Soft Comput (2010) 14:1091–1109
DOI 10.1007/s00500-009-0504-3

ORIGINAL PAPER

Algorithm::Evolutionary, a flexible Perl module for evolutionary computation

Juan Julián Merelo Guervós · Pedro A. Castillo · Enrique Alba

Published online: 1 October 2009

Abstract This paper describes Algorithm::Evolutionary (`A::E`), a Perl module released under an open source license and designed for the exploration and exploitation of evolutionary algorithms. We describe the design decisions taken to enhance flexibility, how performance was improved by using several implementation tweaks, and what kind of design patterns were applied for its development. This work also tries to dispel the myth of low performance of scripting languages by comparing it with a state-of-the-art library (ECJ) written in Java. Besides, we are interested in assessing its efficiency in several possible evolutionary settings, finding out what kind of behavior we can expect, and what can be done to improve it. Applications already written using `A::E` are also described, along with how it can be used to create new operators. Finally, some conclusions are drawn from the design experience.

Keywords Evolutionary algorithms · Evolutionary computation · Perl · CPAN · Frameworks · Design patterns

J. J. Merelo Guervós (✉) · P. A. Castillo
Depto. Arquitectura y Tecnología de Computadores, ETS Ingenierías Informática y Telecomunicaciones, Universidad de Granada, Granada, Spain
e-mail: jmerelo@geneura.ugr.es

P. A. Castillo
e-mail: pedro@geneura.ugr.es

E. Alba
Depto. Lenguajes y Ciencias de la Computación, Universidad de Málaga, Malaga , Spain
e-mail: eat@lcc.uma.es

1 Introduction

At first glance, Perl might not seem the best choice for doing evolutionary computation (EC) (Bäck et al. 1997; Eiben and Smith 2003), since it is an interpreted (or scripting) language (Ousterhout 1998; Loui 2008), and EC applications are claimed to require high performance (mainly in the fitness evaluation stage, but also when applying string or vector processing operations). However, Perl has a feature that makes it ideal for EC (and, obviously, many other applications): it has been designed to work as *glue* (Ousterhout 1998), that is, as a way of interfacing many different devices, protocols, applications, and file formats. Perl features an exhaustive collection of modules (which roughly correspond to *libraries* in other languages) available from a central location called CPAN (Comprehensive Perl Archive Network, at http://cpan.org (Ashton and Hietaniemi 2006), that includes many mature modules that have been in use and development for more than a dozen years) for text and data manipulation (Bates 1993), interfacing with databases and the world wide web (Blansit 2006), handling complex data structures, and even writing Perl programs in Latin (Lingua::Romana::Perligata, written by Damien Conway). In fact, it has also been used extensively for scientific computing (Baiocchi 2004). Performance has been improved in each new version (current stable one at the time of writing this is 5.10), and it has very recently received additional support from big players like Microsoft (Adam 2008), which enhances its status of multi-platform scripting language.

There are also other features that make it outstanding among other languages: the quality of the references available for it, specially the *Camel Book* (Wall et al. 2000) (as of late November 2008, #4 in bestseller in the Unix books category, and #39 in the *Programming* category, after many web books and a few Python and Javascript ones), the *Llama book* (Schwartz and Phoenix 2008), and others such as (Wainwright et al. 2001) (all of them with four or more stars in the Amazon bookstore); the availability of technical support for developers, with highly knowledgeable user communities (called Perl Mongers) in almost any country and human language (as Eric S. Raymond says (Raymond 1998), "Perl's true strength is the collective talents of its enthusiasts"); and, finally, the portability of its implementations: all Perl interpreters compile from the same source, so a program that runs in a platform can run in almost any other machine with a Perl interpreter. Besides, Perl is also the language of choice among many communities, such as BioInformatics (Stajich et al. 2002) and Computational Linguistics (Pedersen et al. 2004).

In the case of evolutionary algorithms, the initial intention for building this library was to use XML (eXtensible Markup Language) documents (Harold 2001; Networks 2009; Ray 2001) as a description of an algorithm from which it could be run, and using the same XML format as a persistent object facility (that is, a way to permanently store the state of an evolutionary algorithm); the language that described Evolutionary Algorithms was called EvoSpec (Merelo et al. 2003), and its specification was done using a DTD (Document Type Definition) or XSchema (equivalent way of describing an XML dialect using XML itself) (Fallside 2004). Classes in `Algorithm::Evolutionary` (`A::E`) have an alternative syntax in XML, and all classes have a constructor that creates an object from its XML description.

However, eventually, XML remained as a format for serializing evolutionary algorithm objects and `A::E` evolved as a way of implementing new ideas in evolutionary computation. And since Perl modules use namespaces as prefixes for compartmentalization (the concept is similar to that used in Java), it was named `Algorithm::Evolutionary` to place the module within the `Algorithm` hierarchy. Its object-oriented design makes the implementation of new operators quite fast (as will be shown within this paper), and the expressiveness of Perl turns the addition of new features in a quite straightforward operation. That means that, even as the performance of programs running might not be as fast as in other languages (such as Java or C++), this disadvantage is more than offset by the fact that development time might be cut short by several orders of magnitude, as proved some time ago by Prechelt (2000) and more recently by Fourment and Gillings (2008), since programs are much more expressive and a lot of features are already prepackaged in standard libraries; this also results in shorter programs that are easier to debug and understand, facilitating its use as a base for doing a slightly different thing via small changes. This makes it convenient for teaching evolutionary computation to beginners, and, it fact, it has been used for postgraduate-level courses in implementation of evolutionary algorithms at the University of Granada.

During the close to seven years (the first announcement, still viewable at http://fon.gs/ae-ann/ was made in 2002) this module has been released, it has been used extensively in many applications, mainly in those created by our own team (Castillo et al. 2008; Merelo et al. 2008b); however, in this paper we will describe the current features of the `A::E` library as compared with other libraries in the market; we will write a small primer on how to use it from scratch, give some measures on its performance in different environments, and finally show how it can be used to create and test a new operator. Our aim is to show the niche in which using this EC library could be more convenient than other alternatives already in the market. Besides, we will try to show how this library conforms to Gagné and Parizeau's criteria for a generic evolutionary computation framework presented in Gagné and Parizeau (2006); a *generic* framework allows, in principle, to represent any kind of evolutionary algorithm, possibly by means of an extension that conforms to the framework's API (*application programming interface*). Another of the issues approached in this paper will be those arising in the design of libraries, how we validate decisions taken in the design and the influence of the choice of internal data structures to represent the building blocks of evolutionary algorithms on performance.

The rest of the paper is organized as follows: Sect. 2 shows the state of the art in evolutionary algorithm frameworks, with special attention to those written in Perl. Right next, Sect. 3 describes the `A::E` evolutionary computation library, its design decisions, and how it can be used. Section 4 makes several performance measures of benchmark programs, checking different Perl interpreters and measuring its performance, and Sect. 5 will show the details on how it can be applied to create a new operator. The paper ends with some discussion, conclusions, and future work.

2 State of the art

Nowadays, there is a broad choice of evolutionary algorithm frameworks almost in every language, and adapted to every need. From specific-purpose libraries such as GAGS (Merelo-Guervós and Prieto 1996) or lil-gp and other GP libraries (Wilson et al. 2004), nowadays libraries emphasize its general applicability and extensibility. For instance, JCLEC (Ventura et al. 2008) is a general-purpose Java evolutionary computation framework, which has been included in the KEEL data mining library (Alcalá-Fdez et al. 2009). Other well-known frameworks include ParadisEO (Cahon et al. 2004), which was initially a branch of the EO Evolving Objects library (Keijzer et al. 2002), and of course ECJ (Luke 2009), probably the most widely-used general-purpose evolutionary computation library, already in double-digit version.

The wide availability and maturity of these libraries has not been a hurdle for the creation of new ones such as Shark (Igel et al. 2008), or Watchmaker (Dyer 2008), both available as open source software. This is due to several factors: some libraries are initially created with a single problem in mind, and eventually expanded to a whole range of problems, until they become a framework. In other cases, they are simply written in a language that is not familiar to the person needing them, or do not include the features that are needed for implementing a new operator or higher-level algorithm.

Scripting languages are usually more flexible in that sense. Many things can be changed in runtime, making designs much more flexible, even at the Application Program Interface (API) level, which can in some cases be built from scratch via runtime commands. This usually comes at the cost of performance, but familiarity with language, speed of implementation, and wide (and, in the case of Perl and some other languages like Ruby, centralized) availability of programming libraries more than make up for it. That is why most of the new frameworks for evolutionary computation (such as EvoGrid, available from http://launchpad.net/evogrid) use these languages (Lee and Kim 2005; Merelo et al. 2007; Klein and Spector 2007).

Despite this flexibility, there are not many libraries that implement evolutionary algorithms in Perl (for a review on evolutionary algorithms in Perl, and a tutorial of the first versions of `A::E`, see Merelo-Guervós (2002); most published modules deal with genetic programming (hereafter GP) in Perl, due to the fact that it is an interpreted language, and it is very easy to evaluate expressions and statements from a program (or script). The first (as claimed by the author) paper published on the subject seems to be one by Baldi et al. (2000), but the source code itself was not published, and no hypothesis can be done on its features. From that moment on, there are several papers that describe Perl implementation of evolutionary algorithms: Kunken (2001) described an application that evolves words that "look as if they were English", or *fake English* words, by trying to evolve them using the same letter pattern that English uses. The same application is also mentioned by Zlatanov (2001), who implements a genetic programming system, with source code available, to solve the same problem.

From then on, there are several papers about doing genetic programming (Koza 1992) in Perl: the first one was written by Murray and Williams (1999), which, despite its title, actually describes a genetic programming system, similar to another mentioned in the PerlMonks site [http://perlmonks.org/index.pl?node_id=31147] (a meeting place for practitioners). Several other introductions to genetic algorithms with code have been published in the same place [http://perlmonks.org/index.pl?node_id=298877, http://perlmonks.org/index.pl?node_id=330315], but the first mention to a module that implements a canonical genetic algorithm was done in Neylon (2001). This module, called `Algorithm::Genetic`, cannot be easily extended or adapted to new paradigms, since it is a single file with all data structures and algorithms used already built-in into the file. McCallum (2003) has also presented a system called PerlGP, used specifically in the context of bioinformatics, which has extensive facilities, including a database back-end for serialization, and its main advantage is that it uses as a programming language for doing GP in Perl itself. Its main drawback is its specificity: it is not intended for general evolutionary computation, and most data structures and methods are geared towards GP.

The most complete (apart from the one presented in this paper) and peculiar implementation of evolutionary algorithms in Perl is called myBeasties (Hugman 2003), which eventually became a module called AI::GP. This system implements different kinds of objects, that can be evolved in many possible ways; there is a language that describes these transformations. It is an interesting system, but its extensibility is not so strong, and the learning curve is also somewhat steep, since it involves learning a new language apart from Perl itself. It is mainly used for evolving Perl scripts, the same way that Genetic Programming evolves Lisp functions, not intended for the implementation of a general evolutionary computation program, which implies also learning structures unfamiliar for the EA practitioner. On the other hand, the most recent is `AI::Genetic::Pro`, which has recently entered version 0.34. The main objective of this module (Lukasz 2009) is to optimize speed through coding the most critical parts in C, through the Perl interface called XS that allows this. In fact, initial tests (see equivalent programs at our CVS server: http://opeal.cvs.sourceforge.net/viewvc/opeal/Algorithm-Evolutionary/benchmarks/, `ai-genetic-pro.pl` and `bitflip.pl`) show that it is several times slower than `A::E`, with extensibility being also sacrificed through the use of this XS API.

The majority of those systems do not make use Perl's capabilities to implement an object-oriented library, easily adaptable and expandable, which have been two of the objectives `A::E`'s designers had in mind.

Creating an EA library is closely related to the design of a language that can be used to describe experiments in that library, so that results can be easily reported and logged, and experiment design can be done without modifying a program, and if possible from somebody who is not familiar with programming. For some time, XML has been the language of choice for representation of complex data structures; that is why it has also been employed traditionally for EAs. The first approach was the EAML language (Evolutionary Algorithm Markup Language, an XML dialect) described by Veenhuis et al. (2000). This language, specified as a set of XML tags with a defined and fixed semantics, specifies an evolutionary algorithm, from which C++ code can be generated. EAML attempts to be an exhaustive description of an evolutionary algorithm, but it does not really achieve its target, since, for instance, variation operators are tags, instead of being attribute values of a generic *operator* tag; this means that adding a new operator [say, a n-ary *orgy* operator (Eiben et al. 1995)] would require a redesign of the language's syntax (defined through a DTD, or Data Type Dictionary). In general, using element names or tags is less flexible than using attribute values, since attributes can have any value or a constrained one (depending on how you define them in the DTD), but validated XML requires tags to be defined in its DTD first. In any case, it is a valid approach for a restricted form of an evolutionary algorithm and it is a step in the right direction. On the other hand, EvoSpec, which is used by `A::E`, tries to overcome these limitations. Other possible approaches are presented by García-Nieto et al. (2007), who published an application programming interface via a web service, allowing any program to interact, via the Internet, and through the interchange of messages formatted in XML; and by Ventura et al. (2008), who also used XML (in a way more similar to EAML than to EvoSpec) to represent evolutionary algorithms.

Being convenient as it is for algorithm description, serializing Perl data structures to XML is not easy, mainly due to the fact that the DTD it is mapped into has to be defined in advance; other languages, such as YAML [Yet Another Markup Language, (Ingerson et al. 2001)], have been designed with those data structures in mind, being at the same time more compact (for instance, an article by regeya (2004) notes a 10% reduction of the number of bytes from representing the same data structure with YAML or XML) and thus convenient. In the latest versions of `A::E` (from 0.56 on), this language has been incorporated and used within a wide variety of tasks, including serialization and program configuration.

3 Overview of the Algorithm::Evolutionary library

In this section we will show the building blocks of an evolutionary algorithm, and how they have been implemented in the `A::E` library. These building blocks, which have been described extensively in papers such as Alba and Troya (2001) and Merelo-Guervós 2002, consist mainly of individuals, operators, and algorithms. *Individuals* are the subjects of evolution, and include mainly a *fitness* and a *chromosome* that is evaluated to obtain it. Individuals are organized in *populations*, and changed using *operators*, which, from as early on as Michalewicz's book (1996) and the design of the EO evolutionary objects library (Keijzer et al. 2002), are considered *independent* of the data structures they operate on. Operators have different scale, from those considered *genetic* that operate on a single or a few individuals, to the *evolutionary algorithms* that operate on a population.

Design decisions have been taken bearing in mind mainly flexibility and easiness of programming, as well as leverage of native Perl capabilities for increased performance and easiness of use, leading to design decisions that will be presented next. First, we will look at the individual level in Subsect. 3.1, then at operators in 3.2, fitness functions will be dealt with in Sect. 3.3, higher-level algorithms are presented in 3.4, and finally ancillary methods and facilities in the last Sect. 3.5.

3.1 Creating and handling single individuals

The basic operation in an evolutionary algorithm is the handling of the data structure used for representing the solution to the problem. While canonical genetic algorithms (Whitley 1994) use bitstrings to represent any solution, evolutionary algorithms in general accept any data structure (in fact, any object) can be used (Alba and Troya 2001; Merelo-Guervós et al. 2000; Michalewicz 1996) to represent them. This fact must be reflected in the library design, which should include a minimal interface for evolved objects: the only common feature of all evolvable data structures is the fitness. Most of them are also sequential, and can be visualized as a vector (of bits, of characters, of floating point numbers or other data structures), which adds other methods: `Atom`, and `size`, which would mean that these operations should be mostly universal for all data structures intended to represent solutions, and thus, the Perl code outlined in Fig. 1 should always work. This interface can be extended trivially to multi-chromosome and *atomic* data structures by considering every chromosome a single atom, and give it size one. Of course, that would mean that general operators like crossover and mutation might not work, which is not really a problem, since it is easy to use specific operators, and there are no implicit defaults in most of the algorithms used.

In order to enforce this interface (conventionally rather than physically, since Perl is not an strongly typed language), an abstract base class, `A::E::Individual::Base` has been created, with several others descending (directly or indirectly) from it: `Bit_Vector`,

```
my $individual = new Algorithm::Evolutionary::Individual::Data_Structure;
for (my $i = 0; $i < $individual->size(); $i ++ ) {
  do_stuff($individual->Atom($i));
}
```

Fig. 1 Elements of the universal interface for individuals used in a program; this program should be valid for any class `$individual` is instantiated into

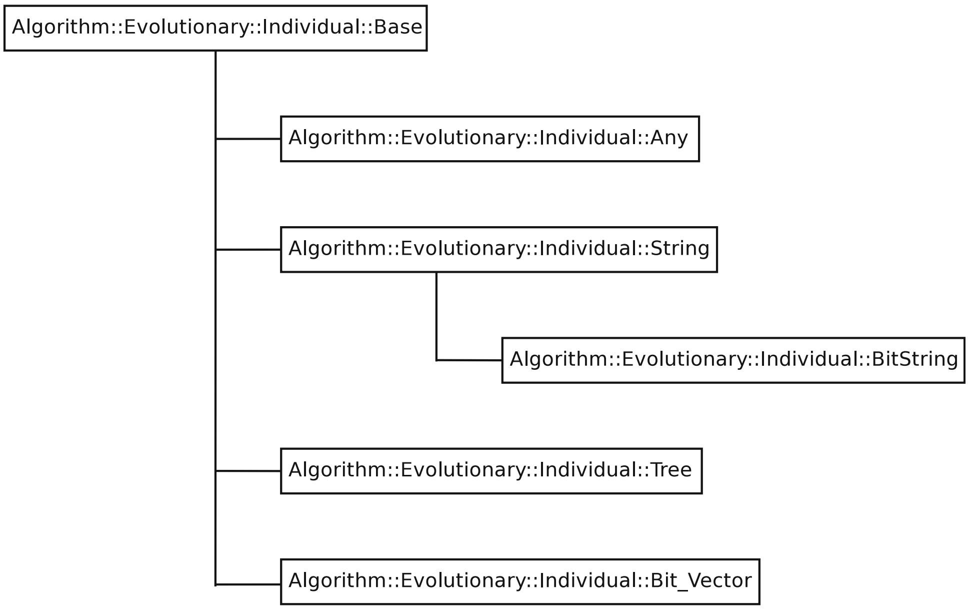
Algorithm::Evolutionary::Individual::Base
Algorithm::Evolutionary::Individual::Any
Algorithm::Evolutionary::Individual::String
Algorithm::Evolutionary::Individual::BitString
Algorithm::Evolutionary::Individual::Tree
Algorithm::Evolutionary::Individual::Bit_Vector

Fig. 2 Class hierarchy for individuals, data structures used to represent problem solutions

`BitString`, `String`, `Vector`, `Tree` and `Any`. The class diagram is shown in Fig. 2; and includes a gamut of possible data structures that encompass many possible problems; besides, there are two possible representations for the canonical bit string: `BitString` and `Bit_Vector`, whose main difference is the internal representation they use; the first uses strings, while the second used bit vectors packed into long integers, which is more efficient memory-wise, but might have an impact on performance, as we will see later on.

Since Perl has no explicit mechanism for enforcing abstract base classes, reimplementation of abstract methods has been forced by raising an exception (using the `croak` instruction, which does precisely that) if a base class *abstract* method is called; this is mostly a warning for reimplementers, but works well enough for anybody wishing to create new classes in this hierarchy.

One of the functions that are frequently performed on individuals is cloning, that is, producing a copy of an individual; this is an operation performed extensively in an EA, used to copy individuals into a new population or produce copies of existing ones to perform operations on them. In fact, the profiling of several programs found that applications were spending more time in this operation than in any other, so the local `clone` method was eventually substituted by the equivalent method of the module `Fast::Clone`. However, `clone` is kept in the interface as a deprecated feature. This change improved performance by up to 20% in some cases.

The advantages of this interface, being minimal, is that it is quite easy to implement new classes by just wrapping a few methods around a Perl module (as it has been done with `Tree` around `Tree::DAG_Node` and with `Bit_Vector` around `Bit::Vector`[1]). In fact, a minimal wrapper implementing the *Decorator* pattern for any class was easily written by just sticking a `Fitness` method around a class that could be defined in runtime; as shown in Fig. 3, this has been managed by using the `A::E::Individual::Any` class.

Finally, serialization is done through two different mechanisms: via the aforementioned YAML and the XML-based EvoSpec (Merelo et al. 2003), which will be used throughout the whole library.

[1] `Tree::DAG_Node` and `Bit::Vector` classes, and many more, can be found in the helpful CPAN, or Comprehensive Perl Archive Network.

```
my @array = qw (1 2 3);
push @array, ['5','6'];
my     $indi = new Algorithm::Evolutionary::Individual::Any 'Object::Array',
\@array;
$indi->Fitness( 0.01 );
```

Fig. 3 Using `A::E::I::Any` as a `Decoration` for the `Object::Array` class, which can then be used as an evolvable object

3.2 Operators

In `A::E`, an operator is anything that acts on a set of individuals (possibly a set with a single element) and returns another set; since parameter passing in Perl does not bind variables to arguments until the subroutine is called, this has been implemented via the use of a single name for the method that actually employs the operator `apply`; thus, in general, any object that implement that method can be considered an operator in `A::E`.

In practice, a hierarchy for operators has also been designed and implemented, including the most usual ones, and falling mainly in three classes: crossover-like, mutation-like and population-level operators. Every individual in the `A::E::Individual` class is provided with at least a mutation and a crossover-like operator, and it is considered a good practice that for every individual data structure that is added, a couple of operators providing this functionality should be also included. Some error checking is done when operators are applied to make sure that the individual has the interface needed for applying the kind of changes the operator does. This design choice also implies that operators such as mutation and crossover are classes independent of the individual, not methods attached to a particular class of individuals, being thus an example of the *Visitor* pattern. This means that which operator is applied to which individual can be decided in runtime, and operator rates as well as anything else related to them can have the maximum flexibility. In this sense, the base class `A::E::Op::Base` also acts as a factory which can receive a serialized version (that is, a text version that can be stored on a file or transmitted to another computer via a network connection) or a description of an operator and produce a full-fledged object, as shown in Fig. 4.

The mechanism that allows all operators to change their probability of application in runtime is to give each one a *priority* (as opposed to a fixed *probability*), as was already done in EO (Keijzer et al. 2002). Priorities become probabilities by adding them up and normalizing to one; but by making each priority independent of the others, it is quite easy to change, at any moment, the probability of an operator by calling the `rate` method; this same method is used to recall the operator priority in higher-level operators that need them. Of course, priorities can be set by just assigning them so that they add up to 1, which would then be equivalent to assigning probabilities. These priorities, however, should not be confused with internal operator application rates on those operators that use them (for instance, a mutation operator that can change a variable number of bits); each operator has an specific interface to set these kind of quantities.

Another couple of functions are needed in this hierarchy; `arity`, which is called every time an operator is applied to find out how many individuals it operates on, and `check`. Since `arity` is called before every operation, it is a candidate for optimization: a cache mechanism included in the Perl `Memoize` module has been used for this. Memoize caches function calls and results automatically, and a single instruction (`memoize(arity)`) is needed to use it. The improvement obtained depends on the cost of the evaluation function, but it can be up to 10% of the program running time. This has been tested via profiling and running of several real-world problems, and not published here since it is not the main point of the paper, and it could distract the reader from the actual focus on design and validation.

Tests were also made to optimize the `check` method (which checks if operands belong to the correct class) also by caching, but results obtained were not satisfactory. However, the best optimization here is to eliminate this function, since it is mainly intended to provide an informative message to the client in case it is applied incorrectly; if performance is an issue, checking can be done at design time and eliminated in runtime.

Eventually, operators can be applied in a loop to create new members of the population using code such as the one shown in Fig. 5. Knowing in advance which individuals the operators are going to be applied to (instead of selecting

```
my $new_op =  Algorithm::Evolutionary::Op::Base->fromXML( $xml );
```

Fig. 4 Creation of an operator using `A::E::Op::Base` as a factory and from an XML string that follows EvoSpec

```
my @offspring;
my $selectedOp = $ops[ $opWheel->spin()];
for ( my $j = 0; $j < $selectedOp->arity(); $j ++ ) {
    my $chosen = $genitors[ rand( @genitors )];
    push( @offspring, $chosen->clone() );
}
my $mutant = $selectedOp->apply( @offspring );
```

Fig. 5 Application of operators (contained in reference to array `$ops`); the operator is randomly chosen by spinning a roulette wheel (`$opWheel`). Depending on the arity of the operator, a set of (clones of) operands is chosen, and eventually the operators are applied

them *within* the operator) offers more flexibility, since complex selection policies can be implemented and created independently of the operator code. In fact, some of them have been implemented and included in the library.

3.3 Fitness classes

Although from the `A::E` point of view a fitness function is anything that receives a chromosome and returns something (anything, really) and it is impossible to include them all, just a few fitness classes have been included in the distribution as an example. These are functions usually employed for testing different evolutionary algorithms, and we found it interesting to distribute them as default. As of June 2009, fitness functions included are the Massively Multimodal Deceptive Problem (MMDP), the P_Peaks and wP_Peaks (Araujo et al. 2008; Giacobini et al. 2006a, Royal Road (Mitchell et al. 1992), ONEMAX, Error Correcting Codes (ECC), the Trap family of functions and the Knapsack problem. In fact, creating a fitness of this class is not compulsory, only convenient. Fitness *objects* can be either members of this hierarchy or references to subroutines, even closures (anonymous subroutines), as shown in Fig. 7.

In this case, the `Base` class includes an evaluation counter and a method for applying the fitness function to a chromosome, called `apply`, as usual. In cases where it is also possible (notably in individuals that use bitstrings as representation, as seen above), a in-memory cache of evaluated individuals is also included. In general, this speeds up the evolutionary algorithm operation between 10 and 25%; this percentage varies with the amount of times the same chromosomes are generated and sent to evaluate. This is actually a trade off between memory usage and speed; it will add up at least the size of a chromosome for each one it is evaluated, so its memory complexity will be of the order of the number of evaluations. This can actually slow down overall runtime, if too much memory is taken, so it will have to be taken into account beforehand and disabled in case the required memory exceeds roughly the available memory. The general interface for the cache is laid out in the `A::E::Fitness::String` abstract base class, which can be used for all fitness functions that need a string (bitstring or otherwise) as input.

Nothing bans the use of external function evaluations; in fact, `A::E` has been used in several papers (Castillo et al. 2002, 2008) as a general evolutionary computation library which calls external programs that evaluate neural networks parametrized in a file or processor microarchitectures. A general facility for this could have been included, but since the interface for fitness functions is so general, what is usually done is to write a wrapper in Perl that creates the command line, runs the external program, and then receives the result.

This brings us to the subject of performance; it is not a secret that Perl performance is on a par with other scripting languages (see Walker 2005 for an evaluation of floating point performance, and Back (2008) for more integer-oriented performance). Without specific optimizations, and in general, Perl (and other scripting languages such as Python or Ruby) will be 20 times slower than compiled or semicompiled languages such as Java or C++. This might be an issue for complex evaluation functions or for big populations. Although Perl has specific optimizations for data processing (the so-called Perl Data Language), and includes interfaces for anything that can be interfaced (for instance, the statistical package R, other languages like Java or C++, or anything with an external API, a website, a GUI or a command-line interface), this means that for high-throughput evolutionary algorithms, once the algorithm has been tested and checked, it might be necessary to write critical parts (such as the fitness evaluation, or anything coming from a legacy application) in other languages.

In particular, however, the speed is more than adequate for the applications we have used it (Araujo et al. 2008; Castillo et al. 2002, 2008; Merelo-Guervós and Castillo-Valdivieso 2004; Castillo et al. 2007, for instance), and the availability of auxiliary libraries in Perl and the speed with which development and experiment processing afterwards can be done more than compensates for its lack of speed. Running a profiling tool over the programs also helps to identify the bottlenecks, which can then be attacked by using different techniques or implementations in other languages. This is not specific to Perl, but has not been generally found in other EC libraries, so we mention it explicitly as a best practice that has been applied in this case.

3.4 Higher level algorithms

With the degree of development that evolutionary computation has reached, it is impossible for a class library to include everything, even more so when it is mainly designed for researchers whose objective is to develop and test new algorithms quickly. However, since a secondary objective is to use the library for teaching evolutionary computation, a sufficient amount of methods used in current EC paradigms must be included, and, moreover, the skeleton of an algorithm should be in place so that the only thing that has to be included by researchers are their new pieces (and, if possible, just the fitness function).

First, the module should include a straightforward, no-fuss way of defining an evolutionary algorithm, so that the user just has to select a representation and a fitness function, and start to run it. That role is fulfilled by the `A::E::Op::Easy` module, which takes as its only

```
my $mmdp = new  Algorithm::Evolutionary::Fitness::MMDP;
my              $generation = Algorithm::Evolutionary::Op::Easy->new( $mmdp ,
$selection_rate , [$m, $c] ) ;
```

Fig. 6 Declaration of an `Easy` algorithm, with all optional parameters; just the fitness function is needed, the other parameters can be left blank and will be substituted by a sensible default

compulsory argument a reference to a fitness function (which can be a closure or anonymous subroutine, or an object in the `Fitness` class hierarchy); in a few lines of code, an evolutionary algorithm can be declared, as shown in Fig. 6. However, a bit of flexibility is also built in: default values for replacement rate and operators can be changed in runtime, and its change will be taken into account within the algorithm code in subsequent generations.

However, one size does not fit all, and a more complex framework must be also included. This framework for a general evolutionary algorithm should ideally follow the *Strategy* or *Template* pattern (Gamma et al. 1993) like the ones used by Alba et al. (2006), which is basically scaffolding for an algorithm whose different parts are decided at runtime. `A::E` includes among others two modules, `GeneralGeneration`, which is a pluggable module for a single generation in an evolutionary algorithm, and `FullAlgorithm`, which wraps around a *generation* object, and adds provision for termination functions that decide whether the algorithm has completed its run. These flexible algorithms allow the selection of lower-level objects in runtime

`FullAlgorithm` is a generational algorithm, and thus includes a slot for a generation object (be it `General-Generation` or other); this makes it lose a bit of generality, but makes it a bit more time-efficient. In general, an evolutionary algorithm is more complex: it includes a *selector*, a *breeder* that generates the new population, and an *insertor* (Keijzer et al. 2002). A module that includes all those facilities is planned in the library roadmap, but has not been included so far. Some examples for each of the pluggable parts are also provided with the general `A::E` distribution: two selectors (which form the small `Selector` class hierarchy), `Roulette Wheel` and `TournamentSelect`, which will be included into `GeneralGeneration` (or similar) objects and three *terminators*, that check that the conditions for termination have been met: `NoChangeTerm`, `DeltaTerm`, and `GenerationalTerm`. All elements needed to create a `FullAlgorithm` object, and the programming interface to run it are shown in Fig. 7.

3.5 Other classes

There is an aptly named `A::E::Utils` class, which, instead of being instantiated into an object, contains a variety of (exported) functions that can be useful elsewhere; `A::E::Run` and `A::E::Experiment` are self-contained algorithms which take as input algorithm description (in YAML or XML) and run it to completion or step by step. This can be used to set up a battery of tests with different fitness functions, for instance, or when a full algorithm with the corresponding data has to be handled

Fig. 7 Declaration of all objects needed for a `FullAlgorithm`, and (in the last line) application to a (previously declared) population

```
my $m = new Algorithm::Evolutionary::Op::Bitflip; #Changes a single bit
my $c = new Algorithm::Evolutionary::Op::Crossover; #Classical 2-point crossover
my $replacementRate = 0.3; #Replacement rate
my $popSize = 20;
my      $selector = new Algorithm::Evolutionary::Op::RouletteWheel $popSize;
#One of the possible selectors
my $onemax = sub {
    my $indi = shift;
    my $total = 0;
    my $len = $indi->length();
    my $i = 0;
    while ($i < $len ) {
 $total += substr($indi->{'_str'}, $i, 1);
 $i++;
    }
    return $total;
};
my $generation =
    new Algorithm::Evolutionary::Op::GeneralGeneration( $onemax, $selector,
[$m, $c], $replacementRate );
my $g100 = new Algorithm::Evolutionary::Op::GenerationalTerm 10;
my $f = new Algorithm::Evolutionary::Op::FullAlgorithm $generation, $g100;
$f->apply( $population );
```

like an object in a library; this kind of things could be useful when setting up a SOAP (Simple Object Access Protocol) (Box et al. 2000) server (Castillo et al. 2002), for instance.

In general, there are little rules to follow for these utilities. They are forced to use the API defined for the existing classes, but, other than that, they are created and set up when and where needed.

3.6 Design patterns

The use of design patterns that is, *idioms* or common ways of performing tasks in a programming language, was introduced n evolutionary computation as far back as 1998 by Smith (1998), Lenaerts and Manderick (1998) and Nic McPhee in his Sutherland framework (McPhee et al. 1998). Lenaerts and Manderick, for instance, mention design patterns such as Factory, Strategy, Builder, Bridge, Flyweight, and Visitor being used in their GPFrame genetic programming framework. All the principles of using design patterns in evolutionary algorithms were used in the following years, and eventually systematized in Gagné and Parizeau (2006) paper, which, nonetheless, emphasizes genericity in EC frameworks rather than the application of specific design patterns in them. Ventura et al. (2008) also discuss the use of design patterns in their JCLEC framework, adding to the patterns discussed above, delegation and others.

In general, as is usually mentioned in the papers above, design patterns speed up the development of software frameworks, make programs more readable, and add to their genericity and flexibility by presenting the developer with well-known and program-independent idioms, that can be easily used, understood, and built upon.

`A::E` took advantage of some design patterns in its design. These will be explained below, in no particular order:

- Factory methods The `new` method in the Base modules (such as A::E::Op::Base) act as factory methods. When they are called with a generic description in XML or YAML of a derived object, or simply with the name of the class, they return an object of the derived class.
- The Builder design patterns takes a description of the object that is going to be built and returns an object of that class. Since `A::E` allows serialization in XML and YAML, the `fromXML` methods acts as such; and since YAML can be converted directly to Perl data structures, functions that take a YAML file or string and return a Perl object also act as Builder, for free. The creation of that representation corresponds to the Memento pattern, a pattern used to externalize the representation of an object without violating encapsulation.
- Strategy is used in several places, but mainly in classes for high-level algorithms such as `A::E::Op::Easy` or `A::E::Op::Generation_Skeleton`. These classes implement the most general version of the algorithm, and defer to runtime the instantiation of particulars such as the specific kind of individual they are going to be applied to, the operators that apply on it, and other features such as algorithm termination or the method used for selecting the individuals that are going to mate.
- The Decoration pattern is used to adapt legacy objects to a new interface. The `A::E::Individual::Any` class implements this interface, by wrapping around an object of any class the interface it needs to be used as an `A::E` subject of evolution. The same kind of pattern is applied when `tieing` some kind of individuals whose internal representation is a String so that they can be used as arrays. This allows to create generic operators on arrays that operate on its structure (for instance, to deal with `Bit_Vectors` and `Bit-String` uniformly, independently of its different internal representation).
- The Visitor pattern is used in all the operators, since they act on a particular class (or class hierarchy), changing the object state, but without belonging to that particular class hierarchy. This kind of pattern was probably used for the first time in an evolutionary algorithm library in the GAGS class library (Merelo-Guervós and Prieto 1996) in the shape of *functors*, functions that are encapsulated in an object and that were used for mutation and crossover operators.

However, these patterns by themselves do not contribute to genericity; they must be included in a framework that allows and operates on, at a certain level (notably the algorithm level) generic data structures. Genericity was defined by Gagné and Parizeau (2006) a set of conditions. Let us see below which ones are met by `A::E`:

- *Generic representation*: `A::E` does not put any constraint on the representation used. A few usual representations are provided, but any module can be used via the `Any` class.
- *Generic fitness*: same as above. Any fitness can be used, and a Decoration pattern can be applied to adapt it to `A::E` interface.
- *Generic operations*: operations are only restricted by the data structures they are applied to.
- *Generic evolutionary model*: several models are provided, including Simulated Annealing (Kirkpatrick et al. 1983), but no restriction is placed on how populations of individuals are manipulated.

– *Parameter management* is not handled at the framework level, since Perl has an extensive range of parameter management modules. Every program uses its own parameter management, although they can draw from the XML/YAML serialization facilities of the library.
– Finally, *configurable output* is not addressed either in `A::E` but delegated to each program that uses it. Experiment output is usually done in YAML, which is a generic data serialization language, and in that sense output is generic and configurable, but no facilities in the library are provided to this end.

Summing up, five out of the six genericity criteria are met by `A::E`; the remaining one, parameter management, is probably not so important, and, in any case, it can be included easily in implementations or provided by Perl itself.

4 Implementation and performance issues

In this section we will address several issues related to implementation and performance. First, we will compare `A::E` with libraries written in other languages in Subsect. 4.1; then we will check how a choice of underlying representation affects performance in Subsect. 4.2, all the while taking into account the differences between versions of the Perl virtual machine.

4.1 Comparing `A::E` to other EC libraries

One of the objectives of this subsection is to assess the speed of `A::E` with respect to other libraries, and find out whether the performance gap is as high as mentioned in the other sections, and as applied to general programs. All the programs we have used in this section are available from the `A::E` CVS site at http://fon.gs/ecbench/, and will eventually be included in the standard CPAN distribution (by the time you read this, they probably have already been included).

The first thing we are doing is to match `A::E` with a widely popular evolutionary computation Java library, ECJ (Luke 2009) in a simple problem, MAX-ONES or ONEMAX (with common parameters shown in Table 1), that tries to maximize the number of ones in a binary string. Time results are shown in Table 2.

Since different languages also deal with memory issues in a different way, and use algorithms with a different size complexity, it is also interesting to see how they scale when the size of the chromosome increases. We have measured ECJ and `A::E` on the ONEMAX problem with different chromosome sizes, and results are shown in Fig. 8

Table 1 Common parameters for the ONEMAX problem as solved using `A::E` or ECJ

Parameter	Value
Population	200
Number of bits	128
Maximum number of generations	500
Crossover probability	80%
Mutation probability	20%
Population substituted	20%

The last parameter refers to the percentage of individuals that are substituted every generation in this steady-state algorithm

Table 2 Benchmarking different libraries using the ONEMAX function

Library	Time	
	Mean ± SD	Median
ECJ	1.48 ± 0.22	1.45
`A::E` with Perl 5.8	1.61 ± 0.15	1.59
`A::E` with Perl 5.10	1.36 ± 0.09	1.33
`A::E` with Perl 5.10*	1.29 ± 0.05	1.27

ECJ is version 16 (downloaded June 28th, 2008); `A::E` corresponds to the version in the same date, as downloaded from CVS, released version 0.61, and `java -v` returns `IcedTea Runtime Environment (build 1.7.0-b21)`

The first `A::E` benchmark has been run using `v5.8.8 built for x86_64-linux-thread-multi`, while the second uses `v5.10.0 built for x86_64-linux`, and was compiled from source (as opposed from downloaded from standard Fedora Core repositories), and the third one (marked with *) tweaks compiler options so that maximum performance is obtained

Time was measured using the `time` shell command, which takes into account only user CPU time, not wall-clock time; every program was run 30 times

All differences are significant according to the t-test. Parameters for running the algorithm are described in Table 2

As Fig. 8 shows, `A::E` starts to beat ECJ when the chromosome size increases. This might be due to two different facts: first, the method used to compute the number of ones is more efficient in the Perl case, increasing time with a smaller constant than the one used in ECJ; and second, since the size of objects created is much bigger which makes the Java garbage collecting engine kick, delaying the final result. At any rate, the fact that performance of a Perl evolutionary computation library is competitive with a Java library is once again well established.

We would like to emphasize that the intention of these benchmarks is not to prove either the superiority of Perl over Java, or of this particular library over others; our intention is only to dispel the myth that Perl is so much slower than Java or other languages (C++) that it is not worth the while to spend time in developing code in it,

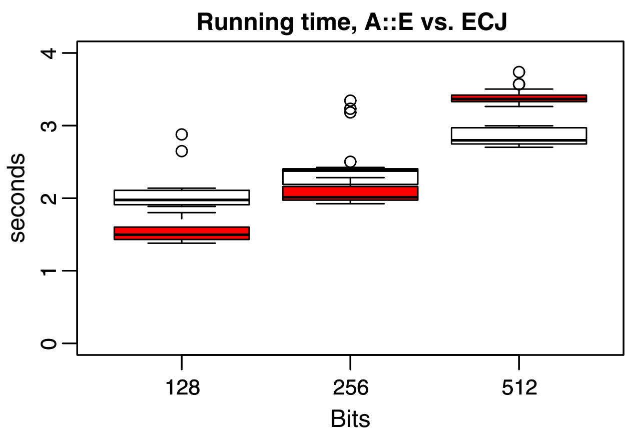

Fig. 8 ECJ (v. 16) versus `A::E` (v 0.66) compared using the latest version of the Java VM and a specially compiled version of Perl 5.11). *Shaded boxes* correspond to ECJ, empty boxes to `A::E`. All data and shell scripts used to compute this can be found in the usual code repository

since, in pure performance terms, anything in Java will be much faster. As is shown in Table 2, when using all tweaks possible, and a state of the art, compiled for optimization, non-threaded version of the Perl interpreter (labeled as 5.10 there), an evolutionary algorithm applied to the ONEMAX problem can run faster in Perl than in Java. You can even obtain a faster version (lowest row) if you tweak with compiler switches when creating the Perl virtual machine, obtaining a custom version of the compiler for the particular machine; results have been shown in the bottom row. In any case, a tweaked version of Java and/or the library or problem will undoubtedly obtain a performance advantage over Perl, but still, it will be worth the while the time spent coding an evolutionary computation problem in Perl, since the wide array of modules and the speed of coding (as shown by Prechelt 2000) more than make up for differences in running time.

Is there a difference between the effort invested in coding a new operator or a whole program in this library and other languages too? Of course there is, but first we will have to take into account that those differences will arise first from the language itself, then from the availability of libraries (or modules) for performing a typical action, and then to the ability of the programmer himself, among other factors. We will use the `sloccount` program, available for most Linux distributions and written by David A. Wheeler. We are comparing `A::E` with the previously mentioned ECJ, JCLEC (Ventura et al. 2008), Watchmaker (Dyer 2008) and Shark (Igel et al. 2008), written in Java and C++. Equivalent files have been chosen to the best of our knowledge, with an emphasis on typical classes which are usually available in all frameworks. Results are shown in Table 3.

In general, the programs written in Perl are much more compact (at least in number of lines of code) than the equivalent in other languages. Even if this is not reflected in the SLOC count, it is also the case that the hierarchies in Java frameworks are deeper than those in Perl. The total number of lines of code of JCLEC and Watchmaker are around 11000 (11,537 for Watchmaker, 10,803 for JCLEC) while `A::E` has 30% less, around 7000; but the comparison is not completely fair, since each package includes different features. ECJ, being much more complex and including many more apps, has three times that account, around 33000 lines of code.

This compactness of code files means that, in general, it is faster (if the only factor in effort is the number of lines of code) to write a new feature in `A::E` (probably, in general in Perl and other scripting languages) than in a strong-typed, compiled language, which has to put more attention to type-safety and error-checking. This also checks with our own experience with other released libraries written in those languages (Keijzer et al. 2002).

4.2 Benchmarking representations

It is interesting also to look at different data structures for representing a bit-string chromosome, and the influence they have in performance; the point is to look at the basic building blocks of an evolutionary algorithm, and how to obtain the best performance for them in this particular context. A program that uses two different bit-string representations, `BitString` and `Bit_Vector` was created, and a program that executes one million bitflips over strings that range from 16 (2^4) to 16384 (2^{14}) was run 30 times. Results are shown in Fig. 9; both available Perl virtual machines (5.8 and 5.10) were also tested.

Figure 9 shows that the best performance is obtained on the 5.10 VM using bit strings as representation; the difference with the worst combination (compact bit vectors with the 5.8 VM) can be around 70%. This hints at the fact that, when performance is an issue, you should try different representations and also test different Perl versions and compile them to optimize throughput.

Next, benchmarks have been run on complete evolutionary algorithms that use different base representations: the *tide* problem using floating point vectors as representation, and the Royal Road (Mitchell et al. 1992) with bitstring representation. This function evaluates a bitstring by blocks, and adds its length to the total fitness only if all bits in the block have the same value. The intention was to obtain a measure of how many *processed chromosomes* (which includes not only the time needed to evaluate them, but also the time invested in applying evolutionary operators at the chromosome and population level) we could

Table 3 Comparison of the number of lines of code in class files among three different frameworks; `A::E` (in its latest released version at the time of writing), ECJ (version 18), JCLEC v. 3 (downloaded from SF during March 2009), Watchmaker (downloaded March 2009, latest version) and Shark 2.2.1

Class	`A::E` 0.66	ECJ	JCLEC	Watchmaker	Shark 2.2.1
Binary	`BitString` 21	`BitVectorIndividual`	`BinArrayIndividual`	N/A	`ChromosomeT_bool`
Chromosome		181	94		99
Binary	`Bitflip`	N/A (included above)	`OneLocusMutator`	`BitStringMutation`	N/A
Mutation	41		31	42	N/A
Binary	`Crossover`	N/A (included above)	`TwoPointsCrossover`	`BitStringCrossover`	N/A
Crossover	35		41	57	
Vector	`Vector`	`IntegerVectorIndividual`	`IntArrayIndividual`	N/A	`ChromosomeT_int`
Chromosome	151	224	94		26
Array	`VectorCrossover`	N/A (included above)	`TwoPointsCrossover`[a]	`IntArrayCrossover`	N/A
Crossover	48		41	56	
Canonical GA	`CanonicalGA`	N/A	`SG`	`SequentialEvolutionEngine`	`CMA`
	56		204	43	248
ONEMAX	`ONEMAX` 26	`MaxOnes`34	N/A	N/A	`countingOnes`40
Roulette Wheel	`RouletteWheel`	`FitProportionateSelection`	`RouletteSelector`	`RouletteWheelSelection`	N/A
Selector	31	38	60	54	

[a] Different from the one above; this one in the `intArray/rec`directory

The number of lines of code has been computed using the language-independent SLOCCount by David A. Wheeler, and includes just the code of the final class, without taking into account included objects or parent classes

The file has been chosen to be, to the best of our knowledge, the closest in function to each other; *N/A* indicates no equivalent file has been found or the concept is not applicable; for instance, in Shark mutation is performed via class methods

File extensions have been suppressed for clarity

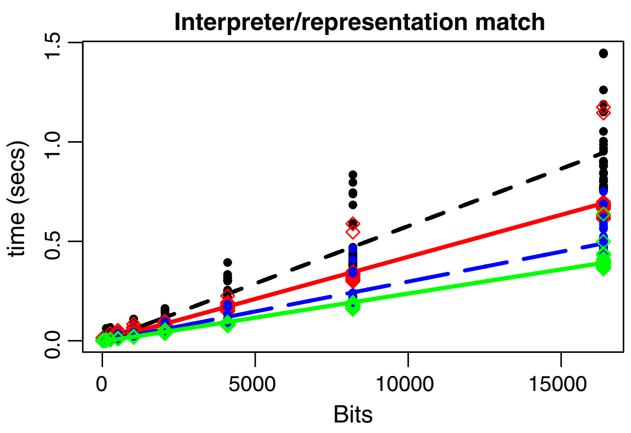

Fig. 9 Comparison (match) between different versions of the Perl interpreter and different internal representations for bitstrings. Raw data is represented by *dots*, while linear fits are drawn using lines. The baseline values (for Perl 5.8) are represented by *dashed lines*, while optimized values (Perl 5.10) use solid lines; bitstrings use *wider lines*, and bitvectors *narrower lines*. Worst result (*narrow*, *dashed*) thus corresponds to 5.8 + bit vector, while the best result (*wide*, *solid*) corresponds to 5.10 + bitstrings. However, if memory occupation is an issue, it would be also interesting to compare memory complexity of both representations. Data for this graph is also available from the CVS server

obtain, and once again to measure the difference between different virtual machines.

The Tide function is defined as follows:

$$T(x,y) = \frac{\sin(x^2+y^2)}{x^2+y^2} \tag{1}$$

but its form does not really matter as much as the fact that it is a floating point function treated as such by the evolutionary algorithm and with a matching data structure to represent it. Population size was set at 100, and the program was run for 500 generations, stopping to measure time after every 100 generations. Results are shown in Fig. 10, and, according to fit, the fastest setup (Perl 5.10) is able to process approximately $5 * 10^5$ chromosomes per second, which is a 20% improvement over older versions of Perl.

Since the previous program used the `Easy` module, another test was done using the more general `FullAlgorithm` module, with standard representation. Results are shown in Fig. 10 (right). In this case population was set to 500, chromosome size at 128, and time values were taken after 100 generations for a total of 500 generations. Once again, 5.10 is the best virtual machine, showing a performance advantage of around 20% over the currently standard (at least in Linux distribution repositories) 5.8. The raw chromosome processing rate is around $2.5 * 10^5$, half that shown before, probably due to the fact that the chromosomes are longer and that a bigger population tips the balance of *evolutionary* processing versus *fitness*

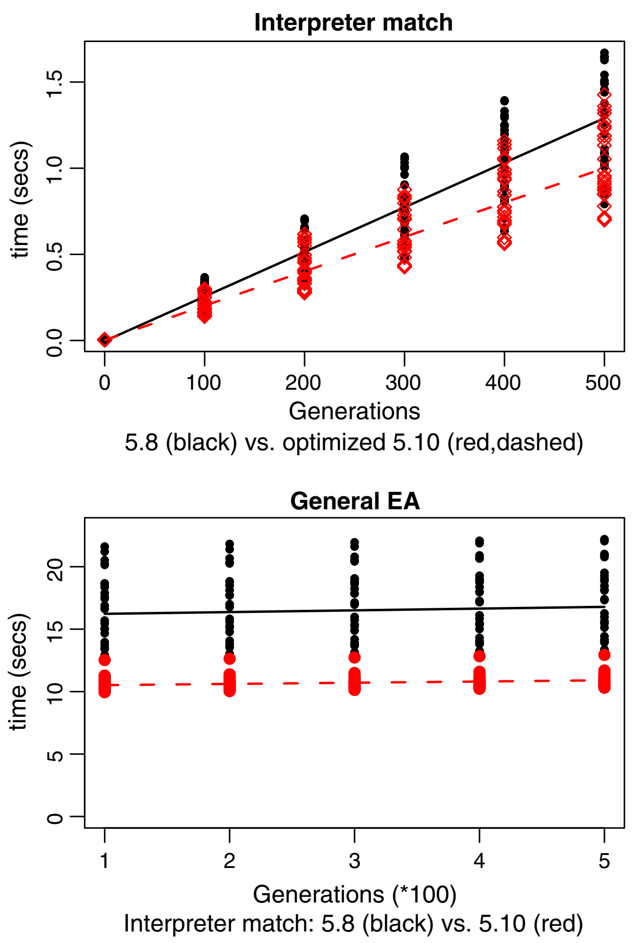

Fig. 10 Raw and average (fitted using a linear model) runtimes for an evolutionary algorithm optimizing the *tide* function using a floating point representation (*top*) and the Royal Road using a binary representation (*bottom*). *Solid black* is used for the 5.8 VM and *dashed red* for 5.10 VM

processing. In any case, both figures are in the same ballpark of tens of thousands of processed chromosomes per second, although that will obviously vary depending on how expensive fitness evaluation is.

5 Creating new operators with `A::E`

Finally, we have also used `A::E` to create a new mutation algorithm and test its effects. Our intention was to take advantage of the capabilities offered by it and the underlying language to create and quickly test it, not so much to check if that operator offers some advantage over traditional mutation operators.

The operator chosen has been called *novelty*, and it sequentially flips a single bit in a chromosome, returning the result only if it has not been evaluated so far. In order to do that, the built-in chromosome/fitness cache featured in some fitness classes was used; the *novelty* operator

Fig. 11 The whole text of the newly created `Novelty_Mutation` operator. `sloccount` by David Wheeler returns 38 lines of code; the rest is documentation

```
use strict; #-*-cperl-*-
use warnings;

package Algorithm::Evolutionary::Op::Novelty_Mutation;

our ($VERSION) = ( '$Revision: 2.4 $ '=~ /(\d+\.\d+)/ );

use Carp;
use Clone::Fast qw(clone);

use base 'Algorithm::Evolutionary::Op::Base';

#Class-wide constants
our $ARITY = 1;

=head2 new( $ref_to_population_hash [,$priority] )

Creates a new mutation operator with an operator application rate
(general for all ops), which defaults to 1, and stores the reference
to population hash.

=cut

sub new {
  my $class = shift;
  my $ref_to_population_hash = shift || croak "No pop hash here, fella!";
  my $rate = shift || 1;

  my $hash = { population_hashref => $ref_to_population_hash };
  my                          $self = Algorithm::Evolutionary::Op::Base::new(
'Algorithm::Evolutionary::Op::Novelty_Mutation', $rate, $hash );
  return $self;
}

=head2 apply( $chromosome )

Applies mutation operator to a "Chromosome", a bitstring, really. Can be
applied only to I<victims> composed of [0,1] atoms, independently of
representation; but
it checks before application that the operand is of type
L<BitString|Algorithm::Evolutionary::Individual::BitString>.

=cut

sub apply ($;$){
  my $self = shift;
  my $arg = shift || croak "No victim here!";
  my $test_clone;
  my $size =  $arg->size();
  for ( my $i = 0; $i < $size; $i++ ) {
    if ( (ref $arg ) =~ /BitString/ ) {
      $test_clone = clone( $arg );
    } else {
      $test_clone = $arg->clone();
    }
    $test_clone->Atom( $i, $test_clone->Atom( $i )?0:1 );
    last  if  !$self->{'_population_hashref'}->{$test_clone->Chrom()}; #Exit  if
not found in the population
  }
  if ( $test_clone->Chrom() eq $arg->Chrom() ) { # Nothing done, zap
    for ( my $i = 0; $i < $size; $i++ ) {
      $test_clone->Atom( $i, (rand(100)>50)?0:1 );
    }
  }
  $test_clone->{'_fitness'} = undef ;
  return $test_clone;
}
```

Fig. 12 Parameter file in YAML used for running experiments on this new operator, which is declared in the 5th line. Using a new operator is just a matter of including its name, parameters needed for its instantiation and priority in a configuration file

```
—
length: 120
pop size: 1024
ops:
   Novelty Mutation:
      - mmdp
      - 1
   Crossover:
      - 2
      - 8
   Bitflip:
      - 1
fitness:
   class: P Peaks
   params:
      - 100
      - 64
max generations: 500
max fitness: 1
selection rate: 0.2
```

(eventually called `A::E` ::Op:: Novelty_Mutation, available from the CVS server and `A::E` from version 0.67) receives a reference to that cache, and newly generated chromosomes are checked against it. If, after flipping all bits in a chromosome none generated a new one, another, random individual was returned.

The whole code (excluding some comments) for this new operator is shown in Fig. 11. For a complex operator such as it is, it has only 38 lines (as computed by `sloccount`, by David Wheeler; many of these lines are scope declarations, which are not compulsory in Perl), and two functions: `new`, to create the new object, and `apply`, to apply the operator to an individual; this individual can be a bitstring, but this operator does not depend on the implementation, since it uses the standard `Atom` function to access each individual bit in the individual. Total development time was a few hours, including the test programs that are compulsory in CPAN distributions.

In a production environment, this is not enough: we had to test the new mutation operator, and also show that it could be done in a short time. Once the operator was included in the library, using it from an existing general program (such as `run-easy-ga.pl`, which is included in the `examples` sub-directory in the distribution) is just a matter of creating a YAML or XML file (YAML, in this case), that describes the algorithm, such as the one shown in Fig. 12 (`ga-novelty-bitflip.yaml`, which can be downloaded from http://fon.gs/ga-novelty/) which describes the values of several parameters, and sets whose ops are going to be used, the name of the variable (`mmdp`) whose cache is going to be used, target fitness, and so on. In any case, it can be set up and run in a matter of minutes. Please note that there is no special provision in the program to use this new class: just declaring it makes it to load the new code and use it; this is an advantage of dynamic languages such as Perl that cannot be matched by compiled languages, which need to have all the code base in advance (although some can also load compiled libraries dynamically through a series of intermediate steps), at least until plugin-based frameworks such as the one described by Wagner et al. (2007) become more commonplace.

Using that and other similar files for the rest of the experiments, we checked it on two well-known functions: the Massively Multimodal Deceptive Problem, and the P-Peaks problem (Giacobini et al. 2006b), two of the fitness functions available as classes in the default library, and measured the number of evaluations needed to reach the optimum, with a cap set to 100,000. Results are shown in Fig. 13.

These two problems are different, and results reached are also different. Apparently, this new operator increases exploration, which results in an improvement in the first problem (probably due to the fact that it avoids inbreeding, thus getting an small edge over the other combinations), but a decrease in performance for the second problem, P-Peaks. This might be due also to the size of the search space. If a significant portion of search space has been already examined, finding new spots using this novelty operator can make exploration more systematic, and thus obtain better results. However, if significant portions have

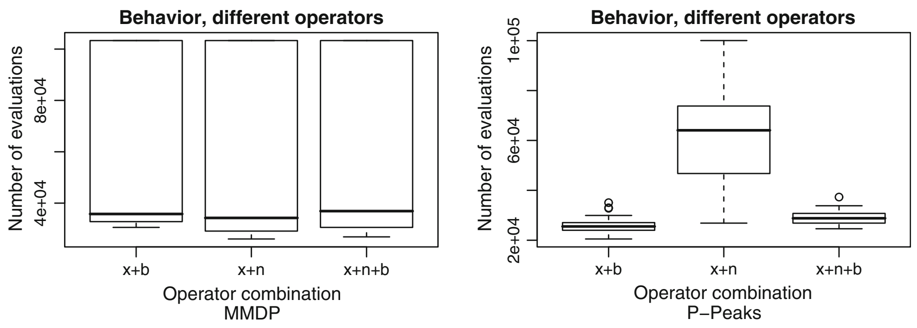

Fig. 13 Number of evaluations until optimum fitness or maximum number of evaluations is reached, for three combinations of operators: crossover (*x*), bitflip mutation (*b*), and *novelty* mutation (*n*) for the MMDP problem (*left*) and P_Peaks (*right*). As it can be seen, crossover + novelty alone reach the best number of evaluations, which is different from the other two in the most difficult problem, MMDP (*left*); however, the best combination for P_Peaks does not include the novelty operator (*right*)

not been explored, *novelty* amounts to flipping one of the first bits of a chromosome, which will not manage to enhance results; and once the population has lost diversity, any bitflip will result in resampling (that is, reevaluating already *known* chromosomes), defaulting to a completely random chromosome, which will be useless.

The whole point of this experiment, however, was not to prove this new operator is better, but the fact that new operators that take full advantage of the capabilities of the framework can be built and tested in a short time, and that it allows to think of genetic operators in new terms, such as this one that uses a cache of known chromosomes. Having a population cache handy, and access to the cache hit rate, opens a new set of possibilities that could be explored in the future.

6 Conclusions and discussion

There is no doubt that algorithms are constrained by the capabilities of the language they are conceived in and the hardware they are written for. Thinking outside a single threaded, sequential machine environment can lead us to design algorithms such as the Evolvable Agents (Laredo et al. 2008a, b), in the same way that a parallel grid-distributed hardware can take us to think of cellular genetic algorithms (Whitley 1993; Alba and Dorronsoro 2008). Thinking in Perl made us think of `A::E`, a Perl module that has been itself evolving for 7 years already, up to the current version (released March 2009).

In this paper we have proved that the performance, at least in a simple problem, is comparable with another evolutionary algorithm library written in Java (ECJ), and that using custom-compiled Perl virtual machines, benchmarking different representations and code profiling, this performance can be enhanced up to 20%. We have also shown the building blocks of an evolutionary algorithm using this library, and how they have already been used in several different application papers. Finally, we have explained how to develop and test a new operator which uses the (obviously not unique, just very handy) capabilities of Perl and the modules included in the library so that development time was quite reduced, and its final form needed only a few dozen lines of code; this operator was tested with mixed results from the algorithmic point of view, but proving its worth at least for a problem.

Since Perl is usually known as a *glue* language, we have used `A::E` in many different environments for many of the research lines we pursue: parallel evolutionary algorithms, hardware optimization, and even assignment of papers to reviewers. Application development is quite fast, quality of results is obviously not affected, and running times are also competitive. The library has been also used to teach implementation of evolutionary algorithms to post graduate students, with good results (which can be observed in the students' class blogs, at http://heliomaster.blogspot.com/ and http://master-ariuca.blogspot.com/).

It is quite clear that it is not worth the while to learn a new language just to program an evolutionary algorithm; most languages already boast EA libraries, some of them with a long and proved history. But it is also true that the most efficient way of programming anything is using the language you feel most comfortable with. Our main intention with `A::E` was to provide Perl programmers with an easy path into evolutionary algorithms that leverages what we (and them) already know about both, and also give language agnostics a library for fast development of evolutionary algorithms with a reasonably fast running time.

There are several improvements we intend to include in the future:

– A version of `A::E` as a POE *component* (available from CPAN at http://fon.gs/pocoae/) has already been created, but it should be enhanced with more parallel and pseudo-parallel execution options.
– Attach a GUI, so that it is easier for newcomers to use the library and execution graphs can be observed online.
– Integrate experiment-analysis scripts as part of the library by standardizing output and using tools to parse and extract statistics from them. An interface to the R statistics package will also be studied.
– Create an standard parallelization tool, using SOAP or other low level protocols such as MPI, in the same way that ParadisEO (Cahon et al. 2004) was initially created as an extension of the EO (Keijzer et al. 2002) evolving objects library.
– Extend it to use Genetic Programming. A very basic GP chromosome is included, but it has not been tested or optimized.
– AJAX interface, to distribute evolutionary algorithms between client and server. Perl is already used in the server in AGAJAJ (Merelo et al. 2008a), an evolutionary algorithm written in Javascript that works natively in browsers, but only as a clearinghouse and transfer method among different clients. Distributing an evolutionary algorithm between different clients and servers will be able to obtain more performance out of the whole setup.
– Link it to other libraries using standard Perl interfaces.

Acknowledgements This paper has been funded in part by the Spanish MICYT projects NoHNES (Spanish Ministerio de Educación y Ciencia - TIN2007-68083) and TIN2008-06491-C04-01 and the Junta de Andalucía P06-TIC-02025 and P07-TIC-03044. We are also grateful to those who have contributed patches and code to `A::E`, too numerous to mention, but fully acknowledged in the general page for the library (found at http://fon.gs/ae-acks/). We are also very grateful to the editor and anonymous reviewers of the first versions of this paper, which have contributed to improve it greatly.

References

Adam K (2008) Microsoft partners with Strawberry Perl to improve the CPAN. http://use.perl.org/~Alias/journal/38036

Alba E, Dorronsoro B (2008) Cellular genetic algorithms. Springer, Berlin

Alba E, Troya J (2001) Gaining new fields of application for OOP: the parallel evolutionary algorithm case. J Object Orient Prog. Web-only version. Available from http://adtmag.com/articles/2001/12/01/gaining-new-fields-of-application -for-oop-the-parallel-evolutionary-algorithm-case.aspx

Alba E, Almeida F, Blesa M, Cotta C, Díaz M, Dorta I, Gabarró J, León C, Luque G, Petit J, et al (2006) Efficient parallel LAN/WAN algorithms for optimization. The MALLBA project. Parallel Comput 32(5–6):415–440

Alcalá-Fdez J, Sánchez L, García S, del Jesus MJ, Ventura S, Garrell JM, Otero J, Romero C, Bacardit J, Rivas VM, Fernández JC, Herrera F (2009) KEEL: A software tool to assess evolutionary algorithms for data mining problems. Soft Comput 13(3):307–318

Araujo L, Merelo-Guervós J, Cotta C, de Vega F (2008) MultiKulti algorithm: migrating the most different genotypes in an Island model. Arxiv preprint arXiv:0806.2843

Ashton E, Hietaniemi J (2006) CPAN frequently asked questions. Tech. rep., Cited 6/10/2006. http://www.cpan.org/misc/cpan-faq.html#What_is_CPAN

Back EC (2008) Ruby vs PHP performance revisited http://elliottback.com/wp/archives/2008/01/17/ruby-vs-php-performance-revisited/

Bäck T, Fogel D, Michalewicz Z (1997) Handbook of evolutionary computation. IOP Publishing Ltd, Bristol

Baiocchi G (2004) Using Perl for statistics: data processing and statistical computing. J Stat Softw 11(1):1–81

Baldi M, Corno F, Rebaudengo M, Reorda MS, Squillero G (2000) Telecommunications optimization: heuristic and adaptive techniques. In: Corne D, Smith G, Oates MJ. chap. GA-based verification of network protocols performance. Wiley, Great Britain, pp 185–198

Bates D (1993) Data manipulation in Perl. Comput Sci Stat Tech. Rep., Available from http://www.ftp.wu-wien.ac.at/pub/lib/info/Data_in_perl.ps.gz

Blansit B (2006) Perl: the duct tape of the internet. J Electron Resour Med Librar 3(2):91

blokhead: A beginning guide to evolutionary algorithms. Available from http://perlmonks.org/index.pl?node_id=298877 (2003)

Box D, Ehnebuske D, Kakivaya G, Layman A, Mendelsohn N, Nielsen H, Thatte S, Winer D (2000) Simple object access protocol (SOAP) 1.1. Available from http://www.w3.org/TR/soap/

Cahon S, Melab N, Talbi E (2004) ParadisEO: a framework for the reusable design of parallel and distributed metaheuristics. J Heuristics 10(3):357–380

Castillo P, Arenas M, Castellano J, Merelo-Guervós JJ, Rivas V, Romero G (2002) Optimisation of multilayer perceptrons using a distributed evolutionary algorithm with SOAP. In: Guervós JJM, Adamidis P, Beyer HG, nas JLFV, Schwefel HP (eds) Parallel problem solving from nature—PPSN VII, 7th international conference, Granada, September 7–11, 2002. Proceedings, no. 2439 in Lecture Notes in Computer Science, LNCS. Springer, pp 676–688

Castillo P, Merelo J, Arenas M, Romero G (2007) Comparing evolutionary hybrid systems for design and optimization of multilayer perceptron structure along training parameters. Inform Sci 177:2884–2905. doi: 10.1016/j.ins.2007.02.021

Castillo P, Fernández G, Mora A, Merelo J, Bernier J, Prieto A (2008) Evolving machine microprograms. In: Genetic and evolutionary computation conference, GECCO2008. ACM Press, New York, pp 1103–1104

Dyer D (2008) Watchmaker framework for evolutionary computation - version 0.5.0. Available from http://blog.uncommons.org/2008/12/10/watchmaker-framework-for-evolution ary-computation-version-050/

Eiben AE, Smith JE (2003) Introduction to evolutionary computing. Springer, Berlin

Eiben A, Van Kemenade C, Kok J (1995) Orgy in the computer: multi-parent reproduction in genetic algorithms. In: Proceedings of the 3rd european conference on artificial life, no. 929 in LNAI. Springer, pp 934–945

Fallside DC (2004) XML Schema Part 0: Primer. Available from http://www.w3.org/TR/xmlschema-0/

Fourment M, Gillings M (2008) A comparison of common programming languages used in bioinformatics. BMC Bioinform 9(1):82

Gagné C, Parizeau M (2006) Genericity in evolutionary computation software tools: principles and case-study. Int J Artif Intell Tools 15(2):173–194

Gamma E, Helm R, Johnson R, Vlissides J (1993) Design patterns: abstraction and reuse of object-oriented design. In: ECOOP'93, object-oriented programming, 7th european conference, Kaiserslautern, July 26–30, 1993: Proceedings, Lecture Notes in Computer Science, vol 707. Springer, pp 406–431

García-Nieto J, Alba E, Chicano F (2007) Using metaheuristic algorithms remotely via ROS. In: GECCO '07: Proceedings of the 9th annual conference on genetic and evolutionary computation. ACM, New York, pp 1510–1510. doi: http://doi.acm.org/10.1145/1276958.1277239

Giacobini M, Preuss M, Tomassini M (2006a) Effects of scale-free and small-world topologies on binary coded self-adaptive CEA. In: Evolutionary computation in combinatorial optimization: 6th european conference, EvoCOP 2006, Budapest, April 10–12, Proceedings. Springer

Giacobini M, Preuss M, Tomassini M (2006b) Effects of scale-free and small-world topologies on binary coded self-adaptive CEA. In: Gottlieb J, Raidl GR (eds) Evolutionary computation in combinatorial optimization—EvoCOP 2006, LNCS, vol 3906. Springer, Budapest, pp 85–96

gumpu: Genetic Programming or breeding Perls. Available from http://perlmonks.org/index.pl?node_id=31147 (2001)

Harold ER (2001) XML Bible. IDG Books worldwide

Hugman J (2003) mybeasties: an object ecosystem. Available from http://sourceforge.net/projects/mybeasties

Igel C, Heidrich-Meisner V, Glasmachers T (2008) Shark. J Mach Learn Res 9:993–996. http://www.scopus.com

Ingerson B, Evans C, Ben-Kiki O (2001) YAML 1.2 specification. Available from http://yaml.org/spec/1.2/

jweed: improving evolutionary algorithm. Available from http://perlmonks.org/index.pl?node_id=330315 (2004)

Keijzer M, Merelo-Guervós JJ, Romero G, Schoenauer M (2002) Evolving objects: a general purpose evolutionary computation library. In: Collet P, Fonlupt C, Hao JK, Lutton E, Schoenauer M (eds) Artificial evolution, 5th international conference, evolution artificielle, EA 2001, Le Creusot, France, October 29–31, 2001, Selected Papers, Lecture Notes in Computer Science, vol 2310. Springer, pp 231–244

Kirkpatrick S, Gelatt C, Vecchi M (1983) Optimization by simulated annealing. Science 220(4598):671–680

Klein J, Spector L (2007) Unwitting distributed genetic programming via asynchronous JavaScript and XML. In: Proceedings of the 9th annual conference on genetic and evolutionary computation. ACM, New York, pp 1628–1635

Koza JR (1992) Genetic programming: on the programming of computers by means of natural selection. MIT Press, Cambridge

Kunken J (2001) The application of genetic algorithms in English vocabulary generation. In: Proceedings of the twelfth midwest artificial intelligence and cognitive science conference 2001. Miami University Press. Available also from http://www.ocf.berkeley.edu/~jkunken/glot-bot/

Laredo JLJ, Eiben AE, Steen MV, Castillo PA, Mora AM, Merelo JJ (2008a) P2P evolutionary algorithms: a suitable approach for tackling large instances in hard optimization problems. In: 14th international euro-par conference, Las Palmas de Gran Canaria, Spain, August 26–29, 2008. Proceedings, pp 622–631. doi: 10.1007/978-3-540-85451-7. http://dx.doi.org/10.1007/978-3-540-85451-7_66

Laredo JLJ, Eiben AE, Steen MV, Merelo JJ (2008b) On the run-time dynamics of a peer-to-peer evolutionary algorithm. In: Rudolph G, Jansen T, Lucas S, Poloni C, Beume N (eds) Parallel problem solving from nature - PPSN X, *LNCS*, vol 5199. Springer, Dortmund, pp 236–245. doi: 10.1007/978-3-540-87700-4_24

Lee W, Kim H (2005) Genetic algorithm implementation in python. In: Proceedings—fourth annual ACIS international conference on computer and information science, ICIS 2005, vol 2005, pp 8–12

Lenaerts T, Manderick B (1998) Building a genetic programming framework: the added-value of design patterns. In: Genetic programming: first european workshop, EuroGP'98: Paris, April 14–15, 1998: Proceedings. Springer, pp 196–208

Loui RP (2008) In praise of scripting: real programming pragmatism. IEEE Comput 41(7):22–26

Lukasz S (2009) AI::Genetic::Pro-efficient genetic algorithms for professional purpose. Available from http://search.cpan.org/~strzelec/AI-Genetic-Pro-0.34/lib/AI/Genetic/Pro.pm

Luke S, et al (2009) ECJ: a java-based evolutionary computation and genetic programming research system. Available at http://www.cs.umd.edu/projects/plus/ec/ecj

MacCallum R (2003) Introducing a Perl genetic programming system–and can meta-evolution solve the bloat problem. In: Genetic programming, 6th european conference, EuroGP. Springer, pp 364–373

McPhee NF, Hopper NJ, Reierson ML (1998) Sutherland: an extensible object-oriented software framework for evolutionary computation. In: Koza JR, Banzhaf W, Chellapilla K, Deb K, Dorigo M, Fogel DB, Garzon MH, Goldberg DE, Iba H, Riolo R (eds) Genetic programming 1998: proceedings of the third annual conference, pp 22–25 Jul. Morgan Kaufmann, University of Wisconsin, Madison. http://www.mrs.umn.edu/~mcphee/Research/Sutherland/sutherland _gp98_announcement.ps.gz

Merelo JJ, Castillo PA, Arenas MG, Romero G (2003) Specifying evolutionary algorithms in XML. In: Mira J, Álvarez (eds) 7th international work-conference on artificial and natural neural networks, IWANN 2003, number 2686–2687 in Lecture Notes in Computer Science, LNCS. pp 503–509. Springer. http://www.springerlink.com/link.asp?id=v6j34778lx0y43m0

Merelo JJ, García AM, Laredo JLJ, Lupión J, Tricas F (2007) Browser-based distributed evolutionary computation: performance and scaling behavior. In: GECCO '07: proceedings of the 2007 GECCO conference companion on genetic and evolutionary computation, pp 2851–2858. ACM Press, New York. doi: http://doi.acm.org/10.1145/1274000.1274083

Merelo JJ, Castillo PA, Laredo JLJ, Mora A, Prieto A (2008a) Asynchronous distributed genetic algorithms with Javascript and JSON. In: WCCI 2008 Proceedings, pp 1372–1379. IEEE Press. http://atc.ugr.es/I+D+i/congresos/2008/CEC_2008_1372.pdf

Merelo JJ, Mora AM, Castillo PA, Laredo JLJ, Araujo L, Sharman KC, Esparcia-Alcázar AI, Alfaro-Cid E, Cotta C (2008b) Testing the intermediate disturbance hypothesis: effect of asynchronous population incorporation on multi-deme evolutionary algorithms. In: Rudolph G, Jansen T, Lucas S, Poloni C, Beume N (eds) Parallel problem solving from nature-PPSN X, LNCS, vol 5199, pp 266–275. Springer, Dortmund. doi: 10.1007/978-3-540-87700-4_27

Merelo-Guervós JJ (2002) Evolutionary computation in Perl. In: Perl Mongers M (ed) YAPC::Europe::2002, pp 2–22

Merelo-Guervós JJ, Castillo-Valdivieso P (2004) Conference paper assignment using a combined greedy/evolutionary algorithm. In: Yao X, Burke E, Lozano JA, Smith J, Guervós JJM, Bullinaria JA, Rowe J, Tino P, Kabán A, Schwefel HP (eds) Parallel problem solving from nature—PPSN VIII, no. 3242 in Lecture Notes in Computer Science,LNCS, pp 602–611. Springer

Merelo-Guervós JJ, Arenas MG, Carpio J, Castillo P, Rivas VM, Romero G, Schoenauer M (2000) Evolving objects. In: Wang PP (ed) Proc. JCIS 2000 (joint conference on information sciences), vol I, pp 1083–1086. ISBN: 0-9643456-9-2

Merelo-Guervós JJ, Prieto A (1996) GAGS, a flexible object-oriented library for evolutionary computation. In: Borrajo D, Isasi P (eds) MALFO96, Procs. of the first international workshop on machine learning, forecasting and optimization, pp 99–105. Available from http://neo.lcc.uma.es/EAWebSite/SKELETON/GPROG/gags-paper.ps.gz

Michalewicz Z (1996) Genetic Algorithms + Data Structures = Evolution programs, 3rd edn. Springer, Berlin

Mitchell M, Forrest S, Holland J (1992) The royal road for genetic algorithms: fitness landscapes and GA performance. In: Towards a practice of autonomous systems: proceedings of the first european conference on artificial life, pp 245–254

Murray B, Williams K (1999) Genetic algorithms with Perl. The Perl Journal 5(1). Also available from http://mathforum.org/~ken/genetic/article.html

Networks O (2009) XML.com: XML from the inside out. Web site at http://www.xml.com

Neylon MK (2001) Algorithm :: Genetic. Available from http://perlmonks.org/index.pl?node_id=81678

Ousterhout JK (1998) Scripting: higher level programming for the 21st century. Computer 31(3):23–30

Pedersen T, Patwardhan S, Michelizzi J (2004) WordNet:: similarity-measuring the relatedness of concepts. In: Proceedings of the nineteenth national conference on artificial intelligence (AAAI-04) pp 1024–1025

Prechelt L (2000) An empirical comparison of seven programming languages. IEEE Comput 33(10):23–29. doi: http://doi.ieeecomputersociety.org/10.1109/2.876288

Ray ET (2001) Learning XML: creating self-describing data. O 'Reilly, Beijing

Raymond E (1998) Book review: the essential Perl books. Linux Journal 46es. Available from http://www.linuxjournal.com/article/2523

regeya: Why YAML? Why not? http://www.kuro5hin.org/story/2004/10/29/14225/062 (2004)

Schwartz RL, Phoenix T Foy BD (2008) Learning Perl, 5th edn. O 'Reilly & Associates, Sebastopol, CA, USA

Smith MA (1998) Building software frameworks for evolutionary computation. In: Porto VW, Saravanan N, Waagen DE, Eiben AE (eds) Evolutionary programming, Lecture Notes in Computer Science, vol 1447. Springer, pp 557–567

Stajich J, Block D, Boulez K, Brenner S, Chervitz S, Dagdigian C, Fuellen G, Gilbert J, Korf I, Lapp H, et al (2002) The bioperl toolkit: Perl modules for the life sciences. Genome Res 12(10):1611–1618. doi: 10.1101/gr.361602

Veenhuis C, Franke K, Kōppen M (2000) A semantic model for evolutionary computation. In: Proceedings IIZUKA ’00, pp 68–73

Ventura S, Romero C, Zafra A, Delgado J, Hervás C (2008) JCLEC: a java framework for evolutionary computation. Soft Comput 12(4):381–392

Wagner S, Winkler S, Pitzer E, Kronberger G, Beham A, Braune R, Affenzeller M (2007) Benefits of plugin-based heuristic optimization software systems. In: Proceedings computer aided systems theory EUROCAST 2007, Lecture Notes in Computer Science, vol 4739. Springer, p 747

Wainwright P, Calpini A, Corliss A, Cozens S, Powers S, Merelo-Guervós J, Saraf A, Nandor C (2001) Professional Perl programming. Wrox Press Inc. Available from http://www.amazon.com/exec/obidos/ASIN/1861004494 or http://www.amazon.co.uk/exec/obidos/ASIN/1861004494

Walker J (2005) Floating point benchmark: Comparing languages. http://www.fourmilab.ch/fourmilog/archives/2005-08/000567.html

Wall L, Christiansen T, Orwant J (2000) Programming Perl, 3rd edn. O ’Reilly & Associates

Whitley L (1993) Cellular genetic algorithms. In: Proceedings of the 5th international conference on genetic algorithms table of contents. Morgan Kaufmann Publishers Inc. San Francisco

Whitley D (1994) A genetic algorithm tutorial. Stat Comput 4(2):65–85

Wilson GC, McIntyre A, Heywood MI (2004) Resource review: three open source systems for evolving programs—Lilgp, ECJ and grammatical evolution. Genetic Program Evol Mach 5(1):103–105

Zlatanov T (2001) Cultured Perl:genetic algorithms applied with Perl create your own darwinian breeding grounds. Available from http://www106.ibm.com/developerworks/linux/library/lgenperl/