Chapter 2
Why yet another one evolutionary optimizer?

He who lives without folly isn't so wise as he thinks.
Francois de La Rochefoucauld

The idea of *Evolutionary computation* implies the existence of suitable tools to perform computations. Such tools have to be designed pondering the environment in which they will operate and the problems to which they will be applied, together with the chosen evolutionary technique. Every design process implies choices, some of which may not be immediately clear to the end user, but can have far-reaching consequences.

The chapter tries to motivate the creation of μGP, yet another one evolutionary optimizer. Its goal is to provide the reader with a rationale for the perceived needs and the consequent taken decisions. The text shows, in an uttermost narrative style, some of the possible alternatives faced during the early design phase.

2.1 Background

The term "evolutionary optimizer" does not indicate a well-defined program structure or user interface, exactly as "word processor" is suitable for a wide range of functional approaches and interfaces. It may be maintained that the purpose of an evolutionary tool is to automate the artificial evolution of a set of solutions to a given problem. This definition brings to light several related, although almost independent, concepts: the definition of the problem itself; the structure of possible solutions to that problem; the evaluation of the *goodness* of candidate solutions; the operations that allow to manipulate candidate solutions.

In many cases these parts are known from the outset: the problem is well defined; the structure of its possible solutions is known; the evaluation of such solutions straightforward; the most sensible transformations on these solutions simply follows from their structure. For example, one may want to solve the *traveling salesman's problem* (TSP). In this case the problem requires to minimize the total length of a

E. Sanchez et al., *Evolutionary Optimization: the μGP toolkit,*
DOI 10.1007/978-0-387-09426-7_2, © Springer Science+Business Media, LLC 2011

path that passes through a number of fixed points and returns to the start[1]. Since it is known from the start that all must be visited once and only once, a possible solution is a permutation of the points. The goodness of a route is the inverse of its length. And it is intuitive that to transform one permutation into another one some form of reordering, such as a swap, has to be performed[2].

The straightforward approach would be to embed all this information in the tool, resulting in a problem-specific application that performs all the computation and eventually provides the user with one or more optimal results. It could be possible to choose in advance the evolutionary approach, tuning the genetic operators for performance. It could also be possible to write some information about the problem directly in the code.

However, it is often perceived as more efficient to reuse the same approach for different, although related, problems. One more mundane example of this is the generation of assembly programs for two different microprocessors. In this case the goal of the programs may be the same, say verifying the design, but their form is necessarily different. Conversely, another example is the generation of programs for a single microprocessor, but with different goals. In this case the form is kept, but the fitness function changes.

We are incline to believe that a truly versatile evolutionary tool is not available at the time we are writing. And such a tool would be useful both for the practitioners and for the researchers. μGP is meant to be able to solve quite different problems, this means that it has to be able to represent quite diverse objects and to assess them using a fitness function which is not known in advance. Thus, both the form and meaning of the individuals cannot be fixed in the code, but a flexible internal representation must be used. This is not just convenient to avoid redundant design efforts, but allows using several different evolutionary approaches for the same problem, or, conversely, to perform evolution on different kinds of individuals, possibly at the same time. The fitness function is unknown to the tool developer not only regarding its possible values, but also regarding its general form.

2.2 Where to draw the lines

From the above discussion it is clear that not all the work can be performed by the evolutionary tool itself: μGP cannot compute the fitness function for a given individual without external help. This stress out the difference between *genotype* and *phenotype*. The tool is able to manipulates solutions at the level of phenotype, while fitness can be defined only at the level of genotype. Indeed, μGP could not even map the phenotype to a sensible genotype, creating a meaningful description of that individual, without additional information. One of the first issues to take care

[1] In graph theory, TSP corresponds to the NP-hard problem of finding the Hamiltonian cycle with the minimal weight.

[2] Remarkably, several approaches in the evolutionary computation literature do neglect this consideration.

of, then, is a classical interface definition problem: it must be decided what part of the work is done at the phenotypic level by the evolutionary engine and what has to be done otherwise, such as by post-processing some results.

An evolutionary process is a closed loop: a population is transformed in a different one by recombining and modifying its component individuals, every new individual is assigned a fitness value and the new population undergoes a *survival* phase. After that the cycle begins anew. There is a feedback from the individuals to the evolutionary core, in the form of a fitness value. The standard practice in electronic design, when implementing a circuit with feedback, is to isolate an inner amplifying block and select the overall system function changing the feedback function. In an analogous way, the evolution of the individuals may be isolated from their fitness computation. This may be done in different ways, changing the definition of what the evolutionary core provides as output and what it accepts as feedback.

This analogy between an electronic circuit and an evolutionary tool is loose, but intuitively it makes sense. In an electronic circuit the purpose of the amplifier is to provide energy to the signals, while the feedback block tells "how wrong" the output is. In an evolutionary process the reproduction phase produces new features (the "energy" of the process), and the fitness function tells how good every solution is. The analogy should not be taken further, as the two domains are too different, but it gives a good starting point to decompose the entire approach.

Another part of the loop that could be separated from the rest is the transformation of the individuals to an external form. The tool does not know, and indeed it should not know, whether it is generating assembly programs, Hamiltonian paths in a graph or coefficients of a polynomial. It stores an internal representation of the evolved individuals, that does not contain information neither about their *semantics*, nor regarding their final appearance.

The main decomposition of the μGP approach is related to the phases of the evolutionary process involved. Every individual is first generated, either during an initial phase or from other individuals, then transformed into the object it represents, and eventually assigned a fitness value. These three phases must be kept as distinct as possible in order to achieve versatility.

2.3 Individuals

There are two main requirements for the internal format of individuals in a versatile evolutionary tool: the representation must allow mapping arbitrary concepts; the representation must allow arbitrary manipulation. The first requirement is stringent, but the latter can be slightly soften. The bottom line is that the representation must guarantee a great expressive power, while permitting a *reasonable* amount of manipulation without *excessive* computational effort. Indeed, the design of the individuals is strongly related to the design of the genetic operators manipulating them. Amongst the cornerstones of natural evolution are the idea of small variations accumulated over generations, and the concept that the offspring inherits from parents

qualifying traits. The artificial evolution process must conform as much as possible: the tool must be able to mutate individuals *slightly*, and breed new specimen without loosing *too much* information.

The two main aspects in defining individuals are: what types of data are stored and how they are structured. Types of data and structure are almost orthogonal aspects. Thus, the two choices may be approached quite independently. Regarding the type of data, there are several alternatives not to limit the application scope. At the two extremes of the spectrum one may find: adopt an extremely *generic* representation that can be tight to any specific problem at a later time; embed all kind of possible representation in the tool and let the final user pick up one for his problem.

The solution adopted in μGP is to embed a limited number of *standard* data types, and let the final user exploit the ones needed. Among the standard types are: integer numbers and real numbers, both with definable ranges. A generic *enumerable* data type with a user-defined set of possible vales, like $\{0,1\}$[3], $\{true, false\}$ or $\{red, blue, green\}$.

Choosing the most generic data and the simplest possible structure, the representation would be is a fixed-length vector. Moving toward the other extreme, there is no clear end to the complexity that can be reached. Indeed, a fixed-length bit vector also allows implementing a wide range of genetic operators with negligible effort. However, while it is theoretically possible to represent any object as a bit vector, this is not unusually a good idea. When solving the TSP, one may encode the vertexes as binary numbers and simply juxtapose them to represent a path. Thus, a fixed-length bit vector would be suitable to encode all possible solutions. It is manifest, however, that such a choice would cause most bit vectors not to encode *any* solution at all, broadening the search space over useless regions.

The problem exists because the concepts that the individuals represent can have some *structure*, and loosing this information always leads to an unreasonable widening of the search space. Dependencies between one part of the individual and another are precious hints in building a viable solution. For example, if an individual expresses a function, there can be dependencies between an operator and other ones, whose result is used as an operand. While the simple vector structure is able to contain a representation of the function, it would not be easy to manipulate it without disrupting the underlying structure, especially if recombination is used.

Moreover, the fixed length of the individuals put an arbitrary limit on the complexity of the possible solutions to the problem. In the cases where this complexity cannot be predicted in advance, it forces the user to either oversize the individuals, or to make (un)educated guesses on the expected optimum solution. Both solutions are plainly unacceptable. Variable-length bit vector would solve the latter problem, introducing only a slight increment in the complexity of the operators.

A far more better possibility in this respect is a tree representation, like the standard genetic programming. It would allow to perform some recombination without disrupting the structure of the individuals, for example by exchanging entire subtrees between two genotypes. When the data inside the tree structure are of different

[3] Why "0" and "1" are considered two constants and not two integer numbers will become clearer in the following.

types, a blind exchange becomes almost unusable. But it is always possible to add information to leaves and nodes to prevent disruptive operations. The only true limitation with a tree structure is that it inherently disallows cyclic dependencies. For example, it would be both tricky and unnatural to represent the recursive definition of the factorial function using a tree, or a backward jump inside an assembly function.

To overcome this limitation, the structure adopted in μGP is based on graphs. More precisely, as it will be apparent in chapter 3, an individual is encoded as a set of directed multigraphs. That is, graphs where a direction is assigned to each edge, and the same pair of vertexes may be joined by more than one edge. Since graphs are not required to be *connected*[4], the use of a set of graphs instead of a single one is not imposed by necessity, it may nevertheless ease the task for the end users. In μGP individuals, some data are inside nodes. Additionally, together with the data types mentioned above, the edges themselves are used to store information. The offspring is thus bred by swapping subgraphs between parents, modifying the graphs structure and altering the data stored inside nodes.

2.4 Problem specification

Tackling a specific problem implies defining an appropriate *fitness function*. That is, how candidate solutions are appraised with respect to the pursued goal. It is not limiting to maintain that the result of an evaluation can be expressed as a positive real value, and that higher scores are better than lower ones. In μGP the fitness is actually a *vector* of real positive values, but this can be seen as a mere simplification when exploiting the tool.

The fitness function cannot be included in the evolutionary core, and there are several alternative to let the user provide it. The fitness function may be added to the evolutionary tool source code and eventually compiled and linked with it. Or it may be provided as an external library dynamically loadable. μGP adopts a quite radical approach: the fitness function is calculated by an external program that is simply *invoked* by the tool.

The nature of the problem also calls for a certain appearance of the solutions. Internally, individuals are encoded as multigraph, but they presumably need to be transformed in some way before being evaluated. Since the fitness evaluator is an external tool, it would be theoretically possible to select a canonical form for representing a multigraph and leave to the fitness evaluator the burden to transform it to a more convenient format. However, to ease the employ of the tool, μGP provides the external evaluator a file describing the individual in a suitable format. Fort example, an assembly program ready to be assembled and linked, or a sequence of cities.

Encoding an individual as a multigraph allows a great generality. However, too much versatility may be deleterious. It must be remembered that an evolutionary

[4] There is a path linking any two vertices in the graph.

optimizer needs to know at least *some* information about the structure of the individuals to evolve. Questions such as "how many cities does the considered TSP instance include?" or "should there be functions in the assembly program, and what is their form?" directly affect the possible operations on the individuals. The answers to questions like these define what is a *legal* individual. That information has to be provided to the tool before evolution can be started. It composes a set of *constraints* that describe the allowed structure of an individual, thus limiting the potentially infinite productions of the tool and avoiding useless computation. For instance, such constraints should not only specify that an assembly function always begin with a certain prologue and end with an epilogue that contain a limited number of parameters, but also what assembly instructions compose them, as well as the body of the function.

To maximize the applicability of the tool, the problem must be specified in a standardized format, readable both by humans and by mechanical tool. Hence, μGP adopts XML for all its input files.

2.5 Coding Techniques

The idea of maximizing the applicability also impact the adopted coding techniques. μGP was originally conceived as a tool to generate assembly test programs for test and validation. It was, nevertheless, a *versatile* tool, in the sense that it could handle the assembly language of different microprocessors.

The first fully operational version was developed in 2002 and it was composed of a few hundred lines of C code and a collection of scripts. The second version was developed in 2003 and maintained since 2006; it consisted of about 15,000 lines in C. This version added several new features and significantly broadened the applicability of the tool. It was able to load a list of parametric code fragments, called *macros*, and optimize their order inside a test program. With time, it has been coerced into solving problems it was not meant for. While useful for improving its performance, this extended usage made the basic limitations of the tool clear, and ultimately led to the need to re-implement μGP from scratch.

This decision follows a complete change of paradigm: the focus passes from the problem to the tool, and the main design goal shift from the solution of a specific class of problems to the development of a tool that can *include* as many as possible. The development of the third version started in 2006 with the intent to provide a clean implementation able to replicate the behavior of the previous version. Additional goals were: maintainability, extendability, and portability. At the end of 2010, the third version of μGP counted up to more than 50,000 lines of C++.

From the programmer perspective, the optimization tool is merely a frontend that parses options and configuration files, and eventually calls functions from a set of libraries. Thus, the *command-line* frontend provided in the distribution can be regarded as a simple example of the use of the underlying libraries.

Libraries themselves are internally organized in layers. The foundation is composed of the routines for handling graphs, taking into account all the user-defined constraints. Piled up on this layer, the user will find the functions for handling individuals, then populations. The whole structure of layers is implemented in C++ trough an extensive use of overloading and inheritance mechanisms.

All genetic operators have been packed inside a different library that make use of the functions to work on individuals. Routines for handling populations and the alternative core evolutionary process are also available. Thus, a programmer may choose at exactly which level insert his code.

Finally, two *auxiliary* libraries complete the set. A powerful mechanism for logging the current status of the process, able to handle different files and different levels of verbosity; and a parser for XML files that have been simply *included* in the project, but it has been developed externally[5].

A number of *ancillary programs* are also included in the basic distributions. These programs do not run or control the evolution process itself, but perform useful actions, easing the work for the final user. Such utilities also exploit the μGP core libraries.

[5] TinyXML was initially written by Lee Thomason, and it is now maintained by the original author with help from Yves Berquin, Andrew Ellerton, and the tinyXml community. The library is available under the *zlib* license on *SourceForge* from `http://sourceforge.net/projects/tinyxml/`.