Hindawi Publishing Corporation
Advances in Operations Research
Volume 2011, Article ID 154516, 14 pages
doi:10.1155/2011/154516

Research Article

Computer Language Effciency via Data Envelopment Analysis

Andrea Ellero and Paola Pellegrini

Department of Management, Università Ca' Foscari Venezia, Cannaregio 873, 30121 Venezia, Italy

Correspondence should be addressed to Andrea Ellero, ellero@unive.it

Received 8 April 2011; Revised 9 September 2011; Accepted 9 September 2011

Academic Editor: Ching-Jong Liao

Copyright © 2011 A. Ellero and P. Pellegrini. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

The selection of the computer language to adopt is usually driven by intuition and expertise, since it is very diffcult to compare languages taking into account all their characteristics. In this paper, we analyze the effciency of programming languages through Data Envelopment Analysis. We collected the input data from *The Computer Language Benchmarks Game*: we consider a large set of languages in terms of computational time, memory usage, and source code size. Various benchmark problems are tackled. We analyze the results first of all considering programming languages individually. Then, we evaluate families of them sharing some characteristics, for example, being compiled or interpreted.

1. Introduction

Which programming language is the best one? In general, the answer could be "no guess," of course. But if we add a precise criterion, asking, for example, for a program with low memory consumption, one would probably not choose Java. In this paper, we propose a way for understanding how to widen this answer, suggesting which programming languages should *not* be chosen after considering a set of different criteria as a whole.

Several languages are available to researchers and practitioners, with different syntax and semantics. As common practice, an implementer chooses the programming language to use according to his feeling and knowledge. In fact, comparing the performance of the various languages is not straightforward. Only few studies deal with such an evaluation [1–5]. Nonetheless, getting a deeper understanding of the language efficiency may be of great help for exploiting the available resources. Two problems are often connected to these analysis. On the one hand, the ability of who writes the code may bias the results. On the other hand, several criteria must be considered in the evaluation, and their relevance cannot be easily defined *a priori*.

The first problem can be in some part overcome by considering the data provided by *The Computer Language Benchmarks Game* (CLBG). They are available on the website http://shootout.alioth.debian.org/gp4/. The CLBG collects source codes produced by several programmers. Implementers volunteer their contributions. They are allowed to choose the language to use and the problem to tackle in a set of about twenty benchmarks. For each of these benchmarks, different workloads are tested. The collected programs are written in about forty very popular programming languages.

The benchmark problems do not require the investigation of a solution space for finding an optimum. As an example, one of these problems requires reading integers from a file and printing the sum of those integers. Hence, all programs reach the same result, and they are tested for computational time, memory usage, and source code size.

Only the best programs are kept for each pair of language and benchmark. If two implementations using the same language exist for a problem, they are evaluated in terms of the three just-mentioned criteria on multiple workloads. If one of them performs worse in terms of at least one criterion and not better in the others, then it is dominated and thus deleted from the program list. This way, the survived codes are the ones that behave the best, or that, at least, are not clearly worse than any other.

The selection operated in the CLBG site, nonetheless, appears quite conservative. It is not of great help for discriminating among languages. In fact, we would like to compare languages considering run time, memory usage, and source code size as a whole, but these characteristics can be neither directly compared, nor *a priori* ordered in a relevance scale. Each single measure may give rise to a different ranking of languages.

A possible way to obtain a single ranking, undistorted by a subjective bias, is through the application of the Data Envelopment Analysis (DEA) [6]. In the literature, DEA has been widely applied in a huge number of contexts, in order to assess the relative efficiency of economic units, called Decision Making Units (DMUs) in DEA's jargon. In our application, each of the languages considered in the CLBG site is assumed to be a DMU.

DEA allows to take into account different inputs and outputs (attributes) of the economic units, creating a unique ranking. Its characteristic of coping with attributes of different nature turns out to be very useful also in the present research. In our case, we consider computational time, memory usage, and source code size as input to the DMU/language while we consider the quantities of programs contributed for solving each workload as outputs.

The philosophy underpinning DEA is to assess each unit using individually optimized attribute weights. The weights are those that lead to the best possible efficiency value for that specific unit. In our context, this means, for example, that the less memory consuming language will have maximal weight associated to memory consumption. This way, no other language will be able to dominate it. Languages which are not on the top of the ranking, even if individually optimized weights are assigned, are definitely classified as inefficient [6].

The rest of the paper is organized as follows. Section 2 introduces the DEA methodology and the particular DEA model we have considered. Section 3 describes the data set. Section 4 reports the results of the analysis, and in Section 5 conclusions are drawn.

2. Efficiency Evaluation via DEA: Choice of the Model

The concept of efficiency is used to measure the quality of a transformation of resources into goods and services. Several definitions of efficiency are commonly adopted in practice. The

so-called Pareto-efficiency, for example, is reached when there are no possible alternative allocations of resources whose realization would bring a gain. In terms of inputs and outputs, this means that no more output can be obtained without increasing some input.

Data Envelopment Analysis is a relatively young but well-established optimization-based technique. As mentioned before, it allows to measure and compare the performance of economic entities, each of them constituting a so-called Decision Making Unit.

DMUs are considered as black boxes converting multiple inputs into multiple outputs. Typical examples of DMUs appearing in DEA applications are universities, hospitals, and business firms. They are compared to each other for ranking their performances.

As already mentioned, we consider each language appearing in the CLBG site as an autonomous unit, that is, we assume it to be a DMU.

With DEA, the performance evaluation of a DMU does not require any *a priori* decision on the weights for inputs and outputs. Such weights are chosen so as to maximize the efficiency, that is, the ratio of the weighted sum of outputs to the weighted sum of inputs of the considered DMU. This kind of weights' choice is in fact the main feature of the DEA methodology. Other common definitions of productivity, instead, typically use *a priori* fixed, subjective, weights.

DEA's efficiency scores are normalized and bounded between 0 and 1. Languages that obtain an efficiency score lower than one, that is, that are not on the top of the ranking, are classified as inefficient [6, 7].

Several DEA models have been developed in the last three decades (see, e.g., [8]). In general, they may differ in the way they treat returns to scale (e.g., constant or variable), in their orientation (input-oriented, output-oriented, nonoriented), in the type of efficiency they measure, and so on. Different models are more or less appropriate to be used depending on the nature of the considered problem: their specific properties make them apt do meet the different properties of the data at hand. We will now briefly describe the model we choose for elaborating computer language efficiency scores.

The original DEA models (see, e.g., the CCR model [6]) were developed by using radial efficiency measure and make it difficult to distinguish between Pareto optimality and a rather weaker efficiency measure called "technical efficiency." For being able to make this distinction, we choose the SBM model [7, 8] that explicitly embeds the slack variables into the model. In this way, the SBM model allows a better stratification of the DMUs' performance levels.

We opt for a nonoriented version, which allows the consideration of both input reductions and output expansions, instead of limiting the analysis to one of these aspects.

Furthermore, since computer language inputs vary greatly in size, we consider the variable return to scale (VRS) version of this model.

Formally, for each DMU, the efficiency score is obtained as the solution of a mathematical programming problem. Consider a set of n decision making units, DMU_1, $\mathrm{DMU}_2, \ldots, \mathrm{DMU}_n$, each of them employing m inputs and producing s outputs. This set is called the technology set.

The SBM model, in its nonoriented version with variable returns to scale, defines the efficiency of a chosen DMU (DMU_0) among n in the following way:

$$\min_{\{\boldsymbol{\lambda}, \mathbf{s}^-, \mathbf{s}^+\}} \frac{1 - (1/m)\sum_{i=1}^{m} \left(s_i^- / x_{i0}\right)}{1 + (1/s)\sum_{r=1}^{s} \left(s_r^+ / y_{r0}\right)} \tag{2.1}$$

subject to

$$
\begin{aligned}
x_{i0} &= \sum_{j=1}^{n} x_{ij}\lambda_j + s_i^-, \quad i = 1, 2, \ldots, m, \\
y_{r0} &= \sum_{j=1}^{n} y_{rj}\lambda_j + s_r^-, \quad r = 1, 2, \ldots, s, \\
&\sum_{j=1}^{n} \lambda_j = 1, \\
&\lambda_j \geq 0, \\
&s_i^- \geq 0, \quad i = 1, 2, \ldots, m, \\
&s_r^+ \geq 0, \quad r = 1, 2, \ldots, s,
\end{aligned}
\tag{2.2}
$$

where x_{ij} is input i of unit j, y_{rj} is output r of unit j, s_i^-, and s_r^+ are the slack variables associated to input i and to output r, respectively.

If the optimal value of the SBM model is not equal to 1, then DMU_0 is inefficient: even with the most favorable weights. DMU_0 turns out to be dominated by at least another DMU. DMUs with efficiency 1 are called SBM-efficient (SBM-efficiency coincides with the so-called Pareto-Koopmans efficiency concept, see, e.g., [7]).

The above model has to be applied for each DMU belonging to the technology set. Remark that computer languages are not completely homogeneous DMUs: as we will discuss in Section 3, they can be *a priori* divided into groups with clearly distinguishable characteristics, as, for example, compiled and interpreted languages. For this reason, it is interesting to use a meta-frontier approach [9, 10], comparing first the languages belonging to an homogeneous group (e.g., compiled with compiled). This way, we obtain group specific efficiency scores: in this framework, the set of efficient DMUs is called the group-frontier. In a similar way, we compute the efficiency scores with respect to all the languages, obtaining the so-called metaefficiency scores. The set of DMUs with metaefficiency score equal to 1 constitutes the meta-frontier. Computing the ratio between metaefficiency and group-efficiency scores (metatechnology ratio) allows to measure the relative efficiency of each language. The higher this ratio, the closer a language behavior to the "best language among the whole technology set."

3. The Dataset

The Computer Language Benchmarks Game has started in 2002 under the name *Great Computer Language Shootout*. It is a collection of programs that tackle 19 benchmark problems, implemented across 38 programming languages, and tested on different workloads.

The code reported is provided by volunteer programmers. They aim at achieving the best result in the competition for each benchmark. All the experiments are run on a single-processor 2 Ghz Intel Pentium 4 machine with 512 MB of RAM and 80 GB IDE disk drive. The operating system used is Ubuntu 9.04 Linux Kernel 2.6.28-11-generic. The specific data considered in the current analysis have been collected on March 09, 2009.

Several programs may be uploaded on the website for a single benchmark. All the programs written in the same language for the same problem are pairwise compared with respect to several criteria. In each comparison, if one of them is not better than the other one according to the whole set of available measurement, and it is worse under at least one measurement, then it is eliminated from the competition. In this way, the survived programs can be considered as the ones achieving the best possible results allowed by a language on a benchmark problem.

Each program is run and measured for every benchmark and each workload. If the program gives the expected output within a cutoff time of 120 seconds, five independent runs are performed. If the program does not return the expected output within a timeout of one hour, it is forced to quit. The time measurements include the program startup time. The memory use is measured every 0.2 seconds. The measurements recorded are the lowest time and highest memory usage from repeated measurements, no matter whether forcefully terminated or not.

The data reported have been object of some intuitive assessment (see, e.g., [11, 12]). To the best of our knowledge, a rigorous multicriteria analysis has never been proposed neither on this nor on other similar datasets.

3.1. Selection of Input/Output Data

Based on the data available, the aim of our analysis is comparing different languages when performing at their best. CPU time, memory usage, and source code size are generally considered the crucial characteristics of the efficiency of computer programs. The latter can be considered as proportional to the implementation effort [3].

For each benchmark, three workloads are tackled. Hence, three measurements are available for each program, both for CPU time and memory usage, which together to source code size represent the 7 inputs used. If multiple programs written in the same language tackle a workload, the input is the sum of the measurements of all of them. For example, if three programs tackle a workload in t_1, t_2, and t_3 seconds, respectively, the input time we use for that workload is $t_1 + t_2 + t_3$. As outputs, we consider the number of programs that tackle each workload. In the example, the output associated to the workload is equal to 3. We consider this output as a proxy of the easiness to use of the language. Despite this is a crucial aspect in the choice of a language, it is indeed not easily quantifiable. In principle, though, the easier the use of a language, the more people will choose it for writing their programs.

In the analysis, we ignore benchmark problems on which less than 30 languages are tested, as well as languages coded for less than 14 benchmark problems.

3.2. The Programming Languages Considered

A set of thirty-five programming languages are compared. They are all well known in the scientific community; their names and the main characteristics of the implementations considered here are reported in Table 1. The first distinction that we make is based on the programming paradigm: imperative, declarative, and multiparadigm languages are present in the list. Such paradigms differ in the concepts and abstractions used to represent the elements of a program (such as objects, functions, variables, and constraints) and the steps that compose a computation (assignation, evaluation, data flows, etc.). Secondly, static and dynamic languages are distinguished: the process of verifying and enforcing the constraints

Table 1: Programming languages considered.

Language	Characteristics
Ada	Multiparadigm, static, compiled
C	Imperative, static, compiled
CTiny	Imperative, static, interpreted
C#	Multiparadigm, static, compiled
C++	Multiparadigm, static, compiled
Clean	Declarative, static, compiled
D	Multiparadigm, dynamic, compiled
Eiffel	Multiparadigm, static, compiled
Erlang	Declarative, dynamic, compiled
Forth	Imperative, dynamic, compiled
Fortran	Imperative, static, compiled
Haskell	Declarative, static, compiled
Java	Imperative, static, compiled
JavaScript	Multiparadigm, dynamic, interpreted
Lisp	Multiparadigm, dynamic, compiled
Lua	Multiparadigm, dynamic, interpreted
Nice	Imperative, static, compiled
Oberon	Imperative, static, compiled
ObjectiveC	Imperative, dynamic, compiled
Ocaml	Multiparadigm, static, compiled
Oz	Multiparadigm, dynamic, compiled
Pascal	Imperative, static, compiled
Perl	Multiparadigm, dynamic, interpreted
PHP	Imperative, dynamic, interpreted
Pike	Multiparadigm, dynamic, interpreted
PIR	Imperative, dynamic, interpreted
Prolog	Declarative, dynamic, compiled
Python	Multiparadigm, dynamic, compiled
Ruby	Multiparadigm, dynamic, interpreted
Scala	Multiparadigm, static, compiled
Scheme	Declarative, dynamic, compiled
SLang	Imperative, dynamic, interpreted
Smalltalk	Imperative, dynamic, compiled
SML	Multiparadigm, static, compiled
Tcl	Multiparadigm, dynamic, interpreted

of types may occur either at compile-time (a static check) or run-time (a dynamic check). Finally, languages implementations may be compiled or interpreted.

3.3. The Benchmark Problems

We consider 14 benchmark problems. The CLBG site reports a detailed description of each of them. Here, we report only their main characteristics.

Binary-Trees

Each program defines, allocates, and deallocates binary trees. A long-lived binary tree is to be allocated; it lives on while other trees are allocated and deallocated. Then, many further trees are allocated, their nodes are walked through, and they are finally deallocated.

Fannkuch

A permutation of n numbers is considered. The problem consists in applying the following procedure: consider the first element i of the permutation; reverse the order of the first i numbers; repeat until the first element is a 1. Do this for each of the $n!$ permutations. Finally, write the first 30 permutations and the number of flips.

Fasta

The expected cumulative probabilities for 2 alphabets must be encoded. Then, DNA sequences are generated by weighted random selection from the alphabets. A given linear generator is to be used. Finally, three DNA sequences are written line-by-line in Fasta format, which is a text-based format for representing either nucleotide sequences or peptide sequences: base pairs or amino acids are represented using single-letter codes.

Mandelbrot

The Mandelbrot set (from $-1.5 - i$ to $0.5 + i$) must be plotted on an n-by-n bitmap.

N-Body

The orbits of Jovian planets have to be modeled by using a simple symplectic integrator.

Nsieve

The prime numbers up to a fixed value are to be counted by using the naïve Sieve of Eratosthenes algorithm [13].

Nsieve-Bits

The Nsieve-bits problem is similar to the previous case, but it is based on arrays of bit flags.

Partial-Sums

An iterative double-precision algorithm must be used for calculating partial sums of eight given series of real numbers.

Pidigits

A step-by-step spigot algorithm [14] is to be used for calculating digits of π. The problem consists in computing the first n digits of π, and printing them 10-to-a-line, with the running total of digits calculated.

Recursive

Three simple numeric functions must be computed, namely, Ackermann [15], Fibonacci [16], and Tak [17]. For this benchmark, the Fibonacci and Tak implementations should either provide separate functions—one for integer calculation and one for double calculation—or provide a function that uses integer calculation with integer parameters and double calculation with double parameters.

Reverse-Complement

The following procedure must be executed: a Fasta format file is to be read line-by-line. Then, for each sequence, the id, the description, and the reverse-complement sequence must be written in Fasta format.

Spectral-Norm

The spectral norm of an infinite matrix A must be computed, with given entries.

Startup

Each program should print "hello world" and exit. This benchmark measures startup costs. Each program is run several times in a loop by a shell script wrapper.

Sum-File

This problem consists in reading integers, one line at a time, and printing the sum of those integers.

3.4. Variability in the Input Data

Figures 1, 2, and, 3 represent the distribution of the values recorded for CPU time, memory usage, and source code size, respectively. The boxplots report the median, and the first and the third quartiles of the distribution of the quantity referred to each language.

We can observe that some languages are rather well performing in terms of CPU time, as C and C++, but are quite poor in terms of source code size. The opposite holds for languages as, for example, Tcl and Python (see, Figures 1 and 2). Source code size is the strength of JavaScript, Perl, Python, and Ruby, as displayed in Figure 3. These results are coherent with the literature [1, 3, 4]. Nevertheless, observing these graphs, we cannot immediately identify efficient versus inefficient languages: each feature suggests a different ranking.

4. Experimental Results

The efficiency of each language is assessed through DEA. As mentioned, inputs are computational time, memory usage, and source code size tested on different workloads, and outputs are the number of times each workload is tackled. The first element to observe is the performance of the various programming languages individually.

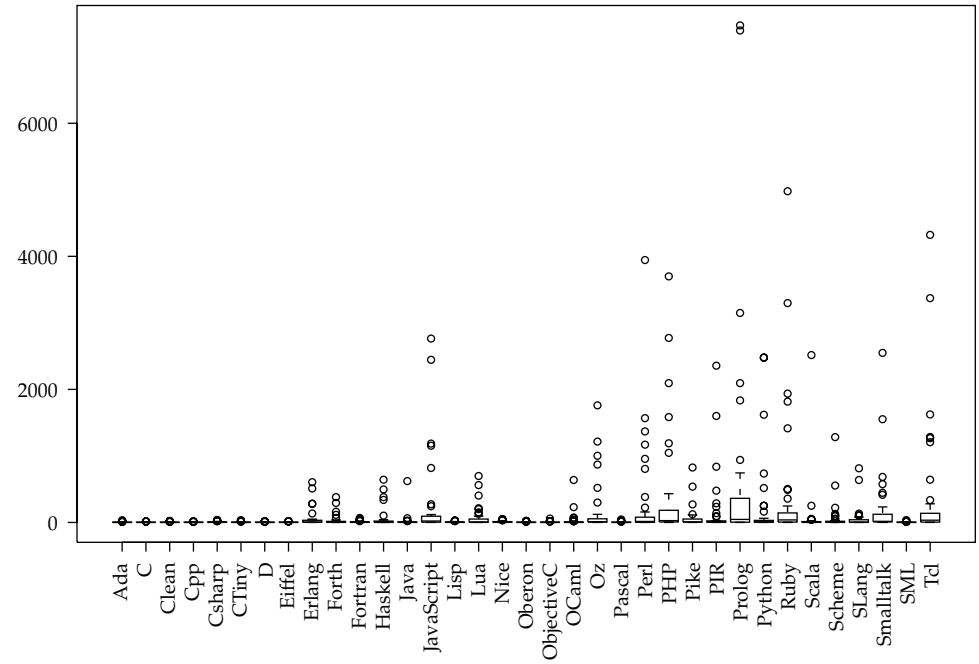


Figure 1: Distribution of the values of CPU time recorded for each language.

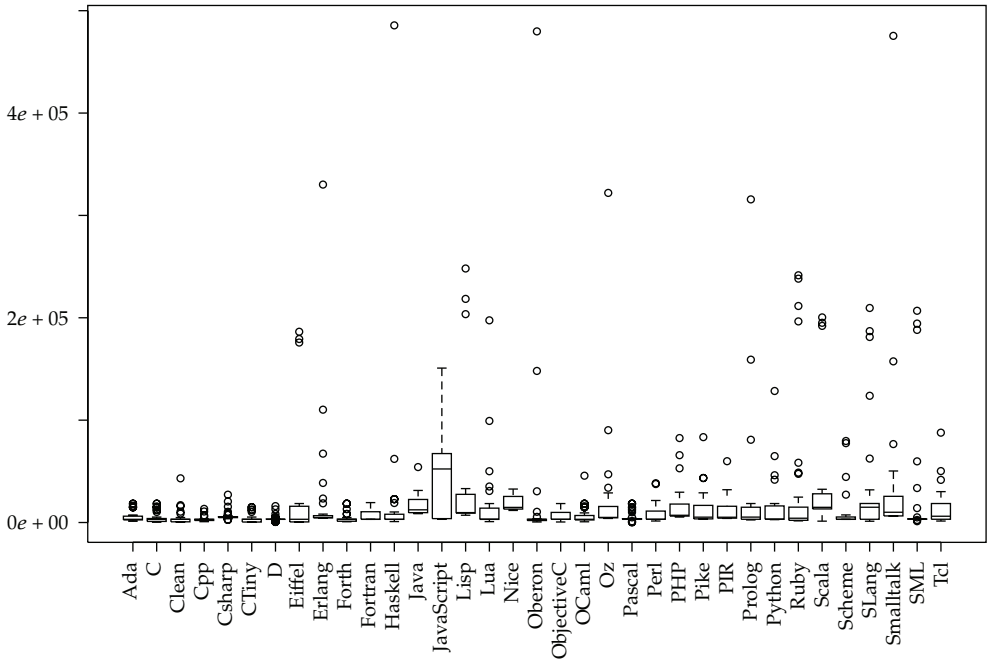


Figure 2: Distribution of the values of memory usage recorded for each language.

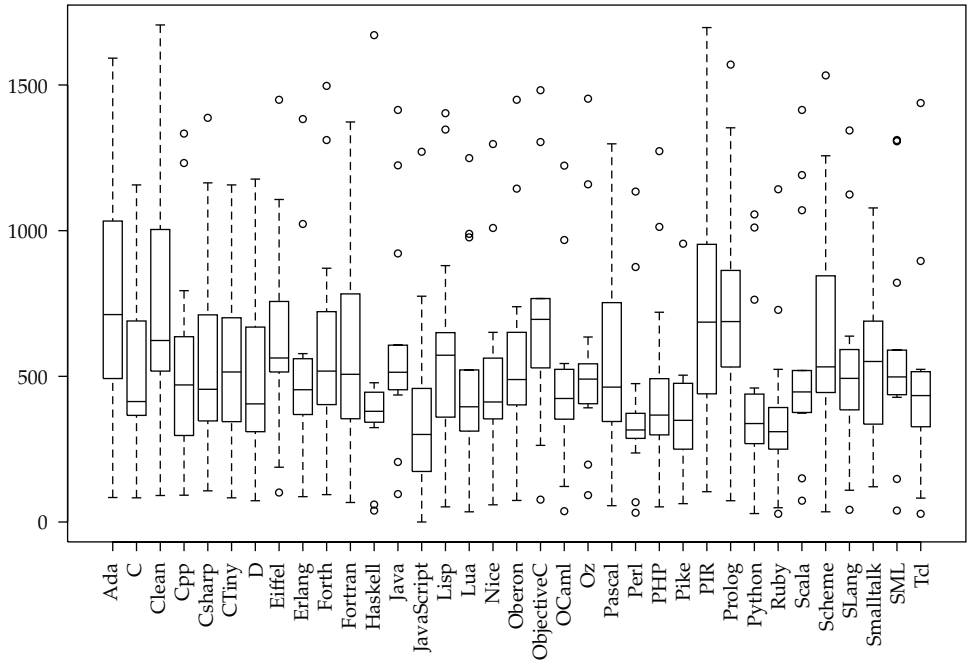


Figure 3: Distribution of source code size of each language.

Figure 4 reports the results on the efficiency of each of language. The boxplots point out the median and the first and third quartiles of the distributions of observations. The order in which the languages are listed places at the extreme left the one for which the median of the observations is maximum; when this value is equivalent for two languages, the standard deviation decides the order.

Table 2 summarizes DEA results. It reports the mean and lower efficiency scores for each benchmark problem, together with the number of relatively efficient languages. Table 3 depicts the detailed efficiency score for each programming language.

As it can be observed, the best performing language results to be C: the programs are efficient on almost all the benchmarks considered. A similar evaluation holds for D, Clean, C++, Pascal, and Python. The efficiency of these languages is very robust: the standard deviation of the relative distributions is very low, and the median value is one in all these cases.

Figure 5 reports the boxplots of the distributions of the metatechnology ratio of the various languages grouped according to three different categorizations. First imperative versus declarative versus multiparadigm, then static versus dynamic, finally compiled versus interpreted languages are considered. As it can be observed, the families named multiparadigm, static, and compiled are the best performing.

The discussion on the convenience of static versus dynamic languages, in particular, has been widespread on various blogs dedicated to computer languages on the web (e.g., [18–24]). An agreement on this issue is still lacking, and in general the arguments offered are not supported by deep experimental analysis. In general, the main merit recognized to the static languages is that they can guarantee the type compatibility of program elements without excessive testing. On the other hand, the principal advantage of dynamic ones is that

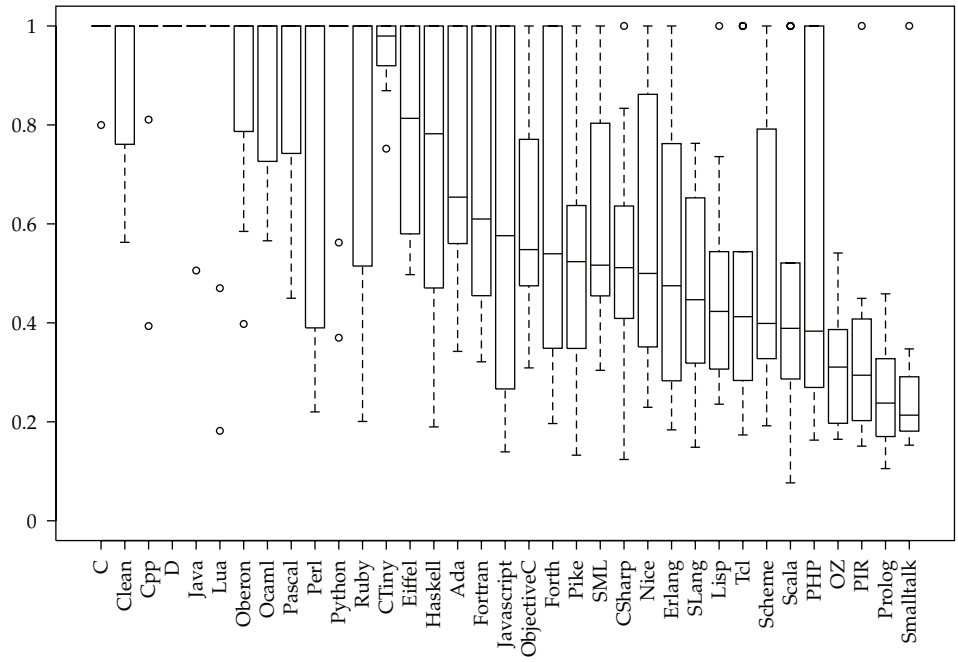


Figure 4: Efficiency of each language on the benchmark problems.

Table 2: Summary of the results of the DEA analysis: mean and lowest efficiency scores over the available set of benchmark problems.

	Binary trees	Fannkuch	Fasta	Mandelbrot	*n*-body	*n*-sieve	*n*-sieve bits
Mean eff.	0.68	0.74	0.72	0.74	0.58	0.65	0.67
Lower eff.	0.18	0.18	0.29	0.34	0.14	0.11	0.15
No. of eff.	15	15	11	14	11	10	14
	Partial-sum	Pidigits	Recursive	Reverse-complement	Spectral-norm	Startup	Sum-file
Mean eff.	0.74	0.69	0.62	0.65	0.63	0.591	0.68
Lower eff.	0.22	0.18	0.17	0.19	0.12	0.08	0.18
No. of eff.	17	13	12	12	11	13	12

they make possible to avoid uninteresting type declarations, which make code more verbose and add noise that distracts from the essence of an algorithm. The analysis proposed here suggests that static languages are in general preferable to dynamic ones.

5. Conclusions

In this paper, we used Data Envelopment Analysis for assessing the efficiency of the main programming languages. We exploit its ability of comparing elements on the basis of several criteria of different nature. In particular, we consider the computational time, the memory usage, and the source code size required for solving various benchmark problems under multiple workloads. The programs analyzed are publicly available in the repository of

Table 3: Detailed results of the DEA analysis.

Language	Efficient	Binary trees	Fannkuch	Fasta	Mandelbrot	*n*-body	*n*-sieve	*n*-sieve bits	Partial-sum	Pidigits	Recursive	Reverse-complement	Spectral-norm	Startup	Sum-file
Ada	4	0.652	0.563	0.727	**1.000**	0.624		**1.000**	0.656	**1.000**		**1.000**	0.557	0.365	0.342
C	13	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	0.8	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**
C Tiny	5	**1.000**	**1.000**	0.752		**1.000**	0.951		**1.000**	0.959	**1.000**	0.869			0.920
C#	1	0.408	0.472	0.636	0.572	0.409	0.709	**1.000**	0.519	0.568	0.312	0.423	0.505	0.124	0.834
C++	12	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	0.393	0.811
Clean	8	**1.000**	0.746	0.761	**1.000**	**1.000**		**1.000**	**1.000**	0.719	**1.000**	**1.000**	0.899	0.563	**1.000**
D	14	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**
Eiffel	5		0.606	0.953	**1.000**	**1.000**	0.526	**1.000**	**1.000**	0.497	0.813	0.556	**1.000**	0.580	0.584
Erlang	2	**1.000**	0.315	0.525	0.453	0.251		0.497	0.403	0.501	**1.000**	0.188		0.084	0.184
Forth	4	**1.000**	**1.000**	**1.000**	0.579	0.349		0.615	0.500	0.421	0.300	**1.000**	0.196	0.413	0.305
Fortran	4	0.321	**1.000**	0.586		0.633	0.954	**1.000**	**1.000**		0.368	0.484	0.426	**1.000**	0.584
Haskell	5		**1.000**			0.564	0.534	**1.000**	0.782	**1.000**	0.407	0.190	**1.000**	**1.000**	0.343
Java	13	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	0.506	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**
JavaScript	5	0.206	0.576	**1.000**		0.139	0.271	0.317	**1.000**			0.261	**1.000**	**1.000**	**1.000**
Lisp	1	0.736	0.395	0.438	0.537	0.544	0.408	**1.000**	0.563	0.236	0.272	0.403	0.462	0.270	0.307
Lua	11	**1.000**	**1.000**	**1.000**	**1.000**	0.182	**1.000**	0.470	**1.000**	**1.000**		**1.000**	**1.000**	**1.000**	**1.000**
Nice	2	0.390	**1.000**	**1.000**		0.382		0.724	0.321	0.229	0.287		0.621	0.088	0.500
Oberon	6	0.398	0.673	**1.000**	**1.000**	**1.000**	0.953		0.901		**1.000**		**1.000**	0.585	**1.000**
Objective-C	1	0.502	0.607	0.907		0.572				**1.000**	0.339	0.771	0.309	0.524	0.475
Ocaml	9	**1.000**	0.755	0.724	0.566	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	0.656	**1.000**	0.726
Oz	0	0.520	0.427	0.349	0.386	0.165	0.222	0.335	0.328	0.541	0.197	0.189	0.272	0.174	0.293
Pascal	9	**1.000**	**1.000**	0.823	**1.000**	0.722	**1.000**	**1.000**	**1.000**	**1.000**	0.450	0.742	0.495	**1.000**	**1.000**
Perl	8	0.390	**1.000**	0.661	**1.000**	0.285	0.220	**1.000**	**1.000**	**1.000**	0.349	**1.000**	0.598	**1.000**	**1.000**
PHP	4	0.245	0.387	0.329	0.379	0.163	**1.000**	0.333	**1.000**	0.269	**1.000**	0.442	**1.000**	0.210	0.438
Pike	1	0.392	0.658	0.539			0.637	0.312		**1.000**	0.348	0.633		0.133	0.508
PIR	1	0.176	0.176	0.294	0.406	0.252	**1.000**	0.393	0.408	0.238		0.449	0.151	0.202	0.433
Prolog	0	0.260		0.390	0.459	0.175	0.105	0.235	0.223	0.432	0.166		0.122	0.266	0.241
Python	12	**1.000**	**1.000**	**1.000**	0.562	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	0.370	**1.000**	**1.000**	**1.000**	**1.000**
Ruby	8	**1.000**	**1.000**	**1.000**	**1.000**	0.253	0.201	0.515	**1.000**	0.317	**1.000**	**1.000**		0.824	**1.000**
Scala	3	**1.000**	**1.000**	0.428	0.379	0.227	0.469	0.521	0.287	0.258	**1.000**	0.291	0.399	0.077	0.375
Scheme	3		0.395	0.583	**1.000**	0.432	0.403	0.307	0.228	0.393	0.192	0.348		**1.000**	**1.000**
S-Lang	0	0.281	0.701	0.437	0.545	0.356	0.248	0.149	0.656				0.447	0.649	0.763
Smalltalk	1			0.347	0.336	**1.000**	0.205	0.227	0.246	0.182		0.213	0.153	0.154	0.180
SML	3	**1.000**	0.441	0.746	**1.000**	0.738	0.464	0.548	**1.000**	0.803	0.375	0.469	0.455	0.304	0.485
Tcl	3	0.288	0.414	0.472	0.544	0.234	0.304	0.284	**1.000**		0.173	0.412	0.248	**1.000**	**1.000**

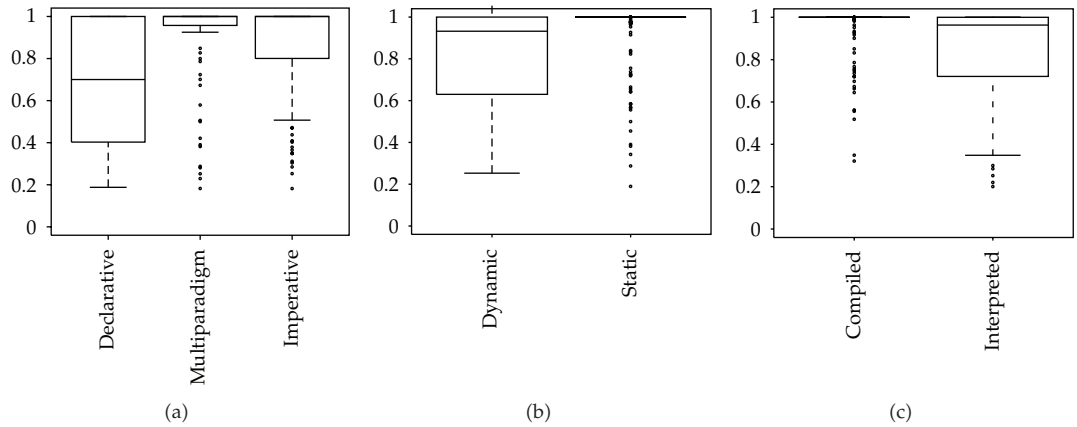


Figure 5: Efficiency of groups of languages: imperative versus declarative versus multiparadigm, static versus dynamic, compiled versus interpreted.

the *The Computer Language Benchmarks Game*. Multiple programs may be contributed for tackling each benchmark problem, and the worst are eliminated from the game. Thus, the implementations reported can be considered as good proxies of the languages' potential [3, 25]. As a consequence, one can argue that, for example, since C is more efficient than Tcl in this data set, this relationship should hold in general terms, whenever codes are properly optimized, at least if the same attributes are considered.

According to the results obtained, the most efficient languages are C, D, Clean, C++, Pascal, and Python while Smalltalk, Prolog, PIR, and OZ are definitely inefficient. Such a conclusion supports the expectation driven by common experience. By grouping languages, it is possible to observe the impact of their main characteristics, such as being imperative, declarative or multiparadigm, static or dynamic, compiled or interpreted. The strength of the influence on efficiency of these features appears different.

Our research aims at supporting with scientific evidence, as far as possible, the choice of the programming language to be used for implementing efficient algorithms. The relevance of such a choice is often neglected, while the proposed analysis shows that it is far from being unimportant. In particular, it appears crucial in the current practice of optimization, in which great effort is devoted to boost the performance of solution algorithms.

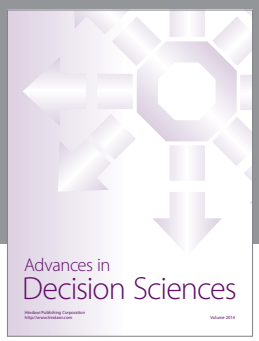
The proposed methodology can be easily applied to different datasets. Undoubtedly, other properties of languages might be considered, such as reliability or syntax toughness, for achieving a finer classification.

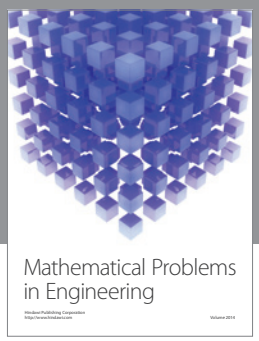
All the available benchmark problems are considered in this phase as equally important. A further refinement of the results may be obtained by weighting them differently according to the similarity to the specific problem one needs to tackle. A possible tool for assigning such weights may be represented by the application of the Analytic Hierarchy Process (AHP) [26, 27], a powerful instrument for multicriteria analysis.

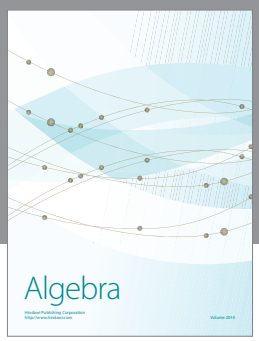
References

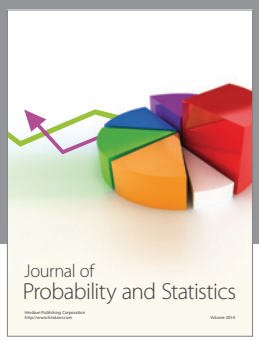
[1] K. Aldrawiesh, A. Al-Ajlan, Y. Al-Saawy, and A. Bajahzar, "A comparative study between computer programming languages for developing distributed systems in web environment," in *Proceedings of the 2nd International Conference on Interaction Sciences: Information Technology, Culture and Human (ICIS '09)*, pp. 457–461, ACM, New York, NY, USA, November 2009.

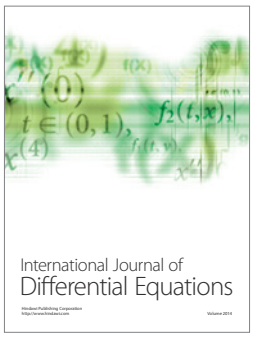
[2] A. R. Feuer and N. H. Gehani, "Comparison of the programming languages C and pascal," *Computing Surveys*, vol. 14, no. 1, pp. 73–92, 1982.
[3] L. Prechelt, "Are scripting languages any good? A validation of Perl, Python, Rexx, and Tcl against C, C++, and java," *Advances in Computers*, vol. 57, no. C, pp. 205–270, 2003.
[4] C. Ruknet, *Guide to Programming Languages: Overview and Comparison*, Artech House, Norwood, Mass, USA, 1995.
[5] M. Shaw, G. T. Almes, J. M. Newcomer, B. K. Reid, and W. A. Wulf, "Comparison of programming languages for software engineering," *Software—Practice and Experience*, vol. 11, no. 1, pp. 1–52, 1981.
[6] A. Charnes, W. W. Cooper, and E. Rhodes, "Measuring the efficiency of decision making units," *European Journal of Operational Research*, vol. 2, no. 6, pp. 429–444, 1978.
[7] K. Tone, "A slacks-based measure of efficiency in data envelopment analysis," *European Journal of Operational Research*, vol. 130, no. 3, pp. 498–509, 2001.
[8] W. W. Cooper, L. M. Seiford, and K. Tone, *Data Envelopment Analysis*, Springer, New York, NY, USA, 2nd edition, 2007.
[9] C. J. O'Donnell, D. S. P. Rao, and G. E. Battese, "Metafrontier frameworks for the study of firm-level efficiencies and technology ratios," *Empirical Economics*, vol. 34, no. 2, pp. 231–255, 2008.
[10] G. E. Battese, D. S. Prasada Rao, and C. J. O'Donnell, "A metafrontier production function for estimation of technical efficiencies and technology gaps for firms operating under different technologies," *Journal of Productivity Analysis*, vol. 21, no. 1, pp. 91–103, 2004.
[11] A. Lopez Ortega, "Programming languages benchmark," Alvaro's Site, 2007, http://www.alobbs.com/1307/Programming Languages Benchmark.html.
[12] G. Marceau, "The speed, size and dependability of programming languages," Square root of x divided by zero, 2009, http://gmarceau.qc.ca/blog/2009/05/speed-size-and-dependability-of.html .
[13] S. Horsley, "The sieve of Eratosthenes. Being an account of his method of finding all the prime numbers," *Philosophical Transactions*, vol. 62, pp. 327–347, 1972.
[14] S. Rabinowitz and S. Wagon, "A spigot algorithm for the digits of Pi," *The American Mathematical Monthly*, vol. 102, no. 3, pp. 195–203, 1995.
[15] W. Ackermann, "Zum Hilbertschen Aufbau der reellen Zahlen," *Mathematische Annalen*, vol. 99, no. 1, pp. 118–133, 1928.
[16] V. E. Hoggatt, and M. Bicknell, "Roots of Fibonacci polynomials," *The Fibonacci Quarterly*, vol. 11, no. 3, pp. 271–274, 1973.
[17] I. Vardi, *Computational Recreations in Mathematica*, chapter 9, Addison-Wesley, Redwood City, Calif, USA, 1991.
[18] C. Beust, "Dynamic and static languages are fighting again," Otaku, Cedric's weblog, 2007, http://beust.com/weblog/archives/000454.html.
[19] C. Beust, T. Neward, O. Beni, and G. Young, "Debate and more insights on dynamic vs. static languages," Sadek Drobis Blog, 2008, http://sadekdrobi.com/2008/05/23/debate-and-more-insights-on-dynamic-vs-static-languages.
[20] S. Drobi, "Debate and more insights on dynamic vs. static languages," Info Queue, 2008, http://www.infoq.com/news/2008/05/more-insights-static-vs-dynamic.
[21] S. Jain, "Remain static or go dynamic?" Pathfinder development, 2009, http://www.pathf.com/blogs/2009/04/remain-static-or-go-dynamic.
[22] M. Podwysocki, "Static versus dynamic languages—attack of the clones," Matthew Podwysocki's Blog, 2008, http://weblogs.asp.net/podwysocki/archive/2008/05/28/static-versus-dynamic-languages-attack-of-the-clones.aspx.
[23] A. Turoff, "Static vs. dynamiclanguages," Notes on Haskell, 2008, http://notes-on-haskell.blogspot.com/2008/05/static-vs-dynamic-languages.html.
[24] B. Venners, "Static versus dynamic attitude," Artima Developer, 2005, http://www.artima.com/weblogs/viewpost.jsp?thread=92979.
[25] L. Prechelt, "Empirical comparison of seven programming languages," *Computer*, vol. 33, no. 10, pp. 23–29, 2000.
[26] T. L. Saaty, "A scaling method for priorities in hierarchical structures," vol. 15, no. 3, pp. 234–281, 1977.
[27] T. Saaty, *Fundamentals of the Analytic Hierarchy Process*, RWS Publications, Pittsburgh, Pa, USA, 200.

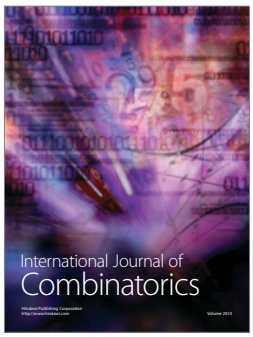


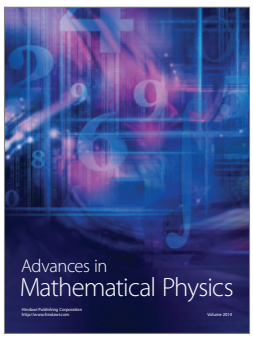


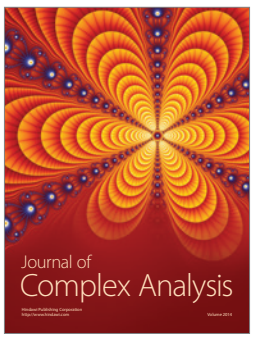


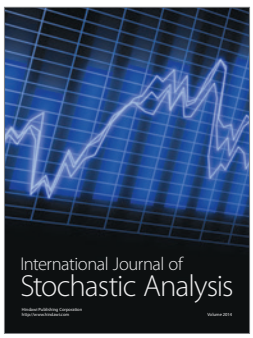


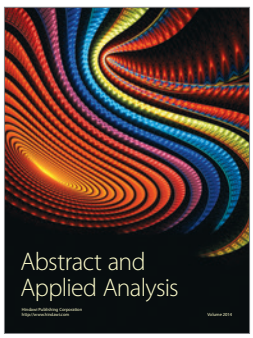


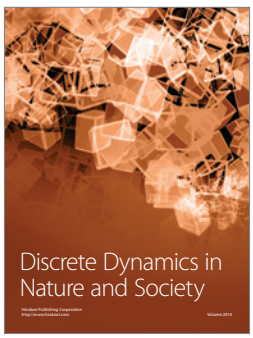


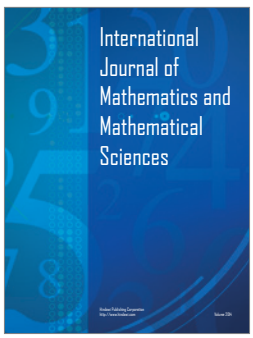


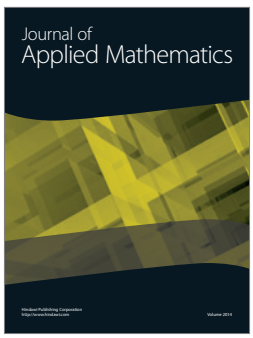












Hindawi
Submit your manuscripts at
http://www.hindawi.com