GUIDE: Unifying Evolutionary Engines through a Graphical User Interface

Pierre Collet, Marc Schoenauer

GUIDE: Unifying Evolutionary Engines through a Graphical User Interface

Pierre Collet[1] and Marc Schoenauer[2]

[1] Laboratoire d'Informatique du Littoral, Université du Littoral - Côte d'Opale
[2] Projet Fractales, INRIA Rocquencourt
Pierre.Collet@univ-littoral.fr, Marc.Schoenauer@inria.fr

Abstract. Many kinds of Evolutionary Algorithms (EAs) have been described in the literature since the last 30 years. However, though most of them share a common structure, no existing software package allows the user to actually shift from one model to another by simply changing a few parameters, e.g. in a single window of a Graphical User Interface. This paper presents GUIDE, a *Graphical User Interface for DREAM Experiments* that, among other user-friendly features, unifies all kinds of EAs into a single panel, as far as evolution parameters are concerned. Such a window can be used either to ask for one of the well known ready-to-use algorithms, or to very easily explore new combinations that have not yet been studied. Another advantage of grouping all necessary elements to describe virtually all kinds of EAs is that it creates a fantastic pedagogic tool to teach EAs to students and newcomers to the field.

1 Introduction

As Evolutionary Algorithms (EAs) become recognised as practical optimisation tools, more and more people want to use them for their own problems, and start reading some literature. Unfortunately, it is today very difficult to get a clear picture of the field from papers or even from the few books that exist. Indeed, there is not even a common terminology between different authors.

Many papers (see e.g. [5]), refer to the different "dialects" of Evolutionary Computation, namely Genetic Algorithms (GAs), Evolution Strategies (ESs), Evolutionary Programming (EP) or Genetic Programming (GP).

¿From a historical perspective, this is unambiguous: broadly speaking (see [10] for more details), GAs were first described by J. Holland [14] and popularised by D.E. Goldberg [12] in Michigan (USA); ESs were invented by I. Rechenberg [20] and H.-P. Schwefel [22] in Berlin (Germany); L. Fogel [11] proposed Evolutionary Programming on the US West Coast; J. Koza started the recent root of Genetic Programming [16].

However, when the novice reader tries to figure out the differences between those dialects from a scientific perspective, she/he rapidly becomes lost: from a distance, it seems that GAs manipulate bitstrings, ESs deal with real numbers and GP handles programs represented by parse-trees. Very good —so the difference seems to lie in *representation*, i.e. the kind of search space that those dialects search on.

But what about EP then ? Original EP talks about Finite State Automata, but many EP papers deal with parametric optimisation (searching a subspace of $\mathbf{R}^n$ for some $n \in \mathbf{N}$) and many other search spaces; and the "real-coded GAs" also do parametric optimisation, while inside the GA community, the issue of representation is intensively discussed [18,26,23], and many *ad hoc* representations are proposed for instance for combinatorial problems [19]. Some ESs also deal with combinatorial representations [13] or even bitstrings [3]; and even in the field of GP, which seems more clearly defined by the use of parse-trees, some linear representations are currently used [17].

So our patient and persevering newcomer starts delving into the technical details of the algorithms, and thinks he has finally found the fundamental differences: those dialects differ by *the way they implement artificial Darwinism.*

- Indeed, GAs use proportional or tournament-based selection, generate as many offspring as there are parents, and the generation loop ends by replacing all parents by all offspring.
- In ESs, each of the μ parents generates a given number of offspring, and the best of the λ offspring (resp. the λ offspring + the μ parents) are deterministically selected to become the parents of the next generation. The algorithm is then called a $(\mu, \lambda) - ES$ (resp. $(\mu + \lambda) - ES$).
- EP looks very much like a $(P+P)-ES$, except that competition for survival between parents and offspring is stochastic.
- In GP, only a few children are created at each generation from some parents selected using tournament-based selection, and they replace some of the parents that are chosen using again some tournament. But wait, this way of doing is precisely called ... Steady-State GA —so this is a GA, then !

And indeed, GP started as a special case of GA manipulating parse-trees and not bitstrings (or real numbers or ...), and became a field of its own because of some other technical specificities, but also, and mostly, because of the potential applications of algorithms that were able to create programs and not "simply" to optimise.

"Ah, so the differences come from the applications then ?" will ask the newcomer. And again, this will only be part of reality, as there is in fact no definite answer: all points of view have their answer, historical, technical, "applicational" or even ... political (however, this latter point of view will not be discussed in this paper, devoted to scientific issues).

But what does the newcomer want to know ? She/He wants to be able to use the best possible algorithm to solve her/his problem. So she/he is certainly not concerned with historical differences (apart from curiosity), and she/he wants to find out first what exactly is an Evolutionary Algorithm, and what different parameters she/he can twiddle to make it fit her/his needs.

Starting from the target application, a newcomer should first be able to choose any representation that seems adapted to the problem, being informed of what "adapted to the problem" means, in the framework of Evolutionary Computation: the representation should somehow capture the features that seem important for the problem at hand. As this is not the central issue of this paper,

we refer the user to the literature, from the important seminal work of Radcliffe [18] to more recent trends in the choice of a representation [6].

The only other thing for which the user cannot be replaced is of course the choice (and coding) of the fitness function to optimise. But everything else that a user needs to do to run an Evolutionary Algorithm should be tuning some parameters, e.g. from some graphical interface: many variation operators can be automatically designed [24,23], and most implementations of artificial Darwinism can be described in a general framework that only requires fine tuning through a set of parameters.

This paper addresses the latter issue, with the *specification of any evolution engine, unified within a single window* of a Graphical User Interface named *GUIDE* (where *evolution engine* means the way artificial Darwinism is implemented in an EA in the selection and replacement steps). In particular, this paper will **not** mention any representation-specific feature (e.g. crossover or mutation), nor any application-specific fitness function.

Section 2 briefly recalls the basic principles of EAs, as well as the terminology used in this paper. Section 3 presents the history of GUIDE, based on the specification language *EASEA* [9]. Section 4 details the Evolution Engine Panel of GUIDE, demonstrating that it not only fulfills the unification of all historical "dialects," but that it also allows the user to go far beyond those few engines and to try many more yet untested possibilities. Finally, section 5 discusses the limitations of this approach, and sketches some future issues that still needs to be addressed to allow a wide dissemination of Evolutionary Algorithms.

2 Basic principles and terminology

Due to the historical reasons already mentioned in the introduction, even the terminology of EAs is not yet completely unanimously agreed upon. Nevertheless, it seems sensible to recall here both the basic skeleton of an EA and the terminology that goes with it. This presentation will however be very brief, as it is assumed that the reader is familiar with at least some existing EAs, and will be able to recognise what she/he knows. Important terms will be written in boldface in the rest of the paper.

2.1 The skeleton

The goal is to optimise a **fitness function** defined on a given search space, mimicking the Darwinian principle that *the fittest individuals survive and reproduce*. A generic EA can be described by the following steps:

- **Initialisation** of *population* Π_0, usually through a uniform random choice over the search space;
- **Evaluation** of the individuals in Π_0 (i.e. computation of their fitnesses);
- **Generation** i builds population Π_i from population Π_{i-1}:
  - **Selection** of some parents from Π_{i-1}, biased by the fitness (*the "fittest" individuals reproduce*);

  - Application (with a given probability) of **variation operators** to the selected parents, giving birth to new individuals, the **offspring**; Unary operators are called **mutations** while n-ary operators are called **recombinations** (usually, $n = 2$, and the term **crossover** is often used);
  - **Evaluation** of newborn offspring;
  - **Replacement** of population Π_i by a new population that is created from the old parents of population Π_{i-1} and the newborn offspring, through another round of Darwinian selection (*the fittest individuals survive*).
- Evolution stops when some predefined level of performance has been reached, or after a given number of generations without significant improvement of the best fitness.

An important remark at this point is that such a generic EA is made of two parts that are completely orthogonal:

- the *problem-dependent* components, including the choice of the search space, or space of **genomes**, together with their initialisation, the variation operators and of course the fitness function.
- on the other hand, the **evolution engine** implements the artificial Darwinism part of the algorithm, namely the selection and replacement steps in the skeleton given above, and should be able to handle populations of objects that have a fitness, regardless of the actual genomes.

2.2 Discussion

This is of course a simplified view, from which many actual algorithms depart. For instance, there are usually two search spaces involved in an EA: the space of genomes, or **genotypic space**, where the variation operators are applied, is the space where the actual search takes place; and the **phenotypic space**, where the fitness function is actually evaluated. The mapping from the genotypic space onto the phenotypic space is called the decoding of solutions, with the implicit assumption that the solution the user is looking for is the phenotypic expression of the genome given by the algorithm. Though the nature of this mapping is of utter importance for the success of the algorithm, it will not be discussed at all here, as GUIDE/EASEA only considers genotypes, hiding the phenotypic space in the fitness.

Other variants of EAs do violate the pure Darwinian dogma, and hence do not enter the above framework: many variation operators are not "blind," i.e. do bias their actions according to fitness-dependent features; conversely, some selection mechanisms do take into account some phenotypic traits, like some sharing mechanisms involving phenotypic distances between individuals [21].

Nevertheless, we claim that the above generic EA covers a vast majority of existing algorithms[1], and most importantly, is a mandatory step for anyone intending to use Artificial Evolution to solve a given problem. In that context,

[1] It even potentially covers Multi-Objective EAs, as these "only" involve specific selection / replacement steps. However, MOEAs are not yet available in GUIDE, but will be soon.

the existence of a user-friendly interface allowing anyone with little programming skills to design an Evolutionary Algorithm following this skeleton is a clear dissemination factor for EC as an optimisation technique: Such were the motivations for EASEA [9] and, later, for GUIDE.

3 GUIDE overview

GUIDE is designed to work on top of the EASEA language (*EAsy Specification of Evolutionary Algorithms*) [9]. The EASEA language and compiler have been designed to simplify the programming of Evolutionary Algorithms by providing the user with all EA-related routines, so that she/he could concentrate on writing the code that is specific to the application, as listed in section 2, i.e.: the genome structure and the corresponding initialisation, recombination, mutation and evaluation functions.

On the evolution engine side, a set of parameters allows the user to pick up existing selection/replacement mechanisms, and was designed to supersede most existing combinations, while allowing new ones. One of the most important features of EASEA is that it is library-independent: from an EASEA source code, the compiler generates code for a target library, that can be either the C++ libraries *GALib* [25] and EO [15], or, more recently, the Java library JEO [1]. The resulting code is then compiled using the routines from the corresponding library —but the user never has to dive into the intermediate complex object-oriented code.

Unfortunately, whereas the goal of relieving the user from the tedious task of understanding an existing EC library was undoubtedly reached since the very first versions of EASEA, back in 1999, the specification of the evolution parameters implicitly supposes some deep knowledge of existing evolution engines. Moreover, the lack of agreed terminology makes it even difficult for EC-advanced users to pick up their favorite algorithm: GA practitioners, for instance, have hardly heard of the replacement step, because in "standard" GAs, the number of generated offspring is equal to the number of parents, and all offspring simply replace all parents (generational replacement). The need for a graphical interface was hence felt necessary quite early in EASEA history.

Such a graphical interface was eventually developed as part of the DREAM European project IST-1999-12679 [2] (Distributed Resource Evolutionary Algorithm Machine —hence the name GUIDE), whose evolutionary library JEO [1] is one of the possible targets for EASEA compilation.

GUIDE: a quick tour

In a programming environment, the Graphical User Interface is the entry point at the highest possible level of interaction and abstraction. In the GUIDE/EASEA programming environment, the idea is that even a non-expert programmer should be able to program an EA using one of the underlying libraries without even knowing about it ! GUIDE must therefore at least allow the user to:

1. specify the numerous parameters of any evolutionary engine by way of a point-and-click interface,
2. write or view the user code related to problem-specific operators,
3. compile the experiment by a simple click,
4. run the experiment, and visualise the resulting outputs in some window(s).

While the second to fourth points above are merely a matter of implementation, and by no way could justify a scientific paper in a conference, the first point not only is original, but also clearly (graphically) highlights the common features of most existing EC paradigms: the user can specify *any* evolution engine within a single window.

The structure of GUIDE reflects this point of view, and offers four panels to the user (see the tags on Figure 1):

Algorithm Visualisation Panel to visualise and/or modify problem-dependent code. It contains a series of text windows, each window referring to the equivalent "sections" of EASEA source code [9] that the user has to type in. This is where the user writes the code for genetic operators such as initialisation, mutation, recombination, and most importantly for the evaluation function.

Evolution Engine Panel for the Darwinian components. This panel will be extensively described in section 4.

Distribution Control Panel to define the way different islands communicate in a distributed model [2]. This panel will soon be adapted to ParaDisEO [7], the recent Parallel Distributed version of EO.

Experiment Monitor Panel to compile and run the experiment. At the moment, only compilation is possible from there. The user has to run the program from outside GUIDE.

GUIDE obviously also offers (see Figure 1) a top-menu bar from where the user can save/load previous sessions to `Files`, choose the `Target Library` and `Build` the executable file —and, as usual, some of these actions can be fired by some icons from the toolbar.

4 Evolution Engine Panel

This section describes the most innovative feature of GUIDE: the panel where the user can specify the evolution engine, either from pre-defined "paradigmatic" engines, or by designing new combinations of selections and replacements fitting her/his taste.

4.1 Evolution Engines

The concept of evolution engine designates the two steps of the basic EA that implement some artificial Darwinism: **selection** and **replacement**. Basically, both steps involve the "selection" of some individuals among a population. However,

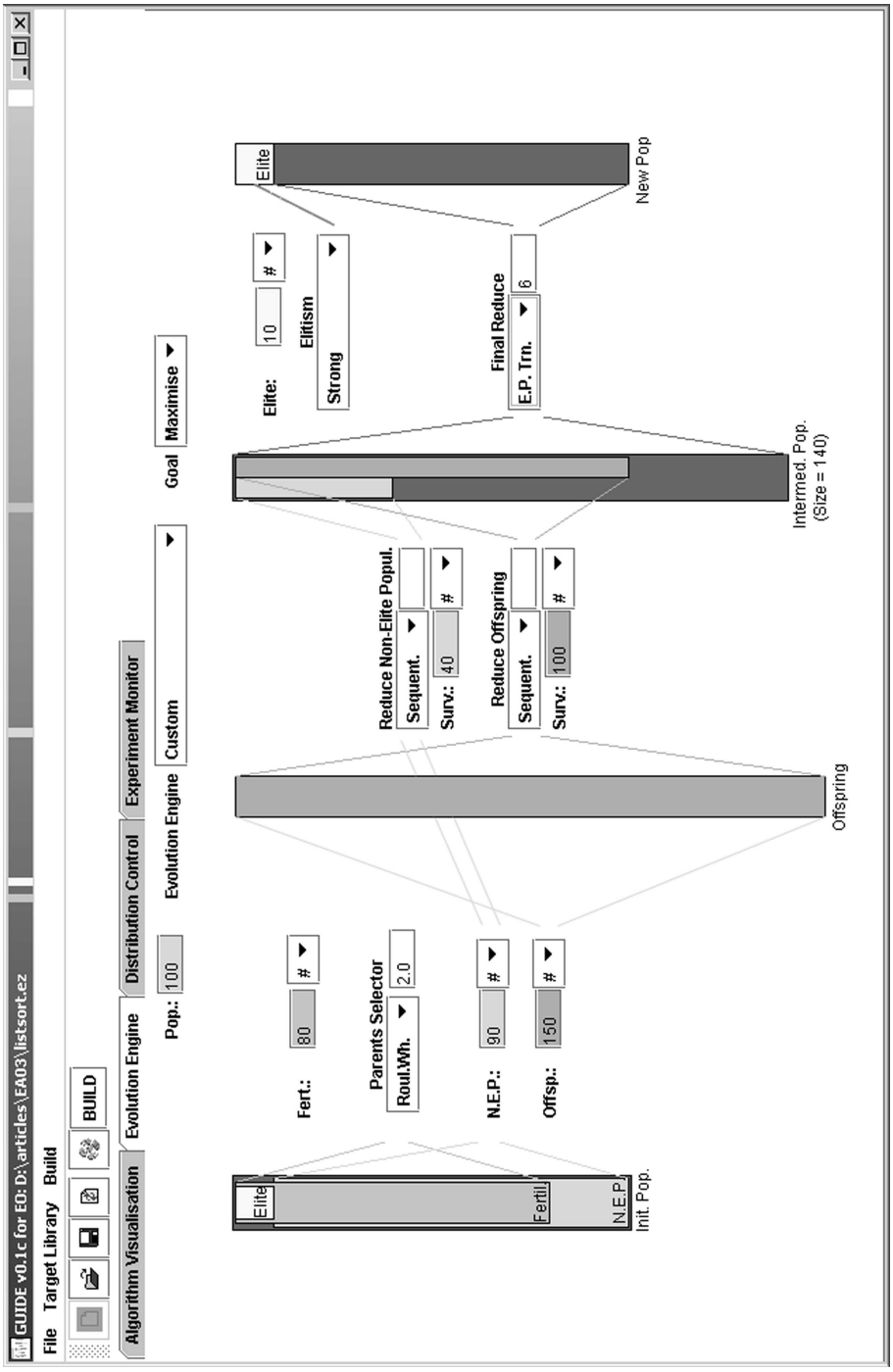

Fig. 1. GUIDE: Evolution Engine Panel

at least one important difference between those steps makes it necessary to distinguish among them, even when only one is actually used (like in traditional GAs or ESs): an individual can be selected *several times* for *reproduction* during the selection step, while it can be selected *at most once* during the *replacement* step (in this case, the non-selected individuals simply disappear).

Another important feature is implemented in a generic way in GUIDE: **elitism**. It will be discussed separately (section 4.2).

4.2 Panel description

Figure 1 shows the Evolution Engine Panel of GUIDE. On top of the panel, the user can specify the population size (here 100), whether the fitness is to be maximised or minimised, and which type of Evolution Engine should be used: this type can be any of the existing known engines, described in next section 4.3, or set to `Custom`, as in Figure 1.

Below are some vertical bars, representing the number of individuals involved in different steps of a single generation: the left part of these bars describes the selection step (including, implicitly, the action of the variation operators, but not their description), and starts with the initial parent population —the leftmost bar, labeled `Init. Pop`. The right two bars specify the replacement step, and end with the `New Pop.` that will be the `Init. Pop.` of next generation —and hence has the same height (number of individuals contained) that the `Init. Pop.` bar.

Parameters input There are two kinds of user-defined parameters that completely specify the evolution engine in GUIDE: the sizes of some intermediate populations, and the type of some selectors used inside the selection and the replacement steps. When a parameter is a type from a predefined list, a pull-out menu presents the possible choices to the user.

All sizes in GUIDE can be set either graphically, using the mouse to increase or decrease the size of the corresponding population, or using the numeric pad and entering the number directly. In the latter case, the number can be entered either as an absolute value, or as a percentage of the up-stream population size —the choice is determined by the small menu box with either the "#" or the "%" character.

Selection The selection step picks up individuals from the parent population `Init. Pop.` and handles them to the variation operators to generate the `Offspring` population. In GUIDE, the parameters for the selection step are:

- the type of selector that will be used to pick up the parents, together with its parameters (if any). Available selectors (see e.g. [8] for the definitions)are `Roulette wheel` and (linear) `ranking` (and an associated selection pressure), `deterministic tournament` (and the associated tournament size), `stochastic tournament` (and the associated probability), plus the trivial `random` (uniform selection) and `sequential` (deterministic selection selecting from best to worse individuals in turn).

- The number of `Fertile` individuals: non-fertile individuals do not enter the selection process at all. This is somehow equivalent to truncation selection, though it can be performed here before another selector is applied among the fertile individuals only.
- The size of the `Offspring` population.

The last field in the "selection" area of the panel that has not been discussed here is the `N.E.P.`, or Non Elite Population, whose size is that of the population minus that of the `Elite` and that will be discussed in section 4.2 below. At the moment, consider that this population is the whole parent population.

In the example of Figure 1, only the 80 best individuals will undergo roulette wheel selection with selection pressure 2.0, and 150 offspring will be generated. (Although roulette wheel is known to have several weaknesses, notably compared to the Stochastic Universal Sampling described by Baker[4] it is still available in GUIDE, mainly because all underlying libraries implement it.)

Replacement The goal of the replacement procedure is to choose which individuals from the parent population `Init. Pop.` and the offspring population `Offspring` will build the `New Pop.` population. In GUIDE, the replacement step is made of three **reduction** sub-steps: a reduction is simply the elimination of some individuals from one population according to some Darwinian **reduce** procedure:

- First, the `Non Elite Population` (the whole `Init. Pop.` in the absence of elitism) is reduced (the user must enter the type of reducer to be used, and the size of the reduced population).
- Second, the `Offspring` population is reduced (and again, the user must set the type of reducer and the size of the reduced population).
- Finally, both reduced populations above are merged into `Intermed. Pop.` (for intermediate population). The corresponding bar is vertically divided into two bars of different colors: the survivors of both populations. This population is in turn reduced into the final `New Pop.`, whose size has to be the size of the parent population: hence, only the type of reducer has to be set there.

The available types of reducers include again the `sequential` and `random` reductions, as well as the deterministic and stochastic `tournaments` (together with their respective parameters), that repeatedly eliminate bad individuals. An additional option is the `EP tournament` (together with the tournament size T): in this stochastic reducer, each individual fights against T uniformly chosen opponents, and scores 1 every time it is better; the best total scores then survive.

Note that many possible settings of those parameters would result in some unfeasible replacements (e.g. asking for a size of `Intermed. Pop.` smaller that the `Pop.` size). Such unfeasible settings are filtered out by GUIDE ... as much as possible.

In the example of Figure 1, the best 40 individuals from the `N.E.P.` (the initial population in the absence of elitism) are merged with the best 100 offspring, and the resulting intermediate population is reduced using an EP tournament of size 6.

Elitism All features related to elitism have been left aside up to now, and will be addressed here. There are two ways to handle elitism in EC literature, termed "strong" and "weak elitism" in GUIDE. The user first chooses either one from the menu in the replacement section, and then sets the number of `Elite` parents in the corresponding input box (setting the `Elite` size to 0 turns off all elitism).

Strong elitism amounts to put some (copy) of the best initial parents in the `New Pop.` *before* the replacement step, without any selection against offspring whatsoever. The remaining "seats" in the `New Pop.` are then filled with the specified replacement. Note that the elite population nevertheless enters the selection together with the other parents —and that, of course, only the `N.E.P.` competes in the reduction leading to the parent part of the `Intermed. Pop`.

Weak elitism, on the other hand, only takes place *after* normal replacement, in the case when the best individual in the `New Pop.` is worse than the best parent of the `Init Pop`. In that case, all parents from the `Elite` population that are better than the best individual of the `New Pop.` replace the worst individuals in that final population. This type of elitism is generally used with an `Elite` size of 1.

For instance in Figure 1, elitism is set to `strong` and the number of elite parents to 10: the best 10 parents will anyway survive to next generation[2]. Only the 90 worse individuals undergo the reduction toward the `Intermed. Pop.` — the 40 best out of these 90 worse will survive this step— and only 90 seats are available in the `New Pop.`, the 10 first seats being already filled by the elite parents.

4.3 Specifying the main evolutionary paradigms

The example in Figure 1 is typically a `custom` evolution engine. This section will give the parameter settings corresponding to the most popular existing evolution engines —namely GAs (both generational and Steady-State), ES and EP. Note that though those names correspond to the historical "dialects," they are used here to designate some particular combination of parameters, regardless of any other algorithmic component (such as genotype and variation operators). However, going back to those familiar engines by manually tuning the different parameters is rather tedious. This is the reason for the `Evolution Engine` pull-out menu on top of the panel: the user can specify in one click one of these five "standard" engines, and instantly see how this affects all the parameters.

[2] Such a strategy is generally used, together with a weak selection pressure, for instance when the fitness is very noisy, or is varying along time.

After choosing one of the pre-defined engines, the user can still modify all parameters. However, such modification will be monitored, and as soon as the resulting engine departs from the chosen one, the pull-out menu will automatically turn back to `Custom`. The predefined engines are:

Generational GA In that very popular algorithm, let P be the population size. P parents are selected and give birth to P offspring that in turn replace the P parents: any selector is allowed, `Fertile` size is equal to `Pop.` size, `Offspring` size is set to `Pop.` size, reduced `N.E.P.` size to 0 (no parent should survive) and `Intermed. Pop.` size to `Pop.` size. In fact, none of the reducers is actually active in this setting (this is a generational replacement).

Furthermore, weak elitism can be set (generally with size 1). `Fertile` size can be reduced (this is equivalent to *truncation selection*) without leaving the `GGA` mode.

Steady-State GA In Steady-State GA, a single offspring is created at each generation, and replaces one of the parent, usually chosen by tournament. Again, any selector is allowed, `Fertile` size is equal to `Pop.` size, but `Offspring` size is set to 1 and the parents are now reduced by some tournament reducer, to `Pop.` size minus one, while the final reducer is not active[3].

The number of offspring can be increased without leaving the `SSGA` mode: there is no clear limitation of this mode, though the number of offspring should be kept small w.r.t. `Pop.` size. And of course, setting any type of elitism here is a misconception.

Evolution Strategies There are two popular evolution engines used in the traditional Evolution Strategies algorithms, the $(\mu, \lambda) - ES$ and the $(\mu + \lambda) - ES$: in both engines, there is no selection step, and all μ parents give birth to λ parents. The μ individuals of the new population are deterministically chosen from the λ offspring in the $(\mu, \lambda) - ES$ and from the μ parents *plus* the λ offspring in the $(\mu + \lambda) - ES$. Both these evolution engines set the selection to `Sequential`, the `Offspring` size and the `Reduce offspring` size to λ (no reduction takes place there) and the final reducer to `sequential`. Therefore, choosing $(\mu, \lambda) - ES$ sets the `Reduce N.E.P.` size to 0 while choosing the $(\mu + \lambda) - ES$ sets it to `Pop.` size.

The $(\mu + \lambda) - ES$ engine already is strongly elitist. The $(\mu, \lambda) - ES$ engine is not, and elitism can be added —but it then diverges from the original ES scheme.

Evolutionary Programming In traditional EP algorithms, P parents generate P offspring (by mutation only, but this is not the point here). Then the P parents plus the P offspring compete for survival. Setting `EP` mode in the Evolution Engine menu sets the `Offspring` size to `Pop.` size, both reduced sizes for `N.E.P.` and `Offspring` to `Pop.` size as well (no reduction here) and the final reducer to

[3] An "age" tournament should be made available soon, as many SSGA-based algorithms do use age as the criterion for the choice of the parent to be replaced.

`EP tournament`. Note that early EP algorithms sometimes used a deterministic choice for the survivors —this can be achieved by choosing the `Sequential` final reducer.

Here again, elitism can be added, but this switches back to `Custom` engine.

5 Discussion and Perspectives

As already argued in section 2.2, the very first limitation of the GUIDE evolution engine comes from the chosen EA skeleton, that does not allow weird evolution engines. However, we firmly believe that most existing EA application use some evolution engine that falls in this framework.

The forthcoming improvements of this part of GUIDE are concerned with adding new selector/reducer options, more specifically selection procedures based on other criteria. The `Age tournament` has already been mentioned in the SSGA context. But all multi-objective selection procedures will also be added (with additional options in the `Maximise/Minmise` pull-out menu.

Going away from the evolution engine, the asymmetry in terms of flexibility between the Evolution Engine and the Algorithm Visualisation Panels cries out for a graphical interface allowing the user to specify the genome structure and the variation operators. Such interface is not as utopian as it might seem at first sight: the genome structure could be specified from basic types, and basic constructors (e.g. heterogeneous aggregations, homogenous vectors, linked lists, trees, ...). And there exist some generic ways to design variation operators for such structures [24].

Of course, last but not least, after the Distribution Panel has been adapted to ParaDisEO, the Experiment Monitor Panel should be redesigned such that the user can graphically plot the evolution of any variable in that window.

As a conclusion, the Evolution Engine Panel of GUIDE is only a first step towards a widely available dissemination tool for EAs. But it already achieves the demonstration that all evolutionary algorithms are born equal if the user is provided with enough parameters to tune. GUIDE Evolution Engine Panel is a visual and pedagogical tool that allows one to understand the intricacies of the different evolutionary paradigms. While, quite often, Graphical User Interfaces reduce flexibility, GUIDE offers at the Evolution Engine level readability, simplicity of use and an easy way to experiment with complex parameters.

References

1. M.G. Arenas, P. A. Castillo, B. Dolin, I. Fdez de Viana, J. J. Merelo, and M. Schoenauer. JEO: Java Evolving Objects. In *GECCO'02*, pp 991–994, 2002.
2. M.G. Arenas, P. Collet, A.E. Eiben, M. Jelasity, J.J. Merelo, B. Paechter, M. Preu, and M. Schoenauer. DREAM: A Framework for Distributed Evolutionary Algorithms. In J.J. Merelo et al., eds., *PPSN VII*, pp 665–675. Springer-Verlag, LNCS 2439, 2002.
3. T. Bäck and M. Schütz. Intelligent mutation rate control in canonical GAs. In Z. W. Ras and M. Michalewicz, eds, *ISMIS '96*, pp 158–167. Springer Verlag, 1996.

4. J. E. Baker, Reducing bias and inefficiency in the selection algorithm. Proceedings of the Second International Conference on Genetic Algorithms, L. Erlbaum Assoc. Eds (Hillsdale), 1987.
5. Th. Bck, D.B. Fogel, and Z. Michalewicz, eds. *Handbook of Evolutionary Computation*. Oxford University Press, 1997.
6. P. J. Bentley and S.Kumar. Three ways to grow designs: A comparison of embryogenies for an evolutionary design problem. In *GECCO'99*, pp.35-43, 1999.
7. S. Cahon, N. Melab, E-G. Talbi, and M. Schoenauer. ParaDisEO-based design of parallel and distributed evolutionary algorithms, this volume, 2003.
8. U. Chakraborty, K. Deb, and M. Chakraborty. Analysis of selection algorithms: A Markov chain approach. *Evolutionary Computation*, 4(2):133–168, 1996.
9. P. Collet, E. Lutton, M. Schoenauer, and J. Louchet. Take it EASEA. In M. Schoenauer et al., eds, *PPSN VI*, LNCS 1917, pp 891–901. Springer Verlag, 2000.
10. D.B. Fogel. *Evolutionary Computing: The Fossile Record*. IEEE Press, 1998.
11. L. J. Fogel, A. J. Owens, and M. J. Walsh. *Artificial Intelligence through Simulated Evolution*. New York: John Wiley, 1966.
12. D. E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison Wesley, 1989.
13. Michael Herdy. Self-adaptive population size and stepsize in combinatorial optimization problems:solving magic squares as an example. In *Proc. GECCO-2002 Workshops*. Morgan Kaufmann, 2002.
14. J. H. Holland. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, Ann Arbor, 1975.
15. M. Keijzer, J. J. Merelo, G. Romero, and M. Schoenauer. Evolving Objects: a general purpose evolutionary computation library. In P. Collet et al., eds, *Artificial Evolution'01*, pp 229–241. Springer Verlag, LNCS 2310, 2002.
16. J. R. Koza. *Genetic Programming: On the Programming of Computers by means of Natural Evolution*. MIT Press, Massachusetts, 1992.
17. Peter Nordin and Wolfgang Banzhaf. Evolving turing-complete programs for a register machine with self-modifying code. In L. J. Eshelman, ed., *ICGA'95*, pp 318–325. Morgan Kaufmann, 15-19 1995.
18. N. J. Radcliffe. Forma analysis and random respectful recombination. In R. K. Belew and L. B. Booker, eds, *ICGA'91*, pp 222–229. Morgan Kaufmann, 1991.
19. N. J. Radcliffe and P. D. Surry. Fitness variance of formae and performance prediction. In L. D. Whitley and M. D. Vose, eds, *Foundations of Genetic Algorithms 3*, pp 51–72. Morgan Kaufmann, 1995.
20. I. Rechenberg. *Evolutionstrategie: Optimierung Technisher Systeme nach Prinzipien der Biologischen Evolution*. Fromman-Hozlboog Verlag, Stuttgart, 1973.
21. B. Sareni and L. Krähenbühl. Fitness sharing and niching methods revisited. *Transactions on Evolutionary Computation*, 2(3):97–106, 1998.
22. H.-P. Schwefel. *Numerical Optimization of Computer Models*. John Wiley & Sons, New-York, 1981. 1995 – 2^{nd} edition.
23. P.D. Surry. *A Prescriptive Formalism for Constructing Domain-specific Evolutionary Algorithms*. PhD thesis, University of Edinburgh, 1998.
24. P.D. Surry and N.J. Radcliffe. Formal algorithms + formal representations = search strategies. In H.-M. Voigt et al., eds., *PPSN IV*, LNCS 1141, pp 366–375. Springer Verlag, 1996.
25. M. Wall. Overview Matthew's Genetic Library. http://lancet.mit.edu/ga/.
26. D. Whitley, S. Rana, and R. Heckendorn. Representation issues in neighborhood search and evolutionary algorithms. In D. Quadraglia et al., eds., *Genetic Algorithms and Evolution Strategies in Engineering and Computer Sciences*, pp 39–58. John Wiley, 1997.