`, and `mymul`. The functionality of the code is the same as that found in the file '`GEspiral.c`' and descriptions of the variables and functions can be found in section Section 7.3.2 [GEspiral.c], page 101.

7.3.6.7 S-Lang start code

The code required to invoke the functions in '`GEspiral.sl`' is shown below. This code is executed just before the GE phenotype in the `objfunc` function described in section Section 7.3.6.5 [EXAMPLES/IntertwinedSpirals/GE_MITGALIB/spiral-slang.cpp], page 107.

```
initVals();
variable i;
variable temp = Double_Type;
variable fit = Double_Type;
fit=0;
for(i=0;i<194;i++)
{
   variable x = Double_Type;
   variable y = Double_Type;
   y=yvals[i];
   x=xvals[i];
   Evolved[i]=
```

The code first initialises the variables declared in '`GEspiral.sl`'. Once a number of working variables are created, the for loop retrieves the x and y coordinate of the current test case from the coordinate arrays `xvals` and `yvals`. The code segment ends with an assignment operation to the current index in the array `Evolved`. This statement is left unfinished to allow the insertion of the GE phenotype code at this point. The remainder of the code is explained in the next section .

7.3.6.8 S-Lang end code

In the previous section, the code segment shows that an assignment operation is started. The code required to complete this assignment and to evaluate the fitness of the classification accuracy of the GE phenotype is shown below.

```
  ;
}
Fitness_Variable=AssignFitness();
```

The code completes the assignment statement and closes the for loop from the start code. Finally, the fitness assigned to the variable `Fitness_Variable`, which is a C variable passed into the S-Lang environment using the `SLadd_intrinsic_variable` function.

7.3.6.9 Example wrapped S-Lang phenotype

A GE phenotype, complete with start and end code is shown in the following code segment.

```
initVals();
variable i;
variable temp = Double_Type;
variable fit = Double_Type;
fit=0;
for(i=0;i<194;i++)
{
   variable x = Double_Type;
   variable y = Double_Type;
   y=yvals[i];
   x=xvals[i];
   Evolved[i]= x + y;
}
Fitness_Variable=AssignFitness();
```

7.3.6.10 'spiral-libtcc.cpp'

The file '`spiral-libtcc.cpp`' contains the functions directly related to the evaluation of each GALib individual using the libTCC library (part of the TinyCC distribution). The intertwined spirals example uses the C API of the libTCC library to interpret, execute, and evaluate the GE phenotype. Evaluation of the phenotype in this way results in significant improvements in execution time over language compilers e.g. GCC, and TCC. The following section describes the various files and functions that are part of the libTCC backend for the intertwined spirals example.

The function `app_init` function, called from the function `main` in the file '`main.cpp`' is responsible for initialising the libGE mapper, loading the BNF grammar file, and setting up the resources required to evaluate the GE phenotypes using the libTCC evaluator in the function `objfunc`.

The `GEGrammarSI mapper` object, which is declared in the '`main.cpp`', is initialised by specifying the maximum number of mapping wraps `wrappingEvents` and the BNF `grammarFile` is loaded using the `readBNFFile` member function. Both of these variables are supplied as parameters to the`app_init` function. If the process of loading the grammar fails, an error message is printed to the standard error stream (`std::cerr`) and the program exits.

In order to minimise the time taken to extract, evaluate and award a fitness to a GE phenotype, the start and end code that is placed before and after a GE phenotype are loaded into character buffers in this function. These buffers are accessed in the function `objfunc`, every time a GALib individual needs is evaluated. In this example, the C code in the file '`spiralstart.c`' and '`spiralend.c`' are read into the `char` arrays `SPstart` and `SPend` respectively. The last task in this function is to load the file '`GEspiral.c`' into another character buffer called `GEfunc`. This file contains the definitions of the functions and variables that will be used either directly or indirectly by the GE phenotype code. For more information about these files, see Section 7.3.2 [GEspiral.c], page 101 and Section 7.3.3 [spiralstart.c], page 102, and Section 7.3.4 [spiralend.c], page 102.

The `objfunc` function is responsible for evaluating an individual (passed in the argument `g`). The `setGenotype()` method from the mapper is used to create a `Genotype` structure within the mapper, from the genome `g`. If the resulting phenotype is not valid, a fitness of 0.0 is returned and the program exits. If the phenotype is valid, then the evaluation of the phenotype using the libTCC library proceeds.

The first task is to set up a new libTCC state which can compile and execute C code. This is achieved by calling the function `tcc_new`. The function `tcc_set_output_type` is then called to specify that compilation of the C code should take place in main memory to maximise performance. The math library `m` is also added to the libTCC state using the `tcc_add_library` function to allow the use of a number of math functions at the evaluation stage.

The next step is to make the various functions and variables that are defined in the code stored in the buffer `GEfunc` available in the libTCC state. This is achieved by calling the function `tcc_compile_string` and passing the current libTCC state and the buffer containing the code. It is important to note that the code in the buffer `GEfunc` is not executed at this stage. The purpose of this call is simply to compile the code and make it available to the phenotype code which we will execute next.

Before the phenotype can be executed in the libTCC state, it must be wrapped with the start and end code stored in the buffers `SPstart` and `SPend`. The `char` buffer `buffer` is used to store the result of concatenating the buffer `SPstart`, the phenotype code, and the buffer `SPend`. An example of this concatenation is shown in Section 7.3.5 [WrappedCPhenotype], page 102. This buffer is then passed to the libTCC state using the `tcc_compile_string` function. If the compilation is successful, the libTCC state contains the executable form of the wrapped phenotype, the supporting functions, and the variables required to evaluate the GE phenotype.

In order to execute the code in memory, the `main` function (declared in the file '`spiralstart.c`') must be explicitly invoked. This function is accessed by first calling the function `tcc_relocate`. This makes all function and variable symbols accessible through the C interface and should be called after every call to `tcc_compile_string`. The function `tcc_get_symbol` is then called with a pointer to the libTCC state, a reference to a variable of type `unsigned long`, and the name of the symbol we require ("main"). If the required symbol is found, the `tcc_get_symbol` function stores its address in the `unsigned long` variable. This variable is then cast into a function pointer of type `int (*p_tccMainFunc)()`. The main function can the be invoked in the libTCC state by calling the function through the pointer i.e. `p_tccMainFunc()`.

One the main function has completed, the function `objfunc` must invoke the function `AssignFitness` to retrieve the fitness value of the GE individual. This is performed using the same method as the main function invocation shown above. A function pointer which matches the signature of the function `AssignFitness` must first be created. In this example, the pointer is called `p_tccFitnessFunc"`. The libTCC function `tcc_get_symbol` is then called to store the symbol location of this function in the temporary variable `val`. This variable is then cast and stored in the function pointer. The return value of the `AssignFitness` function can then be retrieved by invoking the function through the pointer in the normal fashion i.e. `fitness = p_tccFitnessFunc();`. The function then deletes the libTCC state using the function `tcc_delete`.

The function then checks that the fitness value is a valid number. If this test fails, the fitness value is set to 0.

Before the function returns this value, it sets the effective size of the genotype in the genome object. This value is used by the effective crossover operator.

Finally, the fitness is returned and the function exits.

Finally, the `print_individual` function receives a genome as argument, performs its mapping to a phenotype structure, and prints it onto the screen, along with measurements of the genotype, generated phenotype, and the mapping process itself.

7.3.6.11 'spiral-lua.cpp'

The file '`spiral-lua.cpp`' contains the functions directly related to the evaluation of each GALib individual using the Lua evaluator. The intertwined spiral uses the C API of the Lua library to interpret, execute, and evaluate the GE phenotype. Evaluation of the phenotype in this way results in significant improvements in execution time over language compilers e.g. GCC, and TCC. The following section describes the various files and functions that are part of the Lua backend for the intertwined spirals example.

In order to access the Lua C API through the C++ code in '`spiral-lua.cpp`', the `#include <header.h>` directives must be enclosed in a special directive called `extern "C"` e.g.

```
extern "C"
{
#include <lua.h>
#include <lauxlib.h>
#include <lualib.h>
}
```

This tells the compiler that the code in these files is C code and not C++ code. The functions in the file '`spiral-lua.cpp`' make use of the GE `GEGrammarSI` mapper, defined in the '`main.cpp`' file. A global pointer to a `lua_State`, called `L` in this example, is also created at the top of the file.for use by all the functions in the file.

The `app_init` function, called from the `main` function, is responsible for initialising the libGE mapper and initialising the Lua interpreter.

The `GEGrammarSI` mapper object used in this function is declared in the file '`main.cpp`'. The number of wrapping events and the path to the grammar are supplied in parameters to this function. The `mapper` object is initialised by specifying the maximum number of mapping wraps and loading the BNF grammar file using the `readBNFFile` member function. If the process of loading the grammar fails, an error message is printed to the standard error stream (`std::cerr`) and the program exits.

The Lua state is opened in this function by a call to the lua API function `lua_open`. Additional math and io libraries are added to this state using the function `luaL_openlibs`. Once completed, the Lua state is then ready to interpret and execute Lua code.

In order to minimise the time required for individual evaluations in the function `objfunc`, the Lua code in '`GEspiral.lua`' is loaded into the state using the function `luaL_dofile`. This function loads and executes the Lua code in this file which is discussed in more detail in Section 7.3.6.12 [EXAMPLES/IntertwinedSpirals/GE_MITGALIB/GEspiral.lua],

page 113. It is important to note that, because the file only contains variable declarations and function implementations, no code is executed at this time. If an error occurs while loading this file, Lua sets a error message at the top of its call stack. Each element on the Lua stack has an index e.g. an index of `-1` represents the element at the top of the stack. This index value is supplied to the`lua_tostring` to retrieve the error message. The message is then printed to standard error output and the program exits.

The function `objfunc` provides the definition of the evaluation operator required by GALib. This function takes one argument of type `GAGenome` and returns a floating point value representing the fitness awarded to that `GAGenome` object. In order to evaluate the genome, the GAGenome object has to be cast into its derived type (`GEListGenome`). This allows the extra member functions specified in the class `GEListGenome` to be used in this function.

Once cast, the `GEListGenome` is then assigned to the `mapper` object. A libGE `Phenotype` is then extracted from the same `mapper` object. If the phenotype is invalid, a fitness value of 0.0 is returned and the function exits. If the phenotype is valid, the full executable code needs to be constructed so that the phenotype can be evaluated in the Lua backend.

Instead of writing the phenotype code to a new file, the code is constructed in memory by appending the required start code, phenotype code, and end code to a `string` object (The start and end code are described in Section 7.3.6.13 [EXAMPLES/IntertwinedSpirals/GE_MITGALIB/LuaStartCode], page 114 and Section 7.3.6.14 [EXAMPLES/IntertwinedSpirals/GE_MITGALIB/LuaEndCode], page 114).

This code is executed in the Lua runtime by calling the function `luaL_dostring` and passing it a pointer to the Lua state and the buffer containing the code to be executed. If execution fails, the error message is retrieved from the Lua stack and printed to standard error output and the program exits.

If the compilation and execution is successful, the function `lua_getglobal` is called to retrieve the Lua variable `sumFitCaseTimes`. This function pushes the variable on to the top of the stack. The value is checked and retrieved from the stack using the functions `lua_isnumber` and `lua_tonumber`. The value is then assigned to the `fitness` variable.

Before returning the fitness value, the effective size of the GE genotype is stored in the mapper object using the `setEffectiveSize` member function of the `GEListGenome` class.

Finally, the fitness is returned and the function exits.

Finally, the `print_individual` function receives a genome as argument, performs its mapping to a phenotype structure, and prints it onto the screen, along with measurements of the genotype, generated phenotype, and the mapping process itself.

7.3.6.12 'GEspiral.lua'

The file '`GEspiral.lua`' defines the variables and functions required to evaluate the GE phenotype for the intertwined spirals problem using the Lua runtime. It provides the functions `initVals` and `AssignFitness` and the grammar functions `if_cond`, `pdiv`, `myadd`, `mysub`, `mymul` and `myadd`. The functionality of the code is the same as that found in the file '`GEspiral.c`' and descriptions of the variables and functions can be found in Section 7.3.2 [GEspiral.c], page 101.

7.3.6.13 Lua start code

In order to execute the phenotype in the Lua state, code must be prepended and appended to the phenotype. The following code segment shows the code that is prepended to the phenotype. It starts with a call to `initVals` defined in '`GEspiral.lua`' and then sets up some temporary variables. It also sets up a for loop to iterate over each test case. In the body of this loop, the current test case coordinates are retrieved and an assignment statement is started. This statement is left open to allow the addition of the GE phenotype code. The code that is appended to the phenotype code in order to construct a valid Lua program is described in the next section.

```
initVals();
temp = 0.0;
fit = 0.0;
for i=0,193,1 do ";
    x=xvals[i];
    y=yvals[i];
    temp=
```

7.3.6.14 Lua end code

The following code is appended to the phenotype to complete the assignment statement started in the start code and to evaluate the class which should be assigned to the current test case. If the value of the phenotype (stored in `temp` is less than 0, then the class of the test case is 0. If the phenotype evaluated to 0 or greater, then the class of the test case is 1. The class is stored in the array `Evolved` (declared in '`GEspiral.lua`').

```
   ;

   if temp < 0 then
        Evolved[i]=0;
   else
        Evolved[i]=1;
   end
end
Fitness_Variable = AssignFitness();
```

7.3.6.15 Example wrapped Lua phenotype

A phenotype complete with start and end code is shown below as an example.

```
initVals();
temp = 0.0;
fit = 0.0;
for i=0,193,1 do ";
    x=xvals[i];
    y=yvals[i];
    temp=x + y;

   if temp < 0 then
        Evolved[i]=0;
```

```
   else
        Evolved[i]=1;
   end
end
Fitness_Variable = AssignFitness();
```

7.3.6.16 'Makefile'

The '`Makefile`' contained in the example allows you to create a program using a choice of evaluators for your phenotype strings. Use `make` to create all executables, or `make GE*` to create a specific evaluator implementation (where `*` is any of `GCC`, `SLANG`, `TCC`, `LIBTCC`, or `LUA`). At the top of the file there are definitions for the location of the GALib, GE, and evaluator specific header and library files. If you have installed any of these in non-standard locations, make sure the values are adjusted to suit your system.

Note: Some of the evaluators require the pre-compilation of the '`GEspiral.c`'. This is automatically done by the '`Makefile`' when the relevant evaluators are chosen.

7.3.7 GE_EO

The example using the Evolutionary Objects (EO) library can be found in '`EXAMPLES/IntertwinedSpirals/GE_EO`'. The following sections contain a user's guide and a description of the files in this example.

7.3.7.1 EO Example User Guide

The EO library has a powerful command-line parser, which is fully-configurable. This example uses that parser to specify both the standard GA parameters, and also the GE specific parameters.

Parameters are passed to the executable in the following fashion:

```
./executable -param_short_name[=value]
```

or

```
./executable --param_long_name[=value]
```

The following are the parameters accepted by this example, and their default values:

- -h, –help: prints help message (default is 0);
- –stopOnUnknownParam: stop if unknown parameter entered (default is 1);
- -C, –pCross: probability of crossover (default is 0.9);
- -E, –effCross: use effective crossover (default is 0);
- -M, –pMut: probability of mutation (default is 1);
- -b, –pMutPerBit: probability of bit-flip mutation (default is 0.01);
- -s, –sensibleInit: use sensible initialisation (default is 0);
- -x, –minSize: minimum size for random initialisation (default is 15);
- -X, –maxSize: maximum size for random initialisation (default is 25);
- -r, –grow: grow ratio for SI (default is 0.5);
- -d, –maxDepth: maximum depth for SI (default is 10);
- -T, –tailSize: tail size for SI (default is 0);
- -t, –tailRatio: tail ratio for SI (default is 0);
- -g, –grammarFile: grammar file (default is "grammar.bnf");
- -w, –wrappingEvents: number of wrapping events (default is 0);
- -L, –Load: save file to restart run (no default);
- -S, –Status: status file (default is "./<name_of_executable>.status");
- -R, –seed: random number seed (default is random);
- -P, –popSize: population size (default is 100);
- -G, –maxGen: maximum number of generations (default is 10).

For example, to run the `GELUA` implementation, using the grammar file "grammar.bnf", with a population size of 500 individuals, for 50 generations, using random seed 1, and leaving all other parameters to standard values, the command to execute is:

```
GELUA -g=grammar.bnf -P=500 -G=50 -R=1
```

7.3.7.2 'GEEA.cpp'

The file 'GEEA.cpp' contains the main() function, and is the central point of control over the evolutionary process. It begins by including all required EO include files, followed by a declaration of the fitness type to be used, and the type of each individual. The typedef directive declares the type Indi as a substitute name for the individual type eoGE<MyFitT>. The class eoGE is declared in the file 'eoGE.h' and is discussed in the next section. This example uses a custom genotype structure because EO does not have direct support for variable-length integer-based structures. The file 'GEEA.cpp' also declares the libGE GEGrammarSI mapper.

The function main_function() starts by defining all parameters available to the system using an instance of the eoParser class. The name of the parameter, the associated help messages, and the default value are supplied to the eoParser::createParam member function.

The fitness operator is then declared by creating an instance of the template class eoGEEvalFunc. The template in this case concerns the type of individual to be evaluated (called Indi in this example). The operator object is supplied with the libGE mapper object and the path of the BNF grammar file to use.

A number of fitness operator classes are provided in the intertwined spirals example for EO which use a differend backend platform to evaluate the GE phenotype. Each of the evaluation backends are implemented in files named 'eoGEEvalFunc-*.cpp' with '*' replaced by the name of the backend e.g. 'eoGEEvalFunc-slang.cpp'. All the fitness operator classes are called eoGEEvalFunc and each implements the interface required by EO i.e. they implement an operator function () which takes an eo individual, evaluates it, and then assigns a fitness value. The particular backend fitness operator class created at execution time is determined during compilation. It is therefore sufficient at this point to create an instance of the class using the name eoGEEvalFunc. The compilation process is described in more detail later in this section.

The function main_function then declares an objective function call counter and a eoPOP object to store the population of Indi individuals.

The random seed generator and the Indi population are then registered with a new instance of the class eoState. This class is responsible for initialising and evaluating the population in the case of a new run, or loading the population from disk if continuing a previous run.

The definition of roulette-wheel selection and steady-state replacement (GE style) follow, along with the definition of the genetic operators to use (which are encapsulated into a eoTransform object). Finally, a series of continuators, checkpoints, statistics objects and monitors are defined and associated together. For more information on these settings, see the EO documentation.

At the end of the function main_function, the actual evolutionary algorithm object is declared. This eoEasyEA instance is associated with the defined checkpoints, evaluation function, selection procedure, genetic operators, and replacement mechanism. The evolutionary cycle consists merely of associating the EA object with the current population object pop.

The function main, declared at the bottom of the file 'GEEA.cpp' consists of a protected call to the function main_function in a try/catch block to handle any execution exceptions.

7.3.7.3 'eoGE.h'

The file 'eoGE.h' defines the class eoGE, which represents a genome. The eoGE genome consists of a vector of unsigned char elements. This type was chosen because it is typically represented by 8 bits, which is the standard size of GE codons.

The methods printOn() and readFrom() output or read a genome using a standard input or output stream respectively. The method getEffectiveSize() merely returns the recorded effective size of the genotype of this individual.

7.3.7.4 'eoGEInit.h'

The file 'eoGEInit.h' defines the class eoGEInit which contains member functions for population initialisation. The constructor of the class reads, processes, and stores the related EO parameters to select a random or a sensible method for population initialisation.

In the case of a random initialisation, the method operator() calls the method randomInit(). This method creates individuals between the sizes minSize and maxSize containing random unsigned char numbers.

If sensible initialisation is selected, the mapper's init() method is called, and if successful, the contents of its Genotype structure are used to initialise the _genotype object passed as an argument.

7.3.7.5 'eoGEMutation.h'

The file 'eoGEMutation.h' declares the class eoGEMutation, which implements a point mutation operator, as used in standard GE. The mutation is applied to genotypes as arguments to the operator() method (the probability of mutation is supplied in an argument to the constructor).

7.3.7.6 'eoGEQuadCrossover.h'

The file 'eoGEQuadCrossover.h' declares the class eoGEQuadCrossover, which implements the crossover operator to use during the evolutionary cycle. Its constructor receives a boolean argument stating whether to use a standard 1-point crossover or an effective crossover method.

If standard 1-point crossover is specified, a random cut point is selected within each individual. If effective crossover is selected, each cut point must lie within the effective part of each individual (that is, the part of its genome which has been read to create a phenotype). After choosing the cut points, the second half of each individual is then swapped.

7.3.7.7 'eoGEEvalFunc-gcc_tcc.h'

The file 'eoGEEvalFunc-gcc_tcc.cpp' contains the functions directly related to the evaluation of each EO individual using a C compiler such as GCC. The following section describes the various files and functions that are part of the GCC backend for the intertwined spirals example. This section also describes how GCC can be exchanged for another C compiler called TCC (from the TinyCC distribution).

The class eoGEEvalFunc is a template class which allows the individual type EOT to be specified when an instance of the class is created. The constructor is responsible for initialising the GE mapper and setting up the C evaluator backend. The mapper object

is initialised by specifying the maximum number of mapping wraps and loading the BNF grammar file using the `readBNFFile` member function. Each of these values are supplied as parameters to the constructor function. If the process of loading the grammar fails, an error message is printed to the standard error stream (`std::cerr`) and the program exits.

The next task in this function is to load the code that will be placed before and after the phenotype which is contained in the files '`spiralstart.c`' and '`spiralend.c`'. The function loads the contents of these files into the `SPstart` and `SPend` buffers (character arrays). These buffers are then used in the `objfunc` function for every individual that is evaluated using the C compiler backend. For more information about these files, Section 7.3.3 [spiralstart.c], page 102, and Section 7.3.4 [spiralend.c], page 102.

The member `operator()` provides the definition of the operator required by EO for individual evaluation. It takes one argument of type `EOT` which represents an EO individual. The function first checks that this individual has not been evaluated previously by calling the member function `EOT::invalid`. If the individual is invalid, then it needs to be evaluated and assigned a fitness. If the individual is valid, the function exits.

In order to evaluate this individual, the mapper must be supplied with a valid libGE structure. This is constructed by generating an array of integers and copying each element from the EO individual. This array is then supplied to the constructor of the `Genotype` object. This is then assigned to the `mapper` object using the member function `setGenotype`. Once this is complete, a libGE `Phenotype` is extracted from the `mapper` object. If the phenotype is invalid, a fitness value of 0.0 is returned and the function exits. If the phenotype is valid, the full executable code needs to be constructed so that the phenotype can be evaluated in the C backend.

The first step in this process is to create a new empty file called '`individual.c`'. The contents of the buffer `SPstart` are written out to this file, followed by the contents of the phenotype, followed by the contents of the buffer `SPend`. The file '`individual.c`' is then closed. An example '`individual.c`' file is shown in Section 7.3.5 [WrappedCPhenotype], page 102.

At this point, the file contains a full valid C program which should be compilable by any C compiler. In this example, two different C compilers can be used: GCC and TinyCC. If the macro `GESPIRAL_USE_TCC_COMP` was defined when the file '`spiral-gcc.cpp`' was compiled, then the TinyCC C compiler (called `tcc`) is used. If the macro was not defined, the GNU C compiler (called `gcc`) is used. The compiler program is called by invoking a `system` call with the name of the compiler program and the name of the file containing the phenotype and wrapper code i.e. '`individual.c`'. The phenotype file is also linked with the object file produced by the compilation of the file `GEspiral.c`. The `system` call also invokes the execution of the compiled program and redirects all standard output to a file called '`result`'. If the program compiles and the program runs successfully, the file '`result`' should contain the number of fitness cases which were classified correctly.

To extact this value, the result file is opened and the first integer value found is assigned to the variable `fitness`. The result file is then closed.

A number of tests are done on the fitness variable to ensure that it is a number. Finally, the fitness is assigned to the EO individual using the member function `EOT::fitness`. The function then exits.

7.3.7.8 'eoGEEvalFunc-slang.h'

The file 'eoGEEvalFunc-slang.h' declares the class eoGEEvalFunc required by EO for individual evaluation. This implementation uses the S-Lang library.

The class eoGEEvalFunc is a template class which allows the individual type EOT to be specified when an instance of the class is created. The constructor is responsible for initialising the GE mapper and setting up the GCC or TCC evaluator backend. The mapper object is initialised by specifying the maximum number of mapping wraps and loading the BNF grammar file using the readBNFFile member function. Each of these values are supplied as parameters to the constructor function. If the process of loading the grammar fails, an error message is printed to the standard error stream (std::cerr) and the program exits.

The function initialises the S-Lang environment by calling the S-Lang function SLang_init_slang and loads the stdio and math runtime libraries by calling SLang_init_stdio and SLang_init_slmath. If any of these functions fail, the program exits.

Once the S-Lang environment is set up, the file 'GEspiral.sl' (see Section 7.3.6.6 [EXAMPLES/IntertwinedSpirals/GE_MITGALIB/GEspiral.sl], page 109) is loaded using the function SLang_load_file. This file contains the functions and variables required to evaluate a GE phenotype and is only required to be loaded once.

The member operator() provides the definition of the operator required by EO for individual evaluation. It takes one argument of type EOT which represents an EO individual. The function first checks that this individual has not been evaluated previously by calling the member function EOT::invalid. If the individual is invalid, then it needs to be evaluated and assigned a fitness. If the individual is valid, the function exits.

In order to evaluate this individual, the mapper must be supplied with a valid libGE structure. This is constructed by generating an array of integers and copying each element from the EO individual. This array is then supplied to the constructor of the Genotype object. This is then assigned to the mapper object using the member function setGenotype. Once this is complete, a libGE Phenotype is extracted from the mapper object. If the phenotype is invalid, a fitness value of 0.0 is returned and the function exits. If the phenotype is valid, the full executable code needs to be constructed so that the phenotype can be evaluated in the S-Lang backend.

Instead of having to construct the program and write it to a file to be compiled, S-Lang allows the interpretation and execution of the program in memory. This is achieved in this function by creating a string called buffer and appending the start, phenotype, and end code to it. Once the complete program is constructed, the buffer string should contain a program that can execute within the S-Lang runtime.

In order to extract information from the S-Lang runtime environment, a new C++ variable is added to the environment using the SLadd_intrinsic_variable function. This function requires the name of the variable within the S-Lang environment (called Fitness_Variable in this example), the address of the variable in the C/C++ runtime (called fitness in this example), and the type of data stored in the variable (SLANG_INT_TYPE). If any of the S-Lang functions function fail, the program exits.

The contents of the string are executed in the runtime by calling the function SLang_load_string. Once execution has completed, the value in the variable fitness should contain the number of fitness cases which were correctly classified.

A number of tests are done on the fitness variable to ensure that it is a number. Finally, the fitness is assigned to the EO individual using the member function `EOT::fitness`. The function then exits.

7.3.7.9 'eoGEEvalFunc-libtcc.h'

The file '`eoGEEvalFunc-slang.h`' declares the class `eoGEEvalFunc` required by EO for individual evaluation. This implementation uses the libTCC library provided in the TinyCC distribution.

The class `eoGEEvalFunc` is a template class which allows the individual type `EOT` to be specified when an instance of the class is created. The constructor is responsible for initialising the GE mapper and setting up the libTCC evaluator backend. The `mapper` object is initialised by specifying the maximum number of mapping wraps and loading the BNF grammar file using the `readBNFFile` member function. Each of these values are supplied as parameters to the constructor function. If the process of loading the grammar fails, an error message is printed to the standard error stream (`std::cerr`) and the program exits.

In order to minimise the time taken to extract, evaluate and award a fitness to a GE phenotype, the start and end code that is placed before and after a GE phenotype are loaded into character buffers in this function. These buffers are accessed in the function `objfunc` because they are required for every GALib individual. In this example, the C code in the file '`spiralstart.c`' and '`spiralend.c`' are read into the `char` arrays `SPstart` and `SPend` respectively. The last task in this function is to load the file '`GEspiral.c`' into another character buffer called `GEfunc`. This file contains the definitions of the functions and variables used either directly or indirectly by the GE phenotype code. For more information about these files, see Section 7.3.2 [GEspiral.c], page 101 and Section 7.3.3 [spiralstart.c], page 102, and Section 7.3.4 [spiralend.c], page 102.

The member `operator()` provides the definition of the operator required by EO for individual evaluation. It takes one argument of type `EOT` which represents an EO individual. The function first checks that this individual has not been evaluated previously by calling the member function `EOT::invalid`. If the individual is invalid, then it needs to be evaluated and assigned a fitness. If the individual is valid, the function exits.

In order to evaluate this individual, the mapper must be supplied with a valid libGE structure. This is constructed by generating an array of integers and copying each element from the EO individual. This array is then supplied to the constructor of the `Genotype` object. This is then assigned to the `mapper` object using the member function `setGenotype`. Once this is complete, a libGE `Phenotype` is extracted from the `mapper` object. If the phenotype is invalid, a fitness value of 0.0 is returned and the function exits. If the phenotype is valid, the full executable code needs to be constructed so that the phenotype can be evaluated in the libTCC backend.

The first task is to set up a new libTCC state which can compile and execute C code. This is achieved by calling the function `tcc_new`. The function `tcc_set_output_type` is then called to specify that compilation of the C code should take place in main memory to maximise performance. The math library `m` is also added to the libTCC state using the `tcc_add_library` function to allow the use of a number of math functions at the evaluation stage.

The next step is to make the various functions and variables that are defined in the code stored in the buffer `GEfunc` available in the libTCC state. This is achieved by calling the function `tcc_compile_string`, and supplying the current libTCC state and the buffer containing the code as arguments. It is important to note that the code in the `GEfunc` buffer is not executed at this stage. The purpose of this call is simply to compile the code and make it available to the phenotype code which will be compiled next.

Before the phenotype can be executed in the libTCC state, it must be wrapped with the start and end code stored in the `SPstart` and `SPend` buffers. The `char` buffer `buffer` is used to store the result of concatenating the buffer `SPstart`, the phenotype code, and the buffer `SPend`. An example of this concatenation is shown in Section 7.3.5 [WrappedCPhenotype], page 102. This buffer is then passed to the libTCC state using the function `tcc_compile_string`. If the compilation is successful, the libTCC state contains the executable form of the wrapped phenotype, the supporting functions, and the variables required to evaluate the GE phenotype.

In order to execute the code in memory, the `main` function (declared in the file '`spiralstart.c`') must be explicitly invoked. This function is accessed by first calling the function `tcc_relocate`. This makes all function and variable symbols accessible through the C interface and should be called after every call to `tcc_compile_string`. The function `tcc_get_symbol` is then called with a pointer to the libTCC state, a reference to a variable of type `unsigned long`, and the name of the symbol we require ("main"). If the required symbol is found, the `tcc_get_symbol` function stores its address in the `unsigned long` variable. This variable is then cast into a function pointer of type `int (*p_tccMainFunc)()`. The main function can the be invoked in the libTCC state by calling the function through the pointer i.e. `p_tccMainFunc()`.

One the main function has completed, the function `objfunc` must invoke the function `AssignFitness` to retrieve the fitness value of the GE individual. This is performed using the same method as the main function invocation shown above. A function pointer which matches the signature of the function `AssignFitness` must first be created. In this example, the pointer is called `p_tccFitnessFunc"`. The libTCC function `tcc_get_symbol` is then called to store the symbol location of this function in the temporary variable `val`. This variable is then cast and stored in the function pointer. The return value of the `AssignFitness` function can then be retrieved by invoking the function through the pointer in the normal fashion i.e. `fitness = p_tccFitnessFunc();`. The function then deletes the libTCC state using the function `tcc_delete`.

A number of tests are done on the fitness variable to ensure that it is a number. Finally, the fitness is assigned to the EO individual using the member function `EOT::fitness`. The function then exits.

7.3.7.10 'eoGEEvalFunc-lua.h'

The file '`eoGEEvalFunc-lua.h`' declares the class `eoGEEvalFunc` required by EO for individual evaluation. The intertwined spirals example uses the C API of the Lua library to interpret, execute, and evaluate the GE phenotype. Evaluation of the phenotype in this way results in significant improvements in execution time over language compilers e.g. GCC, and TCC. The following section describes the various files and functions that are part of the Lua backend for the intertwined spirals example.

In order to access the Lua C API through the C++ code in the file 'eoGEEvalFunc-lua.h', the #include <header.h> directives must be enclosed in a special directive called extern "C" e.g.

```
extern "C"
{
#include <lua.h>
#include <lauxlib.h>
#include <lualib.h>
}
```

This tells the compiler that the code in these files is C code and not C++ code. A global pointer to a lua_State, called L in this example, is also created at the top of the file.for use by all the functions in the file.

This class is a template class which allows the individual type EOT to be specified when an instance of the class is created. The constructor is responsible for initialising the GE mapper and setting up the libTCC evaluator backend. The mapper object is initialised by specifying the maximum number of mapping wraps and loading the BNF grammar file using the readBNFFile member function. Each of these values are supplied as parameters to the constructor function. If the process of loading the grammar fails, an error message is printed to the standard error stream (std::cerr) and the program exits.

The Lua state is opened in this function by a call to the lua API function lua_open. Additional math and io libraries are added to this state using the function luaL_openlibs. Once completed, the Lua state is then ready to interpret and execute Lua code.

In order to minimise the time required for individual evaluations in the function objfunc, the Lua code in 'GEspiral.lua' is loaded into the state using the function luaL_dofile. This function loads and executes the Lua code in this file which is discussed in more detail in Section 7.3.6.12 [EXAMPLES/IntertwinedSpirals/GE_MITGALIB/GEspiral.lua], page 113. It is important to note that, because the file only contains variable declarations and function implementations, no code is executed at this time. If an error occurs while loading this file, Lua sets a error message at the top of its call stack. Each element on the Lua stack has an index e.g. an index of -1 represents the element at the top of the stack. This index value is supplied to thelua_tostring to retrieve the error message. The message is then printed to standard error output and the program exits.

The member operator() provides the definition of the operator required by EO for individual evaluation. It takes one argument of type EOT which represents an EO individual. The function first checks that this individual has not been evaluated previously by calling the member function EOT::invalid. If the individual is invalid, then it needs to be evaluated and assigned a fitness. If the individual is valid, the function exits.

In order to evaluate this individual, the mapper must be supplied with a valid libGE structure. This is constructed by generating an array of integers and copying each element from the EO individual. This array is then supplied to the constructor of the Genotype object. This is then assigned to the mapper object using the member function setGenotype. Once this is complete, a libGE Phenotype is extracted from the mapper object. If the phenotype is invalid, a fitness value of 0.0 is returned and the function exits. If the phenotype is valid, the full executable code needs to be constructed so that the phenotype can be evaluated in the Lua backend.

Instead of writing the phenotype code to a new file, the code is constructed in memory by appending the required start code, phenotype code, and end code to a `string` object (The start and end code are described in Section 7.3.6.13 [EXAMPLES/IntertwinedSpirals/GE_MITGALIB/LuaStartCode], page 114 and Section 7.3.6.14 [EXAMPLES/IntertwinedSpirals/GE_MITGALIB/LuaEndCode], page 114).

This code is executed in the Lua runtime by calling the function `luaL_dostring` and passing it a pointer to the Lua state and the buffer containing the code to be executed. If execution fails, the error message is retrieved from the Lua stack and printed to standard error output and the program then exits.

If the compilation and execution is successful, the function `lua_getglobal` is called to retrieve the Lua variable `sumFitCaseTimes`. This function pushes the variable on to the top of the stack. The value is checked and retrieved from the stack using the functions `lua_isnumber` and `lua_tonumber`. The value is then assigned to the `fitness` variable.

A number of tests are done on the fitness variable to ensure that it is a number. Finally, the fitness is assigned to the EO individual using the member function `EOT::fitness`. The function then exits.

7.3.7.11 'Makefile'

The file '`Makefile`' contained in the example allows you to create a program using a choice of evaluators for your phenotype strings. Use `make` to create all executables, or `make GE*` to create a specific evaluator implementation (where `*` is any of `GCC`, `SLANG`, `TCC`, `LIBTCC`, or `LUA`). At the top of the file there are definitions for the location of the GALib, GE, and evaluator specific header and library files. If you have installed any of these in non-standard locations, make sure the values are adjusted to suit your system.

The file '`Makefile`' uses the `-include` compiler flag to use only the relevant include file for your choice of evaluator; if your compiler does not accept that flag, you should replace it with the appropriate flag.

Finally, some of the evaluators require the pre-compilation of the file '`GEspiral.c`'; this is done by the '`Makefile`', when the relevant target implementations are chosen.

7.3.8 Intertwined Spirals Performance

This section looks at the performance obtained with most of the search engines tested, for this specific problem. All search engines used a similar experimental setup: steady-state replacement, population size of 500 individuals, 100 generations, probability of crossover of 0.9, and probability of bit-wise mutation of 0.01. Furthermore, each experiment was performed with four different setups:

- Using a standard GE approach;
- Using effective crossover;
- Using sensible initialisation;
- Using a combination of the last two.

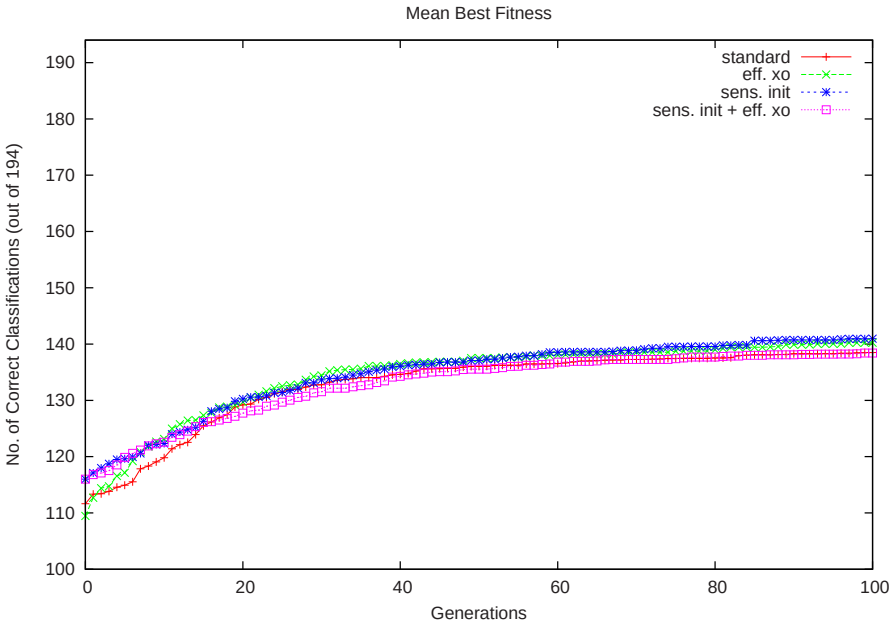
The results show that the standard GE configuration fairs slightly worse than the others and that the use of sensible initialisation alone produces slightly better final individuals than others. The best individual evolved over 30 runs achieved correct classification of approximately 141 individuals.

The average execution times of these runs for each of the five backends and for both search engines are shown below.

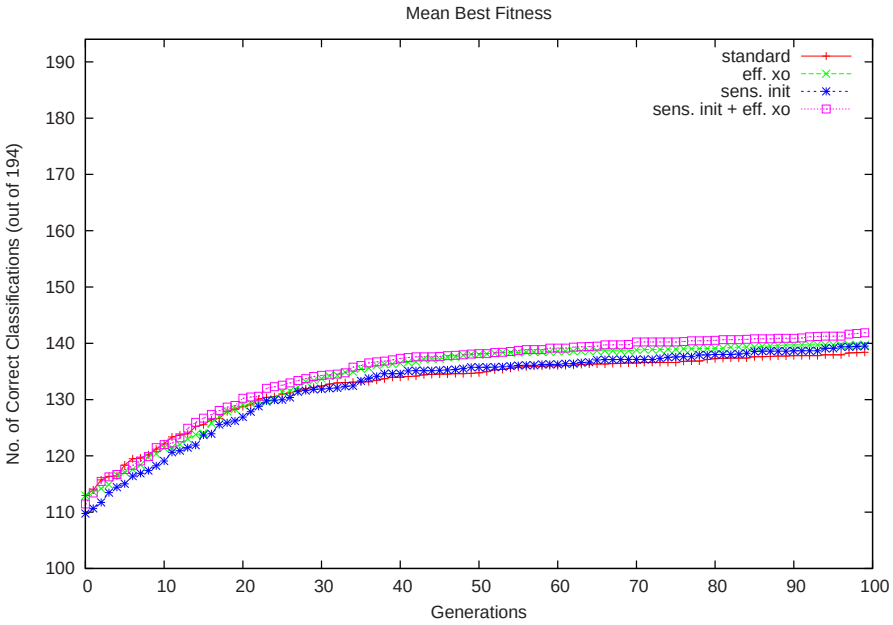
- GCC: 66m41.456s(GALib) 62m54.525s (EO)
- TCC: 12m56.476s (GALib) 12m13.464s (EO)
- LIBTCC: 98m20.012s (GALib) 8m49.939s (EO)
- SLANG: 1m25.292s (GALib) 1m8.705s (EO)
- LUA: 0m46.493s (GALib) 0m38.655s (EO)

For reference, these results were obtained on a PC with a Pentium IV processor running at 2.6GHz, 2048MB of memory, using Linux.

7.3.8.1 GALib performance



7.3.8.2 EO performance



Appendix A Frequently Asked Questions

Q: When trying to compile libGE, I get the following problem:

```
/usr/include/gcc/darwin/2.95.2/g++/stl_vector.h:
In method `_Vector_base<char,allocator<char> >::~_Vector_base()':
/usr/include/gcc/darwin/2.95.2/g++/stl_vector.h:115:
template instantiation depth exceeds maximum of 17
```

A: This seems to be a limitation of older versions of GCC (specifically, versions 2.95 or lower). Try using a newer version of GCC; 3.0 or higher is recommended. (Note: if you have two versions of GCC on your system, specify which version to use through the `configure` script; for example, `./configure CC=gcc-3.0 CXX=g++-3.0`).

Q: When running the `configure` script, the check for GALib fails, but GALib used to run fine on my system.

A: The version of GCC you are using now might be different than the one you used when installing GALib on your system. Try recompiling GALib with the new compiler.

Q: My grammar loads without any errors, but I am getting non-terminal symbols in my phenotype strings, why?

A: Your BNF specification does not conform with the format accepted by libGE. Check the examples provided (see Chapter 7 [Examples], page 57).

Q: When compiling a program using GALib with SLang, I get the following error:

```
/usr/local/include/ga/GA1DArrayGenome.C:342: error: syntax error before `>'
token
```

A: Both GALib and slang took the unfortunate decision of using the variable `ARRAY_TYPE` on their files: slang as an equivalent for its own `SLANG_ARRAY_TYPE`, and GALib as a template name. To solve this problem, make sure you do `#include<ga/ga.h>` **before** you do `#include<slang.h>`.

Q: I am getting the following error:

```
# libGE has been configured to use GALib, but GALib was not found. #
# Please re-install GALib, or reconfigure libGE and re-install it. #
# Alternatively, edit the libGEdefs.h file in the include directory. #
```

A: When you originally configured and installed libGE, a copy of GALib was found on your system, and support for GALib was enabled; however, the GALib headers can no longer be found. This causes the compilation of any program using libGE headers to fail, even if they do not use GALib.

The easiest solution is to reconfigure, recompile and reinstall libGE. Otherwise, remove the line defining the symbol `GALIB_INCLUDEPATH` from the file '`libGEdefs.h`', which was installed in the libGE include directory (typically '`/usr/local/include/GE/libGEdefs.h`').

Appendix B Copying This Manual

B.1 GNU Free Documentation License

Version 1.2, November 2002

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document *free* in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or non-commercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

   If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

   If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

   If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

   It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

   You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

   A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any,

be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their

titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

   You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

   The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

   In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

   You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

   You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

   A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

   If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

   Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warrany Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

   If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

   You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

    The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

    Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

Appendix C References

D. E. Goldberg: Genetic Algorithms in Search, Optimization and Machine Learning. Addison Wesley. (1989)

M. O'Neill: Automatic Programming in an Arbitrary Language: Evolving Programs with Grammatical Evolution. PhD Thesis, University of Limerick (2001)

M. O'Neill and C. Ryan: Grammatical Evolution. IEEE Transactions on Evolutionary Computation, Vol. **5**, No. 4. (2001) 349–358

M. O'Neill and C. Ryan: Grammatical Evolution - Evolving programs in an arbitrary language. Kluwer Academic Publishers. (2003)

R. E. Smith, D. E. Goldberg and J. A. Earickson: SGA-C: A C-language Implementation of a Simple Genetic Algorithm. TCGA Report No. 91002. (1994)

C. Ryan and A. Azad: Sensible Initialisation in Grammatical Evolution. In: GEWS 2003: Grammatical Evolution Workshop, GECCO-2003, Chicago, USA. (2003)

Concept Index

Function Index

I

L

M

O

P

R

S

T

U