OPEAL: Evolutionary algorithms in Perl

Juan Julián Merelo Guervós[1], Francisco Javier García Castellano[1], Pedro Castillo Valdivieso[1], and Maribel García Arenas[1]

GeNeura Team, Depto. Arquitectura y Tecnología de Computadores
Escuela Técnica Superior de Ingeniería Informática
C/ Daniel Saucedo Aranda, s/n
Universidad de Granada (Spain)
tutti@geneura.ugr.es, http://geneura.ugr.es

Abstract. This paper describes OPEAL, a module for implementing evolutionary computation problems in the Perl programming language. It is shown how it leverages Perl features, and how it has been applied to several problems, dealing mainly with interfacing with text files, the world wide web and databases, in the context of the organization of an evolutionary computation conference.

1 Introduction

In this paper we will try to make a introduction to the OPEAL (*original Perl evolutionary algorithm library*) module, a set of classes for writing evolutionary algorithms in Perl.

Perl might not seem the right language for doing evolutionary computation, since it is an interpreted language, and EC applications usually require high performance (mainly in the fitness evaluation stage). However, Perl has another quality that makes it ideal for EC (and, obviously, many other applications): it is ideal as glue, that is, as a way of interfacing many different things, protocols, applications, file formats. Perl features an exhaustive collection of modules (which correspond roughly to *libraries* in other languages) for interfacing with databases, the world wide web, handling complex data structures, and even writing Perl programs in Latin (Lingua::Romana::Perligata, written by Damien Conway). Other features make it outstanding among other languages: the quality of the references available for it, specially the *Camel Book* [1], the *Llama book* [2], and others such as [3]; the quality of its technical support, with highly knowledgeable user communities in almost any country and in any human language; and, finally, the portability of its implementation: 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 (a class that includes all machine/OS combinations with a decent C compiler).

In the case of evolutionary algorithms, we were mainly interested in handling XML (eXtensible Markup Language) documents [4–6]. One of the objectives of OPEAL was to be able to describe an evolutionary algorithm using XML;

having programs that parse the XML file and run a genetic algorithm, 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 describes Evolutionary Algorithms has been called EvoSpec, and its specification by a DTD (Data Type Dictionary) or XSchema (equivalent way of describing an XML dialect using XML itself) [7]. Every class in OPEAL has an alternative syntax in XML, and all classes have a constructor that creates an object from its XML description.

The rest of the paper is organized as follows: next section (section 2) shows the state of the art in Perl evolutionary algorithmss. Right next (section 3) describes the OPEAL evolutionary computation library, its design decisions, and how it can be used. Section 4 a simple application of this library generation of *easily recalled* passwords, and, briefly, other applications such as paper assignment. The paper finishes with some discussion, conclusions and future work.

2 State of the art

As of may 2002, there is no comprehensive Perl evolutionary algorithm module uploaded to CPAN (http://www.cpan.org, the site for all Perl modules), or available on the WWW; and by comprehensive we mean multiparadigm and easily extensible. Most published modules deal with genetic programming in Perl. Since Perl is an interpreted language, it is very easy to evaluate Perl expressions from a Perl program (or script). The first (as claimed by the author) paper published on the subject seems to be one by Baldi et al. [8], 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 evolutonary algorithms: Kunken [9] recently describes an application that evolves words that "look like they were English", or *fake english* words, by trying to evolve them using the same letter patterns than english uses. One of the application presented in this paper tries to be a reimplementation of this method, but using a different method to score the "englishness" of a word. The same application is also mentioned by Zlatanov in [10], who implements a genetic programming system, with source code available, to solve the same problem.

There are several papers about doing genetic programming in Perl: the first one was written by Murray and Williams [11], which, despite its title, actually describes a genetic programming system, similar to another mentioned in the PerlMonks site ([12]; http://www.perlmonks.org is a meeting place for practitioners). Another module that implements a canonical genetic algorithm was released [13], but it cannot be easily extended or adapted to new paradigms.

None of the systems mentioned so far wholly use Perl's capabilities to implement an object-oriented library, easily adaptable and expandable, which have been two of the objectives OPEAL's designers had in mind.

On the other hand, XML and evolutionary algorithms realms have remained traditionally apart from each other, but there have been a few researchers that have approached the topic. Among them, we would like to highlight the EAML

language (Evolutionary Algorithm Markup Language, an XML dialect) described by Veenhuis and others [14]. 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 evolutonary algorithm, but it does not really achieve its target, since, for instance, variation operators are tags, instead of being attributes values of a generic *operator* tag; this means that adding a new operator (say, a n-ary *orgy* operator) would require a redesign of the language's syntax (defined through a DTD, or Data Type Dictionnary). Tag 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, for a restricted form of a evolutionary algorithm it is a valid approach, and it is a step in the right direction; EvoSpec, which is used by OPEAL, tries to overcome these errors. There are also other languages for evolutionary algorithm description, such as EASEA [15]; however, this is a language designed to generate C++ and Java programs from its description, and is not intended as an universal evolutionary algorithm description language that can be parsed from any other language, or generate programs in any language. The main objectives of this merging of XML and evolutionary computation were achieved with the first release of the library, and are described elsewhere [**?**]. This paper focuses in the evolutionary computation aspects of OPEAL, which have been developed since the publication of that paper.

3 Description of OPEAL, an evolutionary algorithms Perl library

OPEAL follows a design philosophy very close to that of EO (Evolving Objects, [16]), that is, flexibility to evolve any data structure, separation of operators from individuals, expandability to any new data structure and operators needed to evolve them, and independence of operators applied to groups of individuals from the structure of the individuals themselves. This same philosophy has been applied to JEO, the evolutionary algorithm library within DREAM ([17], additional material on DREAM available from http://world-wide-dream.org).

The elements of the OPEAL framework are distributed into two broad classes: individuals and operators. Individuals are data structures that can be potentially evolved, and operators are functions that are applied to other elements to change them; these elements can be individuals as well as groups of individuals, or any other data structure that includes individuals (lists, arrays, sets). All individuals subclass the IndiBase base class, while all operators subclass dhe OpBase base class. All elements are serialized in XML, and have an alternative XML representation; each element knows how to read and write itself from and to XML. Individuals use the `indi` tag while operators use the `op` tag. The IndiBase tag is the base class of the individual hierarchy, that includes classes for strings StringIndi (with bitstrings, BinaryIndi as a subclass of this one). It also includes VectorIndi, for vectors of any kind (`float` or `int`, for instante) and GPIndi, for tree indi-

viduals, used for Genetic Programming. The operator hierarchy, with OpBase as base class, includes binary or unary variation operators applied to individuals, such as mutation (Mutation) or crossover (Crossover), selection, insertion, and even generations of a whole algorithm and the algorithm itself. Putting them all in the same hierarchy makes them have an uniform application interface, and easens up serialization in XML. So far, a canonical evolutionary algorithm (which can be applied to any kind of individual), a more flexible EasyAlgorithm (which admits different selection operators and termination conditions) and simulated annealing (SimAnn) have been implemented.

These two class hierarchies pave the way for extending OPEAL. New individuals should go to its hierarchy, extending IndiBase or one of its siblings, and new operators should do the same with OpBase. Writing them is easy enough, following the pattern of the existing one; the most complicated part is the XML serialization, but it is not big deal either; just a matter of writing things between tags and reading them using Perl XML modules, or plain regular expressions. The only real requirement is that new operators have the apply method, which is used to make the operator apply itself to any number of individuals, and that new individuals include the fitness method, which is used to set or get fitness.

The flexibility of OPEAL is shown by integrating several kinds of "chromosomes", or data structures that can be evolved: trees, vectors and strings (which include as a particular case bitstrings). These data structures can be modified by using generic operators (for instance, crossover is generic for any data structure that can be accessed randomly), or specific operators (mutation is usually specific: bitflip mutation can only be applied to bitstrings, while gaussian random mutation applies to number vectors).

Fitness functions are handled as references-to-subroutines (similar to a function pointer in C, or a function reference in C++), which can be used as any other variable within the Perl program. This way, the fitness function is really a variable and can be called from the object that evaluates all the population, generally the evolutionary algorithm object (such as the above mentioned EasyAlgorithm). In some cases, this is the only thing the user will have to do: inserting part of a Perl evaluation function within an XML file.

Finally, OPEAL is freely distributed under the GPL licence, and is available from http://opeal.sourceforge.net via FTP, HTTP and CVS; latest version is 0.4, released in May 2002. This 0.4 version numbering reflects the fact that it's still at beta stage. In the near future, it will also be made available through CPAN, the Perl module repository network.

4 Results

In order to show how easy it is to program non-trivial problems using this library, fake english word generation was chosen. Generating words that look as if they were written in a given language might seem as a lame thing to do, but, in fact, it has got many different applications. For starters, it can be used to generate easy-to-remember passwords. A *fake english* word such as *inenth* or *itheen*

(which are actually words generated by the program) are much easier to remember for an english-speaking person than *rq5tmh* (which is an actual, randomly generated, password used for a PPSN submitter). This obviously saves on customer support calls (or emails), even in the case of PPSN, when there are only a few "customers". Other possible uses are related to computational linguistics: generate fake texts or fake poetry; use frequencies to score words to be used in Scrabble-like games, and apply it to cryptoanalysis, by generating known plaintext with frequencies similar to those used in real language. Some people have also suggested generating names for firms and products in a particular language (or ina mixture of two languages), and, curiously enough, generation of names for characters and places in role playing games. In fact, two of the applications of evolutionary algorithms in Perl [10, 9], unknowing of each other, are exactly that: word generation.

The basis for word generation is the following: each language has a characteristic sound, that depends on how letters are intertwined in, or put at the beginning, or end, of words. Basque (euskera) language has different letter, two-letter, and 3-letter combination frequency than other latin-alphabet languages such as romanian, or tagalog (spoken in Philippines). Analyzing these frequencies, and scoring words depending on them, will accordingly generate words that "sound" as if they belonged to a particular language; since the words have a familiar sound, they should be easy to remember. The length of a word is also characteristic of a language: some languages, like finnish and german, favor longer words, while english have slightly shorter words.

These frequencies and length were then used to evaluate words, by using the following formula:

$$F(I) = \frac{L(I)}{2} + \sum_{n=1,2,3} F(n, I) \tag{1}$$

where $F(I)$ = fitness for individual I; $L(I)$ is score due to length, that is, frequency for that language for that particular length; and $F(n, I)$ is score due to n-gram frequency, that is, the n-gram frequency averaged over number of n-grams in the word; n-grams include also beginning and ending of words. The frequencies for the english language were extracted from the web edition of the *Guardian* newspaper, and thus are actual frequencies, not dictionnary frequencies. Frequencies are normalized so that the most frequent item has an 1 score; all frequencies are relative to that one; for instance most frequent length for english is 3. In order to make length frequency less important in the evolution, it is halved. This analysis yields word scores such as the ones shown in table 1 (on the left).

The parameters and operators used in the algorithm are shown in table 1 (right). Several combinations of 4 variation operators were tested. For starters, **mutation**, **random length-increment** and **crossover** were tested; then, a **permutation** operator was added. In a first set of experiments, all operators were applied with the same application rate; then, the crossover rate was increased so that it was applied as many times as all the rest together. Each com-

Word	Score
Two	0.58
One	0.90
With	0.44
Eskarrikasko	0.68
Guardian	0.66
No	0.99
Yes	0.85
Gibraltar	0.65
The	0.65

Parameter	Value
Population Size	100
Termination Condition	15 generations w/o change
String length range	5-12
Population renewal rate	40%
Selection method	Roulette wheel

Table 1. On the **left**, frequency and length based fitness (*englishness*) for some words. Different words have different fitness, but some non-English words such as *eskarrikasko* has lower fitness than some english words such as *two* or *the*, but higher than others (with); there's not a clear-cut distinction among english and non-english words. On the **right hand side**, parameters used for the evolutionary algorithm. Variation operator application rate is changes, and is explained on the text.

bination was run 10 times, and average fitness, average number of evaluations and standard deviations were computed. Results are shown in table 2.

Operator rates	Fitness	Generations
Mut, XOver, Inc 1/3	1.07 ± 0.20	41 ± 16
Perm, Mut, XOver, Inc 1/4	1.08 ± 0.35	37 ± 23
Perm, Mut, Inc 1/6; XOver 1/2	1.05 ± 0.26	37 ± 17
Perm, Mut, Inc 1/8; XOver 5/8	1.01 ± 0.20	34 ± 14

Table 2. Results of the evolutionary algorithms for password generation. Different operator rates and combinations were tested; the fraction indicates the rate of new individuals generated using that operator. Results are averages over 10 runs. Average fitness is more or less the same for all combinations, since differences are not significative. However, the number of generations that it takes to find the optimum is lower when crossover is applied in the higher proportion.

The conclusion to be drawn from this experiment is that it was quite easy to program this application with OPEAL, and interface it with the current PPSN management scripts, which were also written in Perl; each password was generated in almost real time (on average, each run took less than 2 seconds on a Duron 700 machine), and meaningless, but familiar-sounding words were used, reducing technical support time. The whole source code for this application is available from the authors, and will be released through the OPEAL site soon. A slightly different version can also be run online from the address http://geneura.ugr.es/Mason/EvolveWordsPPSN.html.

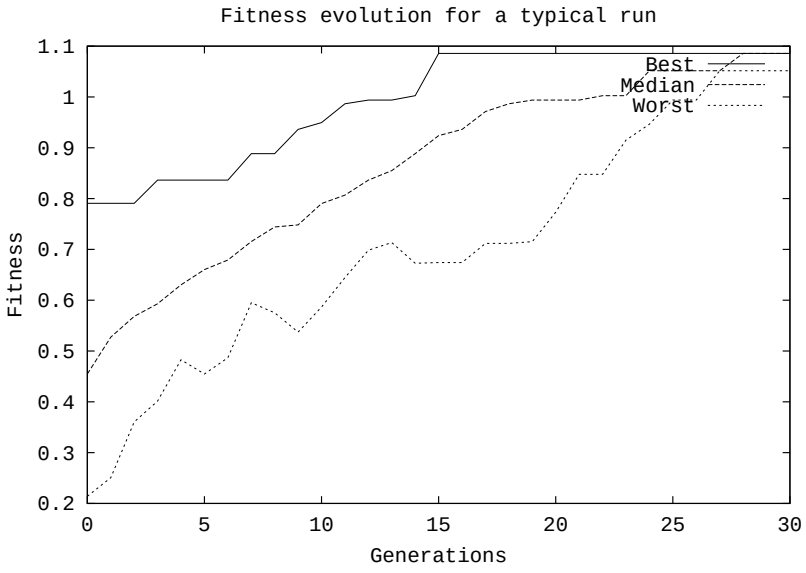

Fig. 1. Fitness plot for the fake english word problem, which shows how some words with a 1.1 fitness (higher than usual english words) are obtained.

In the context of the PPSN2002 conference, OPEAL was also used for assigning papers to referees. The fact that it was programmed in Perl made very easy to interface with the database in which paper and referee data was stored.

So far (May 2002), the program files have been downloaded around 500 times by users all over the world, and it is widely used within the department for several evolutionary-computation related projects. One of the most interesting project is using SOAP (Simple Object Access Protocol) for a ring-shaped parallel evolutionary computation experiment [18, 19] (bot papers are in Spanish). In the latter paper, several EC populations are placed at different nodes with a ring topology. Members of the population are only sent to the next node in the ring, and received from the previous node. Experiments with a few nodes have been performed, which show that it scales up very well, without a huge increase of the network load, and allows efficient parallel computation with heterogeneous nodes.

5 Conclusion, discussion

This paper presents how a Perl evolutionary algorithm library called OPEAL (or Algorithm::Evolutionary in the future), has been applied to two different task within the organization of the PPSN 2002 conference. One of then had no major impact on the conference itself, since it was only a evolutionary algorithm for

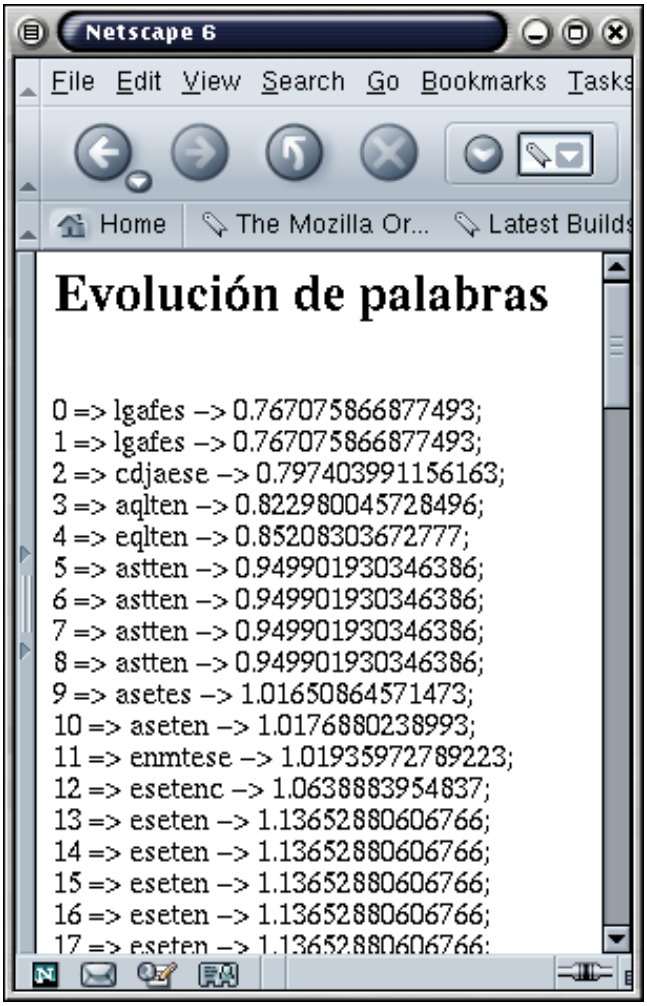

Fig. 2. Running the "evolve words" problem online (from http://geneura.ugr.es/Mason/EvolveWords.html), which is but a sample of the possibilities of running evolutionary algorithms in Perl: in this case, the EA code is embedded in the web page, and combined with the HTML::Mason module to generate the web page that is seen in the browser. In this case, the best word in every generation is shown, together with its fitness. In this case, the words evolved used frequencies for the Spanish language.

generation of passwords for chairpersons and referees; this problem, however, allowed us to check the easiness of programming with OPEAL and interfacing with the current PPSN application. At the same time, this application advanced a bit the state of the art in fake-english word generation, a subject that has lately received some attention; in this case, crossover seemed to have a major role in speeding up password generation.

Future work will mainly concentrate on improving the OPEAL library; preparing it for inclusion in the CPAN repository, and adding new paradigms, such as evolution strategies, and testing extensively genetic programming on this new platform. It will also concentrate on fixing the standards for XML specification of evolutionary algorithms.

Acknowledgements

This work has been supported in part by INTAS 97-30950. I am also grateful to the contributors to a discussion on the spanish open source forum http://barrapunto.com about possible uses for a word generation algorithm.

References

1. Larry Wall, Tom Christiansen, and Jon Orwant. *Programming Perl.* O´Reilly & Associates, 3rd edition, 2000.
2. Randal L. Schwartz and Tom Phoenix. *Learning Perl.* O´Reilly & Associates, 3rd edition, 2001.
3. Peter Wainwright; Aldo Calpini; Arthur Corliss; Simon Cozens; Shelley Powers; JJ Merelo-Guervós; Aalhad Saraf; Chris Nandor. *Professional Perl Programming.* Wrox Press Inc, 2001.
4. Elliotte Rusty Harold. *XML Bible.* IDG Books worldwide, 1991.
5. O´Reilly Networks. XML.com: XML from the inside out. Web site at `http://www.xml.com`.
6. Erik T. Ray. *Learning XML: creating self-describing data.* O´Reilly, January 2001.
7. David C. Fallside. Xml schema part 0: Primer. Available from `http://www.w3.org/TR/xmlschema-0/`.
8. M. Baldi, F. Corno, M. Rebaudengo, M. Sonza Reorda, and G. Squillero. *Telecommunications Optimization: Heuristic and Adaptive Computation Techniques by David Corne (Editor) - George Smith - Martin J. Oates*, chapter GA-Based Verification of Network Protocols Performance. Wiley, 2000.
9. Joshua Kunken. 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 2001, 2001. Available also from http://www.ocf.berkeley.edu/~jkunken/glot-bot/.
10. Teodor Zlatanov. Cultured Perl : Genetic algorithms applied with Perl create your own darwinian breeding grounds. Available from http://www-106.ibm.com/developerworks/linux/library/lgenperl/, August 2001.
11. Brad Murray and Ken Williams. Genetic algorithms with perl. *The Perl Journal*, 5(1), Fall 1999. Also available from http://mathforum.org/~ken/genetic/article.html.

12. gumpu. Genetic programming or breeding perls. Available from http://perlmonks.org/index.pl?node_id=31147, September 2001.
13. Michael K. Neylon. Algorithm::Genetic. Available from http://perlmonks.org/index.pl?node_id=81678, May 2001.
14. Christian Veenhuis, Katrin Franke, and Mario Köppen. A semantic model for evolutionary computation. In *Proceedings IIZUKA*, 2000.
15. P. Collet, E. Lutton, M. Schoenauer, and J. Louchet. Take it EASEA. In Marc Schoenauer, Kalyanmoy Deb, Guenter Rudolph, Xin Yao, and Hans-Paul Schwefel Evelyne Lutton, Juan Julian Merelo, editors, *PPSN VI*, number 1917 in LNCS, pages 891–901. Springer Verlag, 2001.
16. J. J. Merelo. OPEAL, una librería de algoritmos evolutivos en perl. In Sánchez [?], pages 54–59.
17. M. Keijzer; J. J. Merelo; G. Romero; and M. Schoenauer. Evolving objects: a general purpose evolutionary computation library. Springer-Verlag, October 2001.
18. M.G. Arenas, L. Foucart, J.J. Merelo, and P. A. Castillo. Jeo: a framework for evolving objects in java. In *Actas Jornadas de Paralelismo* [20].
19. J. J. Merelo, J.G. Castellano, P.A. Castillo, and G. Romero. Algoritmos genéticos distribuidos usando soap. In *Actas Jornadas de Paralelismo* [20].
20. Merelo; Castellano; Castillo. Algoritmos evolutivos P2P usando SOAP. In Sánchez [?], pages 31–37.
21. *Actas XII Jornadas de Paralelismo*. Universidad Politécnica de Valencia, 2001.
22. E. Alba; F. Fernández; J. A. Gómez; F. Herrera; J. I. Hidalgo; J. J. Merelo; J. M. Sánchez, editor. *Actas primer congreso español algoritmos evolutivos, AEB02*. Universidad de Extremadura, Febrero 2002.