Ranking the Performance of Compiled and Interpreted Languages in Genetic Algorithms

Juan-Julián Merelo-Guervós[1], Israel Blancas-Álvarez[1], Pedro A. Castillo[1], Gustavo Romero[1], Pablo García-Sánchez[1], Victor M. Rivas[2], Mario García-Valdez[3], Amaury Hernández-Águila[3] and Mario Román[4]

[1]*CITIC and Computer Architecture and Technology Department, University of Granada, Granada, Spain*

[2]*Department of Computer Sciences, University of Jaén, Jaén, Spain*

[3]*Tijuana Institute of Technology, Tijuana, Mexico*

[4]*University of Granada, Granada, Spain*

Keywords: Benchmarking, Implementation of Evolutionary Algorithms, OneMax, Genetic Operators, Programming Languages, Performance Measurements.

Abstract: Despite the existence and popularity of many new and classical computer languages, the evolutionary algorithm community has mostly exploited a few popular ones, avoiding them, especially if they are not compiled, under the asumption that compiled languages are *always* faster than interpreted languages. Wide-ranging performance analyses of implementation of evolutionary algorithms are usually focused on algorithmic implementation details and data structures, but these are usually limited to specific languages. In this paper we measure the execution speed of three common operations in genetic algorithms in many popular and emerging computer languages using different data structures and implementation alternatives, with several objectives: create a ranking for these operations, compare relative speeds taking into account different chromosome sizes and data structures, and dispel or show evidence for several hypotheses that underlie most popular evolutionary algorithm libraries and applications. We find that there is indeed basis to consider compiled languages, such as Java, faster in a general sense, but there are other languages, including interpreted ones, that can hold its ground against them.

1 INTRODUCTION

In the same spirit of the *No Free Lunch* theorem (Wolpert and Macready, 1997) we could consider there is a *no fast lunch* (Merelo et al., 2015) hypothesis for the implementation of evolutionary optimization problems, in the sense that, while there are particular languages that might be the fastest for particular problem sizes and specially fitness functions there is no single language that is the fastest for all chromosome sizes, implementations and fitness functions. But the main problem is that implementation decisions, and in many occasions reviewer reports, are based on common beliefs such as thinking that a particular language is the fastest or other language is too slow to even being taken into consideration for implementing evolutionary algorithms.

After initial tests on a smaller number of languages (Merelo et al., 2016) and three different data structures, in this paper we add more languages, different data structures and also, in the case of languages with a particularly bad result, new implementations including some made using released evolutionary algorithm libraries, and finally even found and corrected some bugs.

To use the results we already had, we have re-used the same operations: crossover, mutation and OneMax. In general (Merelo-Guervós et al., 2011) an Evolutionary Algorithm (EA) application will spend the most time running the fitness function and others, such as ranking the population; however, these are well covered by several general purpose benchmarks so they are not the focus of this paper. A priori, this result would extend only to some implementations of genetic algorithms. However, in this paper we would like to present not only the result itself, which is interesting, but also a methodology to first assess new languages for implementing evolutionary algorithms

Merelo-Guervós, J., Blancas-Álvarez, I., Castillo, P., Romero, G., García-Sánchez, P., Rivas, V., García-Valdez, M., Hernández-Águila, A. and Román, M.
Ranking the Performance of Compiled and Interpreted Languages in Genetic Algorithms.
DOI: 10.5220/0006048101640170
In *Proceedings of the 8th International Joint Conference on Computational Intelligence (IJCCI 2016) - Volume 1: ECTA*, pages 164-170
ISBN: 978-989-758-201-1
Copyright © 2016 by SCITEPRESS – Science and Technology Publications, Lda. All rights reserved

for the value or insights they might give to the algorithm mechanism, and second to make real-world measures and benchmark them to test their speed and performance relative to other common languages instead of choosing usual languages based only on past experience and common (maybe mis-) conceptions.

The rest of the paper is organized as follows: coming up next in Section 2, we will present the state of the art of the analysis of EA implementations. Next we will present in Section 3 the tests we have used in this paper and its rationale along with the languages we have chosen for carrying them out. Finally, in Section 4 we will present the results obtained. Finally, we will draw the conclusions and present future lines of work.

2 STATE OF THE ART

The first published benchmarks of evolutionary algorithms (Jose Filho et al., 1994) focused on implementation details using C and C++; since then, there are not many publications on the subject, until recently when Alba et al. examined the performance of different data structures, all of them using the same language, in (Alba et al., 2007). (Merelo-Guervós et al., 2010) described the implementation of an evolutionary algorithm in Perl, also used in this paper, proving that, as a whole, Perl could run evolutionary algorithms almost as fast as Java, but if we took into consideration other factors like actual coding speed measured by single lines of code, Perl was better.

Most papers, if not all, including this one, focus on single-threaded procedural environments; for instance, a recent paper (Nesmachnow et al., 2015) focuses on a single language, C++. In these cases *classical* languages have a certain advantage. However, innovation has not only spawned new languages, but also new architectures like the Kappa architecture (Erb and Kargl, 2015), microservices (Namiot and Sneps-Sneppe, 2014) or service-oriented frameworks (García-Sánchez et al., 2010; García-Sánchez et al., 2013) where we could envision that different parts of an evolutionary algorithm might be written in different languages, but also in new languages better suited for certain tasks. The *no fast lunch* principle enunciated above implies that different languages could be used for distributed systems, with the fastest or most appropriate language used for every part of it.

All in all, existing literature focuses either on a single language and different data structures or different, and mainly popular, languages with a single data structure. In previously published research (Merelo et al., 2016) we focused on fewer languages and mainly tried to measure their scaling behaviour across different lengths, and we found that Java, C# and C obtained the best results. However, we did not attempt to rank languages or measure relative speeds across all tests. This is what we do in this paper, extending the analysis of the previous paper with more languages and implementations. Next we will explain how the experiment was set up and the functions used in it.

3 EXPERIMENTAL SETUP

We have used bitflip mutation, crossover and count-ones or OneMax in this and the previous experiments, mainly since they are the quintessential genetic algorithm operators and a benchmark used in practical as well as theoretical approaches to Genetic Algorithms (GAs). In general, they exercise only a small part of the language capabilities, involving mainly integer and memory-access performance through loops. However, the implementation of these operations is deceptively simple, specially for OneMax, a problem frequently used in programming job interviews.

Loops are also a key component in performance. Most languages allow `for` loops, however, many can also perform *map* implicit loops where a function is run over each component of a data structure, reducing sequential or random access to arrays. When available, we have used these types of functions. This implies that despite its simplicity, the results have wider applicability, except for floating point performance, which is not tested, mainly because it is a major component of many fitness functions, not so much of the evolutionary algorithm itself.

Chromosomes in EAs can be represented in several different ways: an array or vector of Boolean values, or any other scalar value that can be assimilated to it, or as a bitstring using generally "1" for true values or "0" for false values. Different data structures will have an impact on the result, since the operations that are applied to them are, in many cases, completely different and thus the underlying implementation is more or less efficient. Besides, languages use different native data structures to represent this information. In general, it can be divided into three different fields:

- *Strings*: representing a set bit by 1 and unset by 0, it is a data structure present in all languages and simple to use in most.
- *Vector of Boolean values*: not all languages have a specific primitive type for the Boolean false and true values; for those who have, sometimes they

have specific implementations that make this data structure the most efficient. In some and when they boolean values were not available, 1 or 0 were used. *Bitsets* are a special case, using bits packed into bytes for representing vector of bits, with 32 bits packed into a single 4 byte data structure and bigger number of bytes used as needed.

- *Lists* are accessed only sequentially, although running loops over them might be more efficient that using random-access methods such as the ones above.

Besides, many languages, including functional ones, differentiate between Mutable and Constant data structures, with different internal representations assigned to every one of them, and extensive optimizations used in Immutable or constant data structures. Immutable data structures are mainly used in functional languages, but some scripting languages like Ruby or Python use it for strings too.

In this paper more than 20 languages, some of them with several implementations, have been chosen for performing all benchmarks; additionally, another language, Rust, has been tested for one of them, and Clojure using persistent vectors was tested only for OneMax. This list includes 9 languages from the top 10 in the TIOBE index (TIOBE team, 2016), with Visual Basic the only one missing, and 1 more out of the top 20 (or two if we include Octave instead of the proprietary application Matlab). The list of languages and alternative implementations is shown in Table 1.

When available, open source implementations of the operators and OneMax were used. In all cases except in Scala, implementation took less than one hour and was inspired by the initial implementation made in Perl or in Lua. In fact, we abandoned languages such as AWK or FORTRAN when it took too long to make a proper implementation in them. Adequate data and control structures were used for running the application, which applies mutation to a single generated chromosome a hundred thousand times. The length of the mutated string starts at 16 and is doubled until reaching 2^{15}, that is, 32768. This upper length was chosen to have an ample range, but also so small as to be able to run the benchmarks within one hour. Results are shown next. In some cases and when the whole test took less than one hour, length was taken up to 2^{16}.

In most cases, and especially in the ones where no implementation was readily available, we wrote small programs with very little overhead that called the functions directly. That means that using classes, function-call chains, and other artifacts, will add an overhead to the benchmark; besides, this implies that the implementation is not exactly the same for all languages. However, this inequality reflects what would be available for anyone implementing an evolutionary algorithm and, when we think it might have an influence on the final result, we will note it.

Every program used also provides native capabilities for measuring time, using system calls to check the time before and after operations were performed. These facilities used the maximum resolution available, which in some cases, namely Pascal, was somewhat inadequate.

All programs produced the same output, a comma separated set of values that includes the language and data structure used, operand length and time in seconds.

4 RESULTS AND ANALYSIS

All the results have been made available in the repository that holds this paper as well as some of the implementations, at https://git.io/bPPSN16. Implementation and results are available with a free license. The Linux system we have used for testing runs the `3.13.0-34-generic #60-Ubuntu SMP` kernel on an `Intel(R) Core(TM) i7-4770 CPU @ 3.40GHz` CPU. In this paper we will look mainly at the aggregated results, comparing the differences in order of magnitude between the different languages and also how they compare to each other.

To have a general idea of performance and be able to compare across benchmarks and sizes, we have used the language Julia, whose performance is more or less in the middle, as a baseline for comparison and expressed all times as the ratio between the time needed for a particular language and length and the speed for Julia. Ratios higher than 1 mean that the particular language+data structure is faster than Julia, <1 the opposite. Please check the boxplot in Figure 1 for the average and standard deviation across all functions and lengths, when the comparison is possible.

We should first refer to the remarkable performance of Clojure using persistent vectors. Clojure is a functional language, and immutable structures lend themselves better to functional processing. Clojure includes some specific functions for this kind of counting. However, it should be noted that then we could not run Clojure on the rest of the functions, bitflip and crossover.

Next comes Java, with a performance that is around two orders of magnitude better than baseline. In some cases, Clojure using mutable vectors can be very fast too, but its overall performance takes it to the third worst overall performance. This also implies that simply choosing a language is not a guarantee of

Table 1: Languages, versions, URLs, data structure and type of language used to carry out the benchmarks. No special flags were used for the interpreter or compiler.

Language	Version	URL	Data structures	Type
C	4.8.2	http://git.io/v8T57	Bit String	Compiled
C++	4.8.4	http://git.io/v8T57	Bit Vector	Compiled
C#	mono 4.2	https://git.io/vzHDI	Bit Vector	Compiled
Clojure	1.8.0	https://git.io/vzHDe	Bit Vector	Compiled
Common Lisp	0.13.7	https://git.io/vzHyR	Simple Bit Vector	Compiled
Go	go1.2.1	http://git.io/vBSYp	Bit Vector	Compiled
Haskell	ghc 7.10.3	https://git.io/vzHMw	Mutable Vector	Compiled
Java	1.8.0_66	http://git.io/v8TdR	Bitset	Compiled
JavaScript	node.js 5.0.0	http://git.io/vBSYd	String	Interpreted
Julia	0.2.1	http://git.io/vBSOe	Bit Vector	Interpreted
Lua	5.2.3	http://git.io/vBSY7	String	Interpreted
Octave	3.8.1	http://git.io/v8T57	BitVector	Interpreted
PHP	5.5.9	http://git.io/v8k9g	String	Interpreted
Perl	v5.20.0	http://git.io/bperl	String, Bit Vector	Interpreted
Python	2.7.3	http://git.io/vBSYb	String	Interpreted
Python 3	3.4.3	https://git.io/p3deap	Bit Vector, List	Interpreted
Rust	1.4.0	https://git.io/EOr	Bit Vector	Compiled
Scala	2.11.7	http://git.io/vBSYH	String, Bit Vector	Compiled
Ruby	1.9.3p551	https://git.io/rEO	Bit Vector	Interpreted
JRuby	9.0.5.0 (2.2.3)	https://git.io/rEO	Bit Vector	Interpreted
Free Pascal	2.6.2-8	https://git.io/fpeo	Bit Vector	Compiled
Kotlin	1.0.1	https://git.io/kEO	Bit Vector	Compiled
Dart	1.15.0	https://git.io/dEO	List	Interpreted

performance; data structures chosen and implementation play also a major role.

The top 5 is completed with C#, Haskell and Scala, this last using also a particular implementation, BitVector; if a more *natural* BitString is used, its performance is on a par with the aforementioned Clojure. Out of the top 5, three of them are functional languages: Haskell, Clojure and Scala; Clojure is also an interpreted language, with a JIT compiler that targets the Java Virtual Machine. Three of them also use the Java Virtual Machine: Clojure, Java and Scala; C# has its own runtime too, with Haskell having a compiler to native object code, being thus the only *pure* compiled language in this set.

The biggest differences reside among the top 3 languages. Haskell, Scala, Go, C and Perl have a very similar performance, and it should be remarked that Perl is the first interpreted language in the list. Other compiled languages, including C++, have worse performance, and the worst 10 include several compiled languages, as well as functional languages such as clisp. Also Perl in a different implementation, showing once again that a particular language, by itself, need not be a guarantee of performance.

From the best to the worst implementation, there are 4 orders of magnitude of difference. This can be quite important because the range of running time between the slowest and the fastest can go from 1 second to several hours. Languages such as Octave or clisp, maybe even Python and Julia, should be avoided except for problems with a size that allows them to be run in a few seconds.

However, let us rank the languages looking at how they fare, comparing with the other implementations. The averaged ranking shown in Figure 2 averages the position reached for all sizes and functions, and subtracts it from the total number of languages tested, so that bigger is better. Even if on average and constrained to OneMax Clojure is better than Java, this language beats it more times than the contrary, so it becomes first in the ranking. C also achieves a better position than in the previous graph, and Perl a worse position. Two emerging languages which have been added for this paper, Kotlin and Dart, together with JRuby, the implementation of Ruby for the JVM, close the top 10 ranking. You need to look at this averaged rankings as the probability that a particular language is the best for a particular function and size. Languages such as Pascal or Go are more likely to win than JRuby or node. However, some groupings can also be detected. The three best (Java, Clojure and C#) form the first, with a second group of three, up to Free Pascal, and then a gap to #7, which is Haskell. The worst 10, in this case, include two compiled lan-

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

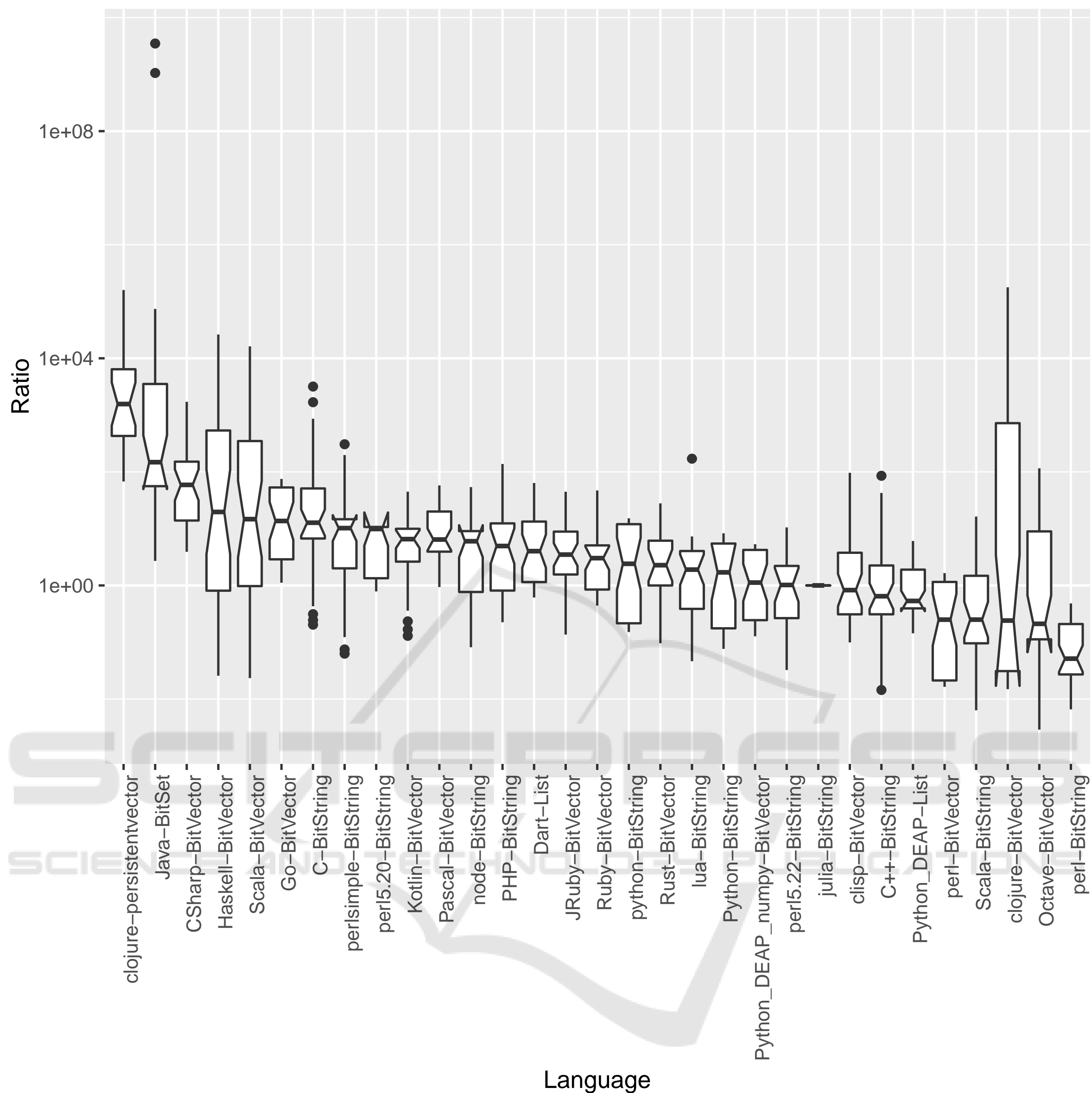

Figure 1: Boxplot of scaled performance compared to baseline Julia. Please note that *y* has a logarithmic scale. The strings indicate the language and the implementation; for instance, Python_DEAP_numpy is a python implementation using the operators in the DEAP framework (Fortin et al., 2012) and `numpy` implementation for vectors.

guages: C++ and Scala with a BitString implementation. The rest are all interpreted languages.
Another interesting result that can be observed in this ranking is the domination of the Bit Vector structure over the rest, bearing in mind that the BitSet used by Java is actually a type of bit vector with a particular implementation. The best bit string implementation is C, followed by Perl, which is actually very close. The best List implementation, by the Dart language is in the middle region by performance and the 9th by ranking. Let us discuss these findings in the next section, together with the conclusions.

5 CONCLUSIONS

In this paper we have measured the performance of an extensive collection of languages in simple and common evolutionary algorithm operations: mutation, crossover and OneMax, with the objective of finding out which languages are faster at these operations and what are the actual differences across lan-

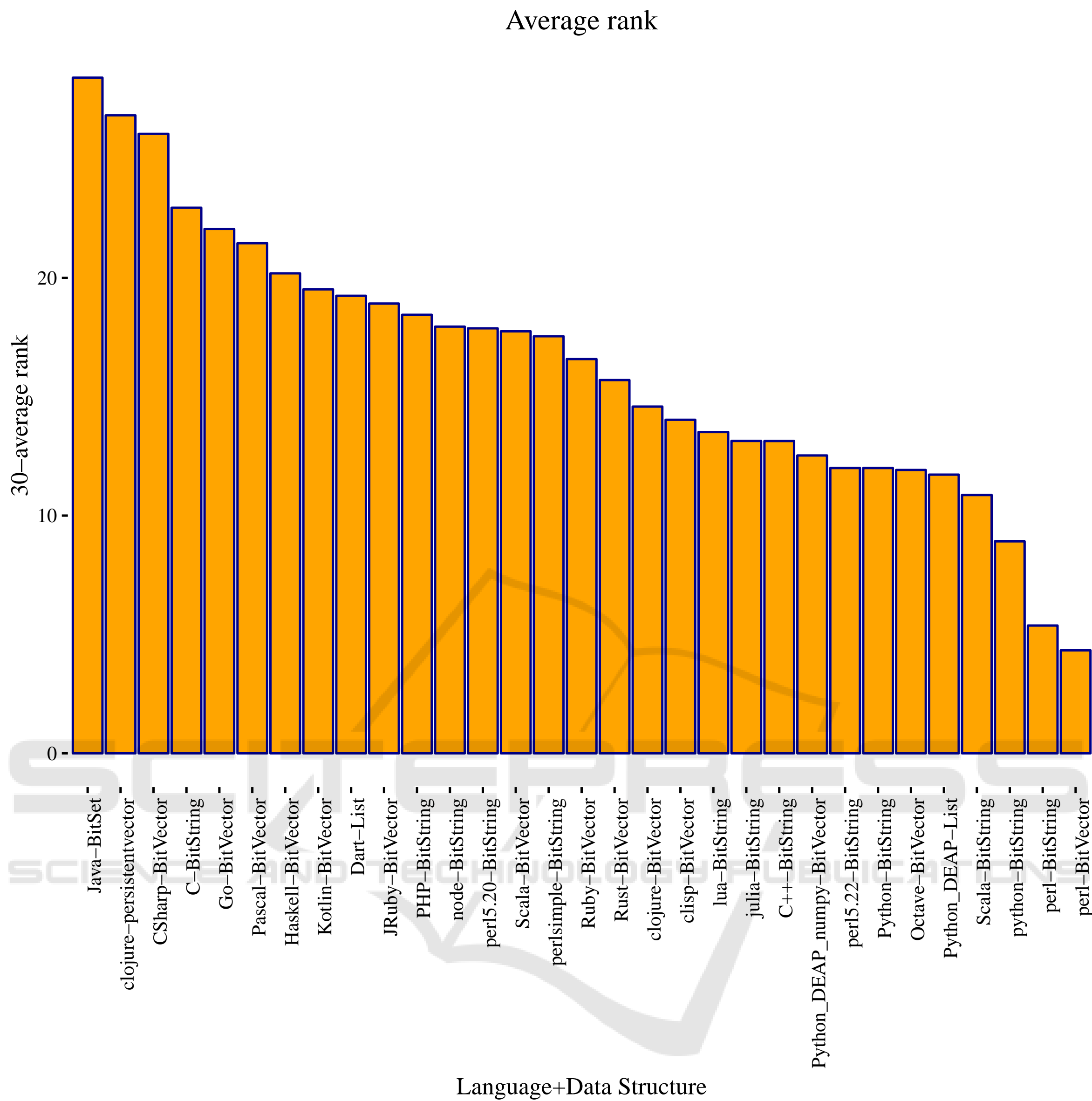

Figure 2: Ranking averaged across all measures, subtracted from the total number of measured languages, so bigger is better.

guages, language types and data structures, the main objective being that the EA practitioner can use these results to make decisions over which language or languages to use when implementing evolutionary languages.

We can conclude that the language usually considered the fastest among the practitioners, Java, indeed holds that position on average, although it can be beaten by Clojure in some functions. Functional languages have a remarkable performance, despite their lack of popularity in the EA community. Using vector of bits rather than bit strings or lists provides, in general, a better performance, and immutable data structures, such as the ones used in functional languages can be accessed and used, in general, faster than mutable structures if available in the same language.

Future lines of work might include a more extensive measurement of other operators such as tournament selection and other selection algorithms. A priori, these are essentially CPU integer operations and their behavior might be, in principle, very similar to the one shown in these operations. It would also be interesting to mix and match different languages, choosing every one for its performance, in a hybrid architecture. Communication might have some overhead, but it might be offset by performance. Combining

some compiled languages such as Go or C with others characterized by its speed in some string operations, like Perl or programming ease, like Python, might result in the best of both worlds: performance and rapid prototyping. Creating a whole multi-language framework along these lines is a challenge that might be interesting in the future.

Besides, in some cases the languages have not been used to their full potential. Concurrent languages such as Scala or Go are actually used sequentially, missing features that a priori would make them stand out over languages not designed with that feature, such as Java.

The full set of languages and tests will also be made available as a Docker container, which can be downloaded easily to run it in particular environments and machines.

ACKNOWLEDGEMENTS

This paper is part of the open science effort at the university of Granada. It has been written using `knitr`, and its source as well as the data used to create it can be downloaded from the GitHub repository[1] https://github.com/geneura-papers/2016-ea-languages-PPSN/. It has been supported in part by GeNeura Team[2], projects TIN2014-56494-C4-3-P (Spanish Ministry of Economy and Competitiveness), Conacyt Project PROINNOVA-220590.

REFERENCES

Alba, E., Ferretti, E., and Molina, J. M. (2007). The influence of data implementation in the performance of evolutionary algorithms. In *Computer Aided Systems Theory–EUROCAST 2007*, pages 764–771. Springer.

Erb, B. and Kargl, F. (2015). A conceptual model for event-sourced graph computing. In *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems*, DEBS '15, pages 352–355, New York, NY, USA. ACM.

Fortin, F.-A., Rainville, D., Gardner, M.-A. G., Parizeau, M., Gagné, C., et al. (2012). Deap: Evolutionary algorithms made easy. *The Journal of Machine Learning Research*, 13(1):2171–2175.

García-Sánchez, P., González, J., Castillo, P., Merelo, J., Mora, A., Laredo, J., and Arenas, M. (2010). A Distributed Service Oriented Framework for Metaheuristics Using a Public Standard. In *Nature Inspired Cooperative Strategies for Optimization (NICSO 2010)*, pages 211–222. Springer.

[1]https://github.com/JJ/2016-ea-languages-PPSN

[2]http://geneura.wordpress.com

García-Sánchez, P., González, J., Castillo, P.-A., García-Arenas, M., and Merelo-Guervós, J.-J. (2013). Service oriented evolutionary algorithms. *Soft Comput.*, 17(6):1059–1075.

Jose Filho, L. R., Treleaven, P. C., and Alippi, C. (1994). Genetic-algorithm programming environments. *Computer*, 27(6):28–43.

Merelo, J. J., Castillo, P. A., Blancas, I., Romero, G., García-Sánchez, P., Fernández-Ares, A., Rivas, V. M., and Valdez, M. G. (2016). Benchmarking languages for evolutionary algorithms. In Squillero, G. and Burelli, P., editors, *Applications of Evolutionary Computation - 19th European Conference, EvoApplications 2016, Porto, Portugal, March 30 - April 1, 2016, Proceedings, Part II*, volume 9598 of *Lecture Notes in Computer Science*, pages 27–41. Springer.

Merelo, J.-J., García-Sánchez, P., García-Valdez, M., and Blancas, I. (2015). There is no fast lunch: an examination of the running speed of evolutionary algorithms in several languages. *ArXiv e-prints*.

Merelo-Guervós, J.-J., Romero, G., García-Arenas, M., Castillo, P. A., Mora, A.-M., and Jiménez-Laredo, J.-L. (2011). Implementation matters: Programming best practices for evolutionary algorithms. In Cabestany, J., Rojas, I., and Caparrós, G. J., editors, *IWANN (2)*, volume 6692 of *Lecture Notes in Computer Science*, pages 333–340. Springer.

Merelo-Guervós, J.-J., Castillo, P.-A., and Alba, E. (2010). `Algorithm::Evolutionary`, a flexible Perl module for evolutionary computation. *Soft Computing*, 14(10):1091–1109. Accesible at http://sl.ugr.es/000K.

Namiot, D. and Sneps-Sneppe, M. (2014). On micro-services architecture. *International Journal of Open Information Technologies*, 2(9):24–27.

Nesmachnow, S., Luna, F., and Alba, E. (2015). An empirical time analysis of evolutionary algorithms as c programs. *Software: Practice and Experience*, 45(1):111–142.

TIOBE team (2016). Tiobe index for april 2016. Technical report, TIOBE.

Wolpert, D. H. and Macready, W. G. (1997). No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 1(1):67–82.