How Can Metaheuristics Help Software Engineers?

Enrique Alba(✉)

Universidad de Málaga, Málaga, Spain
eat@lcc.uma.es
http://www.lcc.uma.es/∼eat

Abstract. This paper is a brief description of the revamped presentation based in the original one I had the honor to deliver back in 2009 during the very first SSBSE in London. At this time, the many international forces dealing with search, optimization, and learning (SOL) met software engineering (SE) researchers in person, all of them looking for a quantified manner of modeling and solving problems in software. The contents of this work, as in the original one, will develop on the bases of metaheuristics to highlight the many good ways in which they can help to create a well-grounded domain where the construction, assessment, and exploitation of software are not just based in human expertise, but enhanced with intelligent automatic tools. Since the whole story started well before the first SSBSE in 2009, we will mention a few previous applications in software engineering faced with intelligent algorithms, as well as will discuss on the present interest and future challenges of the domain, structured in both short and long term goals. If we understand this as a cross-fertilization task between research fields, then we could learn a wider and more useful lesson for innovative research. In short, we will have here a semantic perspective of the old times (before SBSE), the recent years on SBSE, and the many avenues for future research and development spinning around this exciting clash of stars. A new galaxy has been born out of the body of knowledge in SOL and SE, creating forever a new class of researchers able of building unparalleled tools and delivering scientific results for the benefit of software, that is, of modern societies.

Keywords: Search · Optimization · Learning · Metaheuristic
Software engineering · Computational intelligence

1 Converging Trajectories

Software engineering (SE) and complex search/optimization/learning (SOL) are two important and historical knowledge areas in Computer Science (CS), and

Supported by the Spanish-FEDER projects TIN2017-88213-R and TIN2016-81766-REDT.

T. E. Colanzi and P. McMinn (Eds.): SSBSE 2018, LNCS 11036, pp. 89–105, 2018.
https://doi.org/10.1007/978-3-319-99241-9_4

the bases for the vast majority of applications of IT in today's world. But, with a few exceptions, separate research fields.

As to SE, its history is linked to the very nature of computers, with a deep relation to programming and the art/engineering task of planning, executing, and delivering products and services. Since its first conception, it was clear in SE that building, using, extending, and maintaining software is a very complex task, where we would need the help of computer tools to complement human experience and even creativity.

As to SOL, the landscape is still larger and older: optimization is embedded in most activities of life, and computer procedures aim for maximizing benefits, reducing costs, and searching in innovative ways, all playing a major role in every subfield of CS and IT. Indeed, optimization has greatly grown in the company of machine learning, and both admit a point of view in which the procedure for searching/learning (or the found result itself) is the focus of the study.

It was just a matter of time that the two areas got together, enriching each other and merging into a new domain where SE is understood as a real/normal engineering work, with artifacts that can be numerically modeled, and the management of the process and product has a quantitative flavor, that however has been largely dismissed in the SE community (with some important exceptions!).

From seminal works like [2,20] it was clear that SOL could transform the way in which SE deals with products and services, by measuring software quality in a numerical manner (performance, but also usability, security, ...), what would allow *automatic and intelligent* (both!) decision making and guidance. In this context, the search for better software solutions has finally converged to the so called *Search-Based Software Engineering*, a term coined by Harman [16] and popularized by an active community of practitioners across the world. Search and also optimization are there paramount to solve traditional problems of software (like program testing) and support new ones (like automatic repair or identification of potential defects), and thus there is a natural interest in knowing more on the techniques that would allow such an enhanced management of software tools.

Among the many techniques for SOL, metaheuristics [15,27] represent a para-mount field feeding algorithms and operations that would allow numerical modeling of SE problems, and will ease an important boosting in quality of the software we all are building nowadays. Metaheuristics exhibit a short list of requirements to be applied and a long list of benefits. On the requirements, they need **(a)** to be able of encoding a solution to the problem (*phenotype*) in a kind of vector of symbols (*genotype*) with no loss of generality (remember that the memory of a computer is a big vector) and **(b)** to be able of assessing the relative quality of two vectors as potential problem solutions. With only this, an iterative improvement of tentative solutions by means of some (non deterministic) operators is able of reaching solutions for problems out of the reach of classical methods. This is true because metaheuristics can deal with a list of handy real settlements: no need for differentiation, able to manage arbitrary constraints, no need of analytical description of the problem (!), facility to deal with continuous and discrete representations... all of them wanted in any field where problems are frequently ill-defined, very large, and of course NP-hard.

This article contains a brief discussion on what, how, and when these techniques can help SE researchers and practitioners.

In the next section, we will describe the main types of metaheuristics and the many useful extensions to cover virtually any problem that SE wants to throw to them. In Sect. 3 we will present some issues on SBSE and a somewhat new twist, going beyond SBSE to suggest how SE can have an impact in metaheuristics also, a bit on the contrary of what the reader can expect. Then, in Sect. 4 we will define some interesting and new open challenges to the community, just to end in Sect. 5 with conclusions and several of the many future works ahead.

2 A High Level Glimpse on Metaheuristics

Most of works where metaheuristics are used nowadays fit in some manner the resolution of a global optimization problem. An optimization problem is defined as a pair (S, f), where $S \neq \emptyset$ is called the *solution space* (or search space), and f is a function named *objective* function or *fitness* function, defined as $f : S \rightarrow \mathbb{R}$ solving an optimization problem consisting in finding a solution $i^* \in S$ such that: $f(i^*) \leqslant f(i), \forall i \in S$. Note that assuming either maximization or minimization does not restrict the generality of the problem. Depending on the domain which S belongs to, we can speak of binary ($S \subseteq \mathbb{B}^*$), integer ($S \subseteq \mathbb{N}^*$), continuous ($S \subseteq \mathbb{R}^*$), or heterogeneous optimization problems ($S \subseteq (\mathbb{B} \cup \mathbb{N} \cup \mathbb{R})^*$).

A simple classification of the optimization methods used throughout the history of CS is shown in Fig. 1. In a first approach, the techniques can be classified into *exact* versus *approximate* (and *others* as a category for difficult to classify procedures). Exact techniques are based on the mathematical finding of the optimal solution, what can also be described as an exhaustive search until the optimum is found, guaranteeing the optimality of the obtained solution. However, these techniques present many practical drawbacks. The time they require, though bounded, may be very large, especially for NP-hard problems. Furthermore, it is not always possible to find such an exact technique for every problem. Indeed, exact techniques often require from the solved problem to exhibit special types of constraints or features (e.g., derivability, continuity, and having an analytical expression -not that usual!-). This makes exact techniques not to be a good choice in many occasions, either because their time and memory requirements can become unpractical or because the real problem does not really show the expected requirements to admit a solution with an exact technique. For this reason, approximate techniques have been widely used by the international research community in the last few decades. These methods (sometimes) sacrifice the guarantee of finding the optimum in favor of providing a satisfactory solution within reasonable times an real resource consumption.

Among approximate algorithms, we can find two types: *ad hoc* heuristics, and *metaheuristics*. Ad hoc heuristics can be further divided into *constructive* heuristics and *local search* methods. Constructive heuristics are usually the swiftest methods. They construct a solution from scratch by iteratively incorporating components until a complete solution is obtained, which is returned as

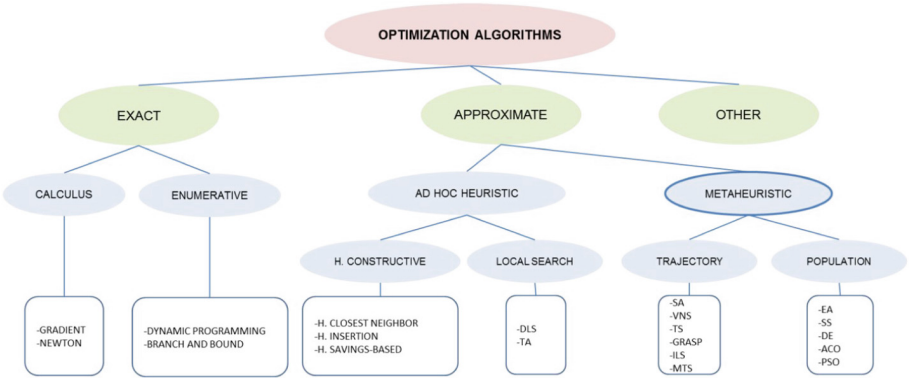

Fig. 1. Taxonomy of search algorithms.

the algorithm output. Finding a constructive heuristic that actually produces high quality solutions is a nontrivial task, since it mainly depends on the problem, and requires a thorough understanding of it. For example, in problems with many constraints, it could happen that many partial solutions do not lead to any feasible solution.

Local search or gradient descent methods start from a complete solution. They rely on the concept of *neighborhood* to explore a part of the search space defined for the current solution until they find a local optimum. The neighborhood of a given solution s, denoted as $N(s)$, is the set of solutions (neighbors) that can be reached from s through the use of a specific modification operator (generally referred to as a *movement*). A *local optimum* is a solution having equal or better objective function value than any other solution in its own neighborhood. The process of exploring the neighborhood, finding and keeping the best neighbor in the way, is repeated until the local optimum is found (or a maximum search budget has been exhausted). Complete exploration of a neighborhood is often unapproachable, therefore some modification of this generic scheme has to be adopted. Depending on the movement operator, the neighborhood varies and so does the manner of exploring the search space, simplifying or complicating the search process as a result.

During the 70's, a new class of approximate algorithms appeared whose basic idea was to combine operations in a *structured* (family-like) way in a higher level to achieve an efficient and effective search of the problem landscape. These techniques are called *metaheuristics*. The term was first introduced by Glover [14], and until it was ultimately adopted by the scientific community, these techniques were named modern heuristics [28]. This class of algorithms includes many diverse techniques such as ant colony, evolutionary algorithms, iterated local search, simulated annealing, and tabu search. A survey of metaheuristics can be found in [4,15]. Out of the many descriptions of metaheuristics that can be found in the literature, the following fundamental features can be highlighted:

- They are general strategies or templates that guide the search process.
- Their goal is to provide an efficient exploration of the search space to find (near) optimal solutions.
- They are not exact algorithms and their behavior is generally non deterministic (stochastic).
- They may incorporate mechanisms to avoid visiting non promising (or already visited) regions of the search space.
- Their basic scheme has a predefined structure.
- They may use specific problem knowledge for the problem at hand, by including some specific heuristic controlled by the high level strategy.

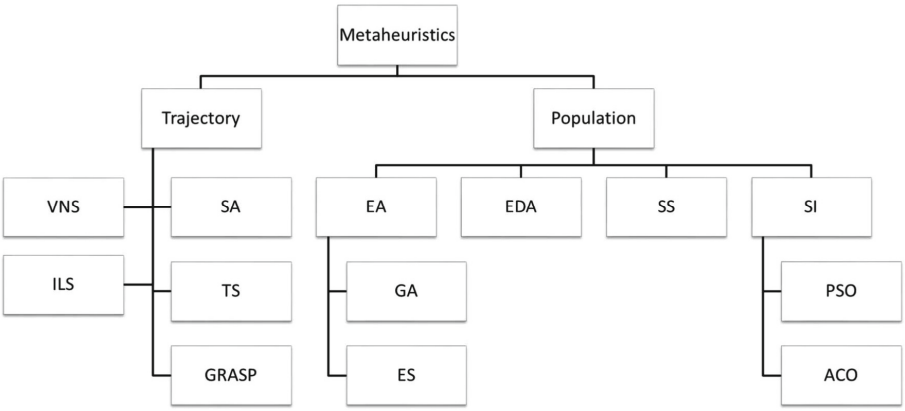

Fig. 2. Taxonomy of metaheuristics [13].

In other words, a metaheuristic is a general template for a non deterministic process that has to be filled with specific data from the problem to be solved (solution representation, specific operators to manipulate them, etc.), and that can tackle problems with high dimensional search spaces. In these techniques, the success depends on the correct balance between *diversification* and *intensification*. The term diversification refers to the evaluation of solutions in distant regions of the search space (with some distance function previously defined for the solution space); it is also known as *exploration* of the search space. The term intensification refers to the evaluation of solutions in small bounded regions, or within a neighborhood (*exploitation* in the search space).

The balance between these two opposed aspects is of the utmost importance, since the algorithm has to quickly find the most promising regions (exploration), but also those promising regions have to be thoroughly searched (exploitation). We can distinguish two kinds of search strategies in metaheuristics [4]. First, there are "intelligent" extensions of local search methods (*trajectory-based* metaheuristics in Fig. 2). These techniques add to the basic local search method some mechanism to escape from local optima (which would otherwise stuck in it).

Tabu search (TS) [14], iterated local search (ILS) [15], variable neighborhood search (VNS) [24] or simulated annealing (SA) [21] are some techniques of this kind. These metaheuristics operate with a single solution at a time, and one (or more) neighborhood structures. They are usually fast in converging to a solution (high exploitation), though suboptimal solutions are frequently found.

A different strategy is followed in ant colony optimization (ACO) [11], particle swarm optimization (PSO) [9] or evolutionary algorithms (EA) [3]. These techniques operate with a set of solutions at any time (called colony, swarm or population, respect.), and use a learning factor as they, implicitly or explicitly, try to grasp the correlation between design variables in order to identify the regions of the search space with high-quality solutions (*population-based techniques* in Fig. 2). In this sense, these methods perform a biased sampling of the search space. They tend to make a good exploration of the search of potential solutions to the problem, but a slow final tuning towards the optimal one.

Of course, a combination of trajectory and population-based techniques is in order, and thus the design of new metaheuristics is a healthy line of research. Using theoretical results to build algorithms is today a way of avoiding any useless wandering in the infinite set of combinations, either by analyzing the components of the algorithms [29], the search space [7] or the way in which the technique is expected to work versus how it actually works [22].

Finally, metaheuristics are not black boxes nor general recipes to success. Researchers need to know them, go in deep on their working principles and extract the most of their power by understanding how they search for the optima. In order to extend the basic families for difficult applications, it is very common that researchers need to learn on multiobjective optimization [10], parallel structured models [1], and combinations to machine learning [6,23].

3 Search for Software and Software for Search: SBSE and SAAL

In this section we briefly describe some existing ways of profiting from metaheuristics to face a variety of problems in SE. In all cases, the first step consists in describing the problem in a precise and quantitative way, as expected in any engineering domain. This usually means to define a global optimization problem in terms of objective functions and some type of constraints [8,18].

From the seminal work introducing the name for this field [17] the leading "S" has been read as "Search". We however think that, as of today, this term should be broaden, so as to talk on search, optimization, and learning. As a consequence, we propose here the term "SOL" as the modern meaning for the leading "S" in the term SBSE. Indeed, machine learning tools (like clustering, predicting, data science, probability analysis...) are ready normal in the field, as well as many applications put the stress in the optimization process instead of in the search technique used. Whether "SOL-Based Search Engineering" will become popular or not depends on the community of researchers in next years.

The first obvious goal, and the base for SBSE, is then solve problems in SE by using SOL algorithms. In addition, at the end of this section, we will give a twist on a different perspective were software knowledge is used to improve SOL algorithms, a not so well-known task that we here dare to name for the first time here as *Software Aware ALgorithms* (SAAL).

To finish this initial introduction to the section we will quickly review the field at an international level. To this end, we have computed the world questions on this topic from 2004 to now (according to Google Trends). In Fig. 3 we show the results (percentage of representative queries on the term SBSE). It can be seen that in years 2006–2007 a few important peaks of interest existed, that where later repeated in 2008–2009, and with a lower intensity from 2010–2011 and later years. As of today, the term and domain seems to attract a moderate attraction, though here we cannot judge because these are just percentages.

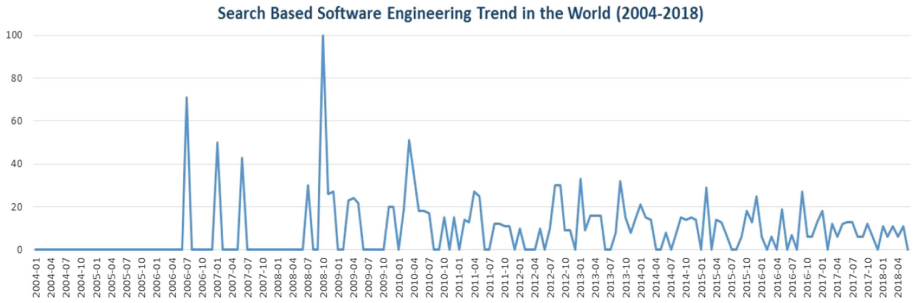

Fig. 3. Search-based Software Engineering in the world: interest between 2004 and 2018 according to Google Trends.

In Fig. 4 we include the relative interest raised from 2004 to 2018 but this time including metaheuristics to have a comparative picture. It seems that the algorithmic domain is far more developed, since the trend is always well above SBSE. This is just to say that there is more to be gained in SBSE by taking algorithms and solutions to SBSE till this field is fully developed at international level. In a (non shown here) comparison of SBSE to the whole domain of software engineering the picture shows a much larger interest for software engineering and SBSE is almost not noticeable (visually it is a plain zero-like line on axis X), a trend that we all should have to change by providing breakthrough results and a more intense dissemination in the SE community.

3.1 SOL-Based Software Engineering

Let us begin with the benefits of using SOL in general, and metaheuristics in particular for software engineering. We can list them here:

- SE Problems will be precisely defined and thus quantification and numerical comparisons are possible. Though it might look as a minor benefit, it is a

paramount one, since SE has been playing around for its whole life (and still now, in many parts of the world) with fuzzy ideas of "flexibility", "power", and other supposedly attractive features that have never been quantified nor compared in a fair way between products and services in the field.

- Software artifacts, like instructions, variables, objects, classes, functions, and many others get now a precise numerical definition and manipulation. Metrics can be defined, and advances can be objectively stated. This is in the base of the very definition of Science and Engineering, and metaheuristics give the intelligent and automatic manner of evolving, testing, designing, and understanding them.
- Scalability is another important benefit here. No more solving problems of one page of code, since actual software systems have millions of lines of code. The use of parallelism, cloud computing, and the fundamentals like numerical efficiency, robustness, and SOL theory work together to offer a unique domain of tools for the researcher and practitioner.
- Generality of studies, knowledge, and results. Indeed, metaheuristics can be applied to any SBSE problem. Investing time in them is worth, since you learn for future applications as you solve your current one. Also, getting surprising results on apparently different activities like how to prioritize software tests/tasks, assign persons to software projects, and dealing with the complexity of the execution flow, is only possible because of them, and offer a unique unified point of view. Your time in learning and using metaheuristics will pay you back for sure.
- The understanding on the problem and its context, and the solutions you get (e.g., by using genetic programming or constructive algorithms like ACO) allows knowledge accumulation. While other techniques are obscure in the solutions they provide (notably neural networks) metaheuristics offer you a white box access to solutions and even to the process by which you got them, an appreciated feature for any tool in Science.

The following sensible question is now: where can we use and find these benefits in software engineering? Well, the list will never be comprehensive, but

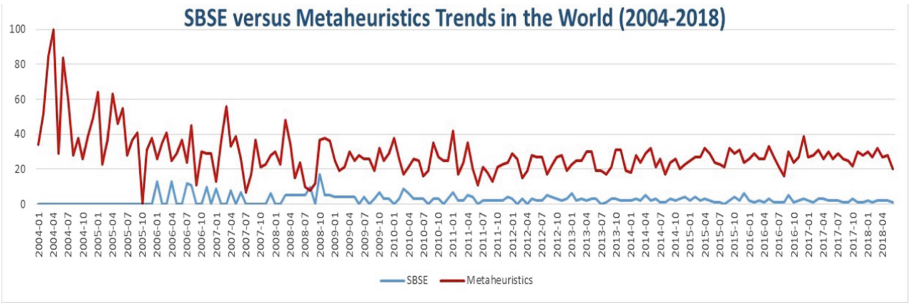

Fig. 4. SBSE and metaheuristics in the world: interest between 2004 and 2018 according to Google Trends.

here we have some of the major fields endorsed because of the utilization of SOL techniques:

- Software Planning
- Requirement Analysis and Design
- Coding Tools and Techniques
- Testing, Validation, Verification, and Debugging
- Distribution, Maintenance, Enhancement
- General Management of Software Products and Services
- Concurrent Systems
- Networks and Critical Systems
- Human Decision Making
- Evolution of Interfaces
- Web Services
- Web Ontologies
- Evolution of Architectures

There are applications of difficult classification, like the domain of *quantitative quality* [5], that usually touches many sub-fields. Also, the list does not inform on the relative importance in terms of attention received in research, and thus, e.g. the reader should be aware of the massive number of scientific articles on the different flavors of testing compared to the rest of research lines.

Before we close this first SBSE section, let us mention some high level hints for researchers. Of course, this is an informal exercise, so others could think differently. The first key factor in a good design of metaheuristics for SE is the decision on whether or not we need non-traditional representations (trees, graphs, Petri nets, automata...) that could suit the problem much better than a simple vector of symbols. The second key factor is the definition of a smart fitness function: we are not forced to use any particular function, so let us select (devise!) one with nice properties for the induced landscape, so as to ease the location of the optima. And finally, the third key factor is to use specialized operators in the algorithms, since the basic ones (one point crossover, immediate neighborhood search, arithmetical operations) are far from being good to find a solution for complex problems, where efficiency and efficacy are mandatory. The more the problem knowledge used, the better.

There are of course more hints, but this could become and endless list, so let us summarize in a brief manner some guidelines in a nutshell, organized according to what you need:

- According to the **representation**, some advices are those: if it is binary then try SA, GA, CHC, EDA or ILS. If it is a tree, go for GP. If you have a float representation go for ES, PSO, DE or CMAES. If it is a permutation representation, try to use GAs with special operators for permutations, or VNS and specialized algorithms for your problem.
- If your search problem has an underlying **graph** structure, go for ACO.
- If you face a very **expensive fitness function**, then use parallel metaheuristics. Also, use theory to lean the algorithm. Finally, try to use a kind of surrogates (like e.g. neuronal networks) for the fitness calculation.

- If your environment/function is **dynamic**, go for cellular GAs, PSO, ES or SA. Distributed versions of metaheuristics also pay off. In all cases, using an explicit memory on past experiences and a strong exploration component (e.g. hypermutation) is in place.
- If your problem is **multimodal** then go for structured EAs like cellular EAs or distributed EAs (nothing to do with parallelism, this is on structuring the population).
- If **constraints** are the hard part of your problem, then try to build hybrids (memetic algorithms?) and use specific operators considering them; search in the contour of the feasible regions defined by the constraints, optimal solutions often live there.

We stop here, though we could go on in talking on the means for a proper *initial population* (seeding with latin squares, Voronoi initialization, ad hoc sets of representative individuals merged with random ones...), the way in which the *random numbers* are generated (beyond `random`, `random48` standard methods, like using white noise from atmosphere or at least using Meresenne twister), and details like whether a new solution is accepted if its fitness is *equal* to the best existing one (yes, do accept it to scape from plateaus!).

3.2 Software Aware Algorithms (SAAL)

We now will discuss in a brief manner a kind of very related topic, but a non well-known one. The very nature of algorithms describes them as procedures to solve a problem, and then no mention to computer programs is often done. In this sense, algorithms are developed by mathematicians and other researchers whose goal is to develop the abstract technique and think in their *theoretical* behavior and complexity. However, for the vast majority of situations in modern research, algorithms need to be implemented in a computer, and thus they will enter the realm of *programming artifacts*. As any other software, algorithms running on a computer use data structures, flow-control sentences, and have a wide variety of details on how to best implement them.

The research questions here are many: are SOL algorithms well/correctly implemented? Is there a best way to do so? What data structures should we use? If using object orientation, what is the best way to define the system of classes? Are we paying attention to compiler directives and operating systems? ...and a long etcetera.

It may seem that these questions are marginal, but think in the following problems if we dismiss them:

Implementing the idea. Like for any software, we will always have a reasonable doubt that the implemented algorithm is the designed abstract algorithm. In fact, SOL algorithms are seldom having a specification for guiding a later implementation. Errors and defects might appear, taking the actual runs far from their expected behavior. Since metaheuristics are non deterministic techniques, it can be hard to know whether a bad or unexpected

results is due to the actual behavior of the correct algorithm or to the wrong implementation of the correct algorithm.

Representation of vectors. Since we will have thousands and millions of temporary variables containing the bits/integer/float variables, using high level data structures could lead to an unaffordable management of memory, either using too much of it and thus preventing it from running, or making the algorithm very slow. Some applications could even need to pack variables into basic data types like integers, with an additional time cost of packing/unpacking for their interpretation.

Object orientation. Many important issues appear when using OO for implementing metaheuristics: should we explicitly call the garbage collector? Static objects are managed far more efficiently than normal ones, so should we define static data structures as often as possible? Are operators part of the class of the individual or part of the algorithm?

Compiler directives. Different languages compile (or interpret) according to explicit directives that the researcher can use to improve efficiency. Should we use `-O3` in `gcc`? What is the best way to optimize a compiled Java program? Should we compile for the processor architecture? When making one run becomes a task of one week (or we aim real time response), this is relevant.

Variables. Many guidelines on programming exist, and most are up when implementing metaheuristics. A first question is whether researchers are following them or not...A paramount issue is for example on using global variables, something not good for the quality (understanding, error analysis, extensibility) of a program, but at the same time often more efficient than using local variables. Finding the right balance is difficult in most languages, do we even care about this? If not, then why to bother on not using goto jumps, raising/catching exceptions or giving meaningful names to variables? Why should we only take arbitrary best practices? Let us take them "all".

Toolboxes versus languages. Many researchers select their implementation tools so as to only focus on the application, and this means going into *closed* packages like MATLAB, SPSS and similar ones. The many advantages (all is inside, the researcher already knew them, no need of advanced programming skills) sometimes do not compensate for the long running times, the many unknown internal calls to intermediate procedures, and the lack of extensions available in them (parallelism, for example). General purpose programing languages (Java, C, C++, Python, R, Javascript...) allow much more freedom (and problems) for implementation. Of course, every language also has a baseline speed that needs to be considered when starting a new project.

After reading the previous list, any sensible researcher should be now concerned on whether he/she is actually able of getting much more from his/her present implementation. Here an intriguing question: what is the actual quality of the many libraries we all are using with no formal proof that they are implementing the actual expected algorithms and (at least) show a reasonable efficiency? We take software as a black box and apply it, take correctness for granted, and then try to explain results that could be unexplainable due to software errors. Also, we might be missing tons of new research lines in not making

our algorithms aware of the operating system, not to mention being aware of the energy consumption (for running algorithms e.g. in smartphones) or the underlying hardware (processor power, available memory, network communications).

In summary, it seems weird that the entire world is developing software for search, optimization and learning while not using for this the well-known SE tools and knowledge to do so. In the domain of SBSE, it is especially weird that researchers are dismissing this in their papers and public libraries. The reader might think that this is just a personal tick or some kind of luxury task, however research results show how you can save 30% or more time with a good implementation [25]. This time can be used for additional runs, a deeper understanding, or to write more scientific papers per year. Indeed, efficiency is just part of the quality of any software package; other factors like maintainability, reliability, extensibility, usability, also apply to public SOL software, and thus should be relevant in future studies.

4 Open Challenges and Ideas for the Future

In this section we will find an analysis of the potential challenges and good directions for the interplay between SE and metaheuristics in the future. They will be presented in two parts: a first set of challenges for SBSE (short and long term), and some final challenges for SAAL. Of course, mentioning challenges is not a formal scientific task, so this is clearly subject to debate. Indeed, since the whole existing community, plus the newcomers to arrive, will be deciding on how and what is important as a future challenge, this is just an exercise of good guessing to help interested readers.

Let us start with potentially interesting SBSE challenges. We can there define the following ones in the short term:

- Stopping Criteria
  Analyze the techniques both under a predefined effort (to later evaluate the resulting fitness), solution quality (to later evaluate the needed fitness effort), study the algorithm convergence (phenotype and genotype ones), and explore combined stopping condition and their effect on the results. Limited budget, for example, are of great interest to set the interest of any new technique. Also, what is the meaning of the results for the SBSE problem?
- Landscape Visualization
  Knowing the problem you are solving is very important to design a good algorithm. Visualization of search landscapes, of the work of search operators, or of the contents of the resulting solution is very important. Looking for new tools like *parallel coordinates* [19] or *local optima networks* [26] could be worth.
- Landscape Characterization
  We really need to know, visualize and use the properties of the search landscape of our problem: multimodality, epistasis, discontinuities, pla-teaus, etc. Metrics encoding this, such as the number of suboptimal

solutions, or the fitness-distance correlation, are of much interest to allow the designer to arrive to a good algorithm design.

- Human Competitive Results
  Having an intelligent technique to beat a human is difficult, we indeed should go for specialized sub-domains and very concrete tasks, and then computer aided software management would become universal. To this end, programming interfaces, testing GUIs, and web services to assess software packages, are interesting and expected for all of us.

Let us now shift to a short discussion on potential long term challenges:

- Multiobjective Optimization
  There is a rich set of information in the metaheuristic field: goals, algorithms, metrics, statistics are available to study and use for SE. Decision making (versus optimization) is a growing field also here, and higher level tools are wanted to help innovation in SBSE. It seems that this is presently happening, but when going to the details of existing articles, they can be classified as out of the main stream in multiobjective optimization, because of their lack of basic contents in the paper in this sense.
- Interactive Optimization
  Humans work better with visual concepts, and then a better relation to GUIs is expected to appear, so that we approach the industry of software with usable tools embedding our SBSE solutions inside.
- Prediction
  The software industry (well, we all) need an assisted prediction of what is going to happen, so that we can react in advance and have a global better management of product and services. Predicting how and why we need a next release of software packages, where will be hidden the next defect, error or failure (not the same!) in software, the future status of a project, etc. are all very welcomed SBSE outputs.
- Hybrid SOL Algorithms
  SBSE is a field where a natural handshaking between exact and approximate algorithms is happening. Also in a natural way, researchers should build hybrid techniques with existing knowledge, wherever it is coming from. In this sense, personal preferences for a technique just create an unwanted bias: let us use the best techniques for the (sub)problem at hands and combine them to make more powerful ones.
- On Line (Dynamic) Optimization
  Up to now, most SBSE problems are static, that is, the target function to optimize behaves the same along the whole run of the algorithm. However, for more complex problems in databases, shared programming environments, testing, etc. could need to rely on the existing knowledge on *dynamic optimization problems* (DOP), like adding memory to our algorithm, use hypermutation for a enhanced exploration, and relying to self-adaptation for improved performance.

- Applying New Models of Search
  SBSE researchers know a great deal of SOL algorithms, but this last domain is so large and evolves so quickly that they should be continuously looking for new trends, like running algorithms on GPUs, programming for smartphones to execute solvers on them, using cloud computing to deal with real-size software testing, or refactoring/repairing big software projects found in Github.

We now finish this section with some open challenges for SAAL research, where our solvers are seen as software programs to be understood and improved:

- Software Libraries
  Here, we should start designing libraries with (e.g.) extended UML tools, discuss on the best class architecture for complex algorithms, find errors in them by testing the source code, and analyze quality metrics (from specification/requirements to real usability). Taking the correctness and quality of optimization, search and learning libraries for granted is a risk that we cannot take in the domain, if we think in ourselves as good scientists and engineers.
- Data Structures
  Since there are so many variables and objects in a complex SBSE solver, we should care on how a population or an individual should be better implemented in any population-based technique. The more a data structure is used in our solver, the deeper the analysis on its memory and CPU consumption should be addressed.
- Profiling
  Once our solver is programmed, or while it is being improved, profiling is a must. Gathering information on the solver components, on how executions are done, and proposing a better implementation, all come handy to make better and more efficient research. Of course, after the profiling analysis we should act, and thus review the design and the implementation of our software before putting it to solve a SBSE problem.
- Program Complexity
  It is very common that papers lack of any information on the computational complexity of their solutions. As so, many solvers published today have a low impact in research. Characterizing program complexity of well-known techniques is a must, to know whether we should lose our time in learning them or not, because they would not work even in similar slightly larger conditions of problem size. Something similar happens with the parameterization used to solve a problem: the robustness (sensitivity) of the parameters as implemented in the solver has to be assessed by researchers in every research paper.
- New Frontiers
  There are many goals that researchers need to consider if aiming a wide impact: deal with programs having million of lines, going for complete Github repositories as a routine task, create SBSE tools for software companies, etc. Besides, lots of specialized information are needed in SBSE

before going to implement parallel solvers, properly analyzing multiobjective approaches (yes, this can not be done in one month), target drivers and operating systems as the goal of research, apply data science to software, and a plethora of points of view to come.

5 One Conclusion, Many Future Works

This article is a summary of the talk delivered at SSBSE 2018, what in turns is a fresh view of the talk offered in the very first SSBSE 2009 in London. The topic is a perspective on how metaheuristics can help researchers and practitioners in software engineering. In this sense, this is not a normal research paper, but a position paper showing a filtered vision on the last years in this domain, according to the opinion to the author. Of course, others can think differently.

Our main conclusion here is a positive one: SBSE is a healthy domain and getting more specialized every day. There is much to find in metaheuristics to help SBSE researchers define SOL problems in the domain of software engineering, so as to quantify and give fair numerical analyses in a field where industry and even academia have neglected a bit the engineering part: quantify, understand, manage, optimize. That should be normal in any engineering activity [12], and thanks (in part) to SBSE, this is happening.

We have summarized many types of metaheuristics to help readers, as well as gave some hints to guide newcomers. We have introduced the idea of changing "search" by "search, optimization, and learning" as a new meaning for the first "S" in "SOL-Based Software Engineering". We do think that more than search is happening, so such a re-interpretation of the leading "S" comes in place. We even dared to introduce for the first time the concept of going on the other way, and using software engineering concepts, tools and research for building algorithms: SAAL. Much can be gained if we all consider going that way in a structured manner, and some existing works point to the right direction.

A big deal of cross-fertilization is needed between SOL and SE. Challenges are many in scientific areas like the algorithm types (many-objective, dynamic, uncertain problems) as well as on the new definition of problems (complex modeling of real world SE tasks) and the overall needed set of best practices for researchers, something that should come as soon as possible to avoid meaningless studies showing, sometimes, minor/incremental contributions to the community. Being able of understanding and reading results in the two domains will lead sooner than later to a completely new/separate body of knowledge for this field, with a potential huge impact in today's economy and science.

References

1. Alba, E.: Parallel Metaheuristics: A New Class of Algorithms. Wiley, Hoboken (2005)
2. Alba, E., Troya, J.M.: Genetic algorithms for protocol validation. In: Voigt, H.-M., Ebeling, W., Rechenberg, I., Schwefel, H.-P. (eds.) PPSN 1996. LNCS, vol. 1141, pp. 870–879. Springer, Heidelberg (1996). https://doi.org/10.1007/3-540-61723-X_1050

3. Bäck, T., Fogel, D.B., Michalewicz, Z. (eds.): Handbook of Evolutionary Computation. Institute of Physics Publishing Ltd., Bristol (1997)
4. Blum, C., Roli, A.: Metaheuristics in combinatorial optimization: overview and conceptual comparison. ACM Comput. Surv. **35**(3), 268–308 (2003)
5. Boehm, B.W., Brown, J.R., Lipow, M.: Quantitative Evaluation of Software Quality. In: Proceedings of the 2nd International Conference on Software Engineering (ICSE 1976), pp. 592–605. IEEE Computer Society Press (1976)
6. Calvet, L., De Armas, J., Masip, D., Juan, A.A.: Learnheuristics: hybridizing metaheuristics with machine learning for optimization with dynamic inputs. Open Math. **15**, 261–280 (2017). https://doi.org/10.1515/math-2017-0029
7. Chicano, F., Ferrer, J., Alba, E.: Elementary landscape decomposition of the test suite minimization problem. In: Cohen, M.B., Ó Cinnéide, M. (eds.) SSBSE 2011. LNCS, vol. 6956, pp. 48–63. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-23716-4_7
8. Clark, J.A., et al.: Formulating software engineering as a search problem. IEE Proc. Softw. **150**(3), 161–175 (2003)
9. Clerc, M.: Particle Swarm Optimization. Wiley, Hoboken (2010)
10. Coello Coello, C., Lamont, G.B., Van Veldhuizen, D.A.: Evolutionary Algorithms for Solving Multi-Objective Problems. Springer, New York (2007). https://doi.org/10.1007/978-0-387-36797-2
11. Dorigo, M.: Optimization, learning and natural algorithms. Ph.D. thesis, Politecnico di Milano (1992)
12. Fenton, N.E.: Software measurement: a necessary scientific basis. IEEE Trans. Softw. Eng. **20**(3), 199–206 (1994)
13. Ferrer, F. J.: Optimization techniques for automated software test data generation. Ph.D. thesis, Universidad de Málaga (2016). https://riuma.uma.es/xmlui/handle/10630/13056. Accessed 25 June 2018
14. Glover, F.: Future paths for integer programming and links to artificial intelligence. Comput. Oper. Res. **13**(5), 533–549 (1986)
15. Glover, F.: Handbook of Metaheuristics. Kluwer, Dordrecht (2003)
16. Harman, M., Afshin Mansouri, S., Zhang, Y.: Search-based software engineering: trends, techniques and applications. ACM Comput. Surv. **451**, 1–64 (2012)
17. Harman, M., Jones, B.F.: Search-based software engineering. Inf. Softw. Technol. **43**(14), 833–839 (2001)
18. Harman, M., Jones, B.F.: Software engineering using metaheuristic innovative algorithms: workshop report. Inf. Softw. Technol. **43**(14), 905–907 (2001)
19. Inselberg, A.: Parallel Coordinates: Visual Multidimensional Geometry and Its Applications. Springer, New York (2009). https://doi.org/10.1007/978-0-387-68628-8
20. Jones, B.J., Sthamer, H.-H., Eyres, D.: Automatic structural testing using genetic algorithms. Softw. Eng. J. **11**, 299–306 (1996)
21. Kirkpatrick, K., Gelatt, G.D., Vecchi, M.P.: Optimization by simulated annealing. Science **220**(4598), 671–680 (1983)
22. Luque, G., Alba, E.: Math oracles: a new day of designing efficient self-adaptive algorithms. In: Proceedings of GECCO (Companion), pp. 217–218 (2013)
23. Memeti, S., Pllana, S. Binotto, A., Kolodziej, J., Brandic, I.: Using Metaheuristics and Machine Learning for Software Optimization of Parallel Computing Systems: A Systematic Literature Review. arXiv:1801.09444v3 [cs.DC], https://doi.org/10.1007/s00607-018-0614-9 (2018)
24. Mladenovic, N., Hansen, P.: Variable neighborhood search. Comput. Oper. Res. **24**(11), 1097–1100 (1997)

25. Nesmachnow, S., Luna, F., Alba, E.: An empirical time analysis of evolutionary algorithms as C programs. Softw. Pract. Exp. **45**(1), 111–142 (2015)
26. Ochoa, G., Veerapen, N.: Mapping the global structure of TSP fitness landscapes. J. Heuristics **24**(3), 265–294 (2018)
27. Osman, I.H., Laporte, G.: Metaheuristics: a bibliography. Ann. Oper. Res. **63**, 513–623 (1996)
28. Reeves, C.R. (ed.): Modern Heuristic Techniques for Combinatorial Problems. Wiley, Hoboken (1993)
29. Villagra, A., Alba, E., Leguizamósn, G.: A methodology for the hybridization based in active components: the case of cGA and scatter search. Comput. Int. Neurosci. **2016**, 8289237:1–8289237:11 (2016)