Assessing Single-Objective Performance Convergence and Time Complexity for Refactoring Detection

David Nader-Palacio
Universidad Nacional de Colombia
Bogota, Colombia
danaderp@unal.edu.co

Daniel Rodríguez-Cárdenas
Universidad del Rosario
Bogota, Colombia
danielh.rodriguez@urosario.edu.co

Jonatan Gomez*
Universidad Nacional de Colombia
Bogota, Colombia
jgomezp@unal.edu.co

ABSTRACT

The automatic detection of refactoring recommendations has been tackled in prior optimization studies involving bad code smells, semantic coherence and importance of classes; however, such studies informally addressed formalisms to standardize and replicate refactoring models. We propose to assess the refactoring detection by means of performance convergence and time complexity. Since the reported approaches are difficult to reproduce, we employ an Artificial Refactoring Generation (ARGen) as a formal and naive computational solution for the Refactoring Detection Problem. ARGen is able to detect massive refactoring's sets in feasible areas of the search space. We used a refactoring formalization to adapt search techniques (Hill Climbing, Simulated Annealing and Hybrid Adaptive Evolutionary Algorithm) that assess the *performance* and *complexity* on three open software systems. Combinatorial techniques are limited in solving the Refactoring Detection Problem due to the relevance of developers' criteria (human factor) when designing reconstructions. Without performance convergence and time complexity analysis, a software empirical analysis that utilizes search techniques is incomplete.

CCS CONCEPTS

• **Mathematics of computing** → *Combinatorial optimization*; *Mathematical software performance*; • **Computer systems organization** → *Maintainability and maintenance*;

KEYWORDS

Combinatorial Optimization, Mathematical Software Performance, Refactoring, Software Maintenance

ACM Reference Format:
David Nader-Palacio, Daniel Rodríguez-Cárdenas, and Jonatan Gomez. 2018. Assessing Single-Objective Performance Convergence and Time Complexity for Refactoring Detection. In *GECCO '18 Companion: Genetic and Evolutionary Computation Conference Companion, July 15–19, 2018, Kyoto, Japan.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3205651.3208294

*Associate Professor Universidad Nacional de Colombia.

GECCO '18 Companion, July 15–19, 2018, Kyoto, Japan

ACM ISBN 978-1-4503-5764-7/18/07…$15.00
https://doi.org/10.1145/3205651.3208294

1 INTRODUCTION

Providing refactoring recommendations is a widespread practice that assists developers in ameliorating the maintainability and readability of their code [4]. Nonetheless, the procedure for automatically generating sequences composed of refactoring operations or the Refactoring Detection Problem (RDP) remains a complex [12] and arduous task for maintenance groups [13, 14].

Approaches that suggest refactoring operations must be very clear on how those refactorings are generated. Indeed, each approach must give a sound, concise and justified answer to the following questions: Which are the variables, the hyper-parameters, and the constraints of the model? How are the refactorings built and performed? Is the refactoring detection a truly multi-objective problem? Unfortunately, current research [16–18, 21, 22, 24–26] proposes informal optimization models for the RDP without proper performance convergence and time complexity analysis, making the experiments difficult to compare. The implementation and execution of refactorings from the reported models are ambiguous. For instance, Ouni *et al*'s "DefectCorrection" algorithm did not clarify how the refactoring sequences were computed on the code to assemble final recommendations [23]. Therefore, performance convergence and time complexity analysis are required to empirically evaluate search-based techniques [5, 8].

In this paper, we employ the Artificial Refactoring GENeration (ARGen), an approach for detecting massive refactoring operation sets. ARGen allows researchers to *estimate* the impact of proposed changes to the source code before such changes are implemented [1] We analyze the convergence and time complexity of ARGen with two baseline single optimization techniques and a Hybrid Adaptive Evolutionary Algorithm (HaEa) [8]. Eventually, the research is expected to contribute to the empirical analysis of the application of single-objective techniques in software maintenance problems.

2 A REPRODUCIBILITY ANALYSIS OF THE REFACTORING OPTIMIZATION MODELS

The reproducibility analysis -whose purpose is to manually verify the limitations of automatically performing software refactoring- consisted in assigning a value to each reproducibility characteristic (or dimension traced a conceptual matrix[2]) for every reported model. The models were then ranked: zero points indicates the model was **easy** to reproduce; one or two points **moderately hard** to reproduce; three or more than three points **hard** to reproduce. If the behavior preservation dimension was unclear, the approach

[1]Regarding the impact on the research community and industry, all the artifacts of the project are available for any researcher or developer who requires performing refactoring analysis: https://github.com/danielrcardenas/argen.
[2]Find the complete table in: https://argeneration.github.io/assets/pdfs/longtable.pdf

was labeled as *hard to reproduce* since the key characteristic of any refactoring is the ability to preserve the functionality. From Table 1, we conclude the search-based models fail in explaining the way of reproducing refactoring techniques (no evidence of a real refactoring outcome). Moreover, none of the papers perform algorithm analysis convergence, except for Seng, et al. [29].

3 REFACTORING DETECTION DESCRIPTION

Sets of classes, methods, fields and refactoring operations [3] are properly represented to be utilized in a combinatorial framework besides the metrics [4] utilized in the objective function. A formalism for refactoring is appropriated whether the research community desires to extend any technique, enclose other formalism in the framework (e.g., applying a hyper-heuristic or memetic algorithm for the software refactoring problem [3, 28]), or represent the refactorings in a data structure different from sequences (e.g., Directed Acyclic Graphs).

3.1 Formal Definitions

Object-oriented programming (OOP) is a software development style organized around objects, which can be used in paradigms like imperative, functional, or logic programming [19]. An object is defined as a run-time entity composed of 1) a state represented by a set of internal objects named attributes and 2) a set of subroutines named methods [27].

Definition 3.1. (System Under Analysis definition). SUA is a software system composed of classes, methods, and attributes, where objects instantiate classes and communicate with each other through messages.

Definition 3.2. (Class definition). A class, prototype or template is a set of similar objects including collections of similar attributes and methods. In fact, classes are the programming-time counterparts of objects [27]. Assume a single class c_α is a Cartesian product represented by $c_\alpha = identifier \times Attribute(s) \times Method(s) = str \times str^* \times str^*$, where str^* is a Kleene on str string abstraction or chain. A set of classes is a power set of c_α: $C \subseteq \wp(c_\alpha)$ and $|C| = k$, where k is the total number of System Under Analysis (SUA) classes and $c = \{c_\alpha \in C | 1 \leqslant \alpha \leqslant k\}$.

Definition 3.3. (Method Definition). Methods are subroutines which define the set of valid messages the object may accept. Methods are represented by Kleene star. Set $M = str^* = \bigcup_{n=0}(str)^n$ and $\forall_{1 \leqslant \alpha \leqslant k}$ states $|M| = \beta_\alpha$ is finite. So $M \subseteq \mathbb{N}$ and $M(c_\alpha) = M_\alpha$.

Definition 3.4. (Attribute Definition). Attributes (or fields) are sets of internal objects that represent a state. Attributes are represented by Kleene star. Set $A = str^* = \bigcup_{n=0}(str)^n$ and $\forall_{1 \leqslant \alpha \leqslant k}$ states $|A| = \gamma_\alpha$ is finite. So $A \subseteq \mathbb{N}$ and $A(c_\alpha) = A_\alpha$.

Definition 3.5. (Refactoring Definition). The refactoring process consists in re-constructing the code design of a SUA without affecting the behavior functionality [4]. A refactoring is a function $R_\delta : \Omega \longrightarrow$ (Code Modification) where δ represents a specific refactoring operation and $\Omega = c_s \times A_s \times M_s \times c_t$ is a *Cartesian product* parameter.

[3]Find the list of refactoring operations in: https://argeneration.github.io/assets/pdfs/RefRefactors.pdf

[4]Find the list of metrics in: https://argeneration.github.io/assets/pdfs/RefMetrics.pdf

Refactoring properties are: 1) a solution set of all the possibilities of refactorings functions with refactoring parameters, coined as $R(\Omega)$, so $RI \in R(\Omega)$, is a specific solution or Refactoring Instance, 2) a refactoring recommendation S_i is a set of Refactoring Instances RI and a subset of $R(\Omega)$.

Definition 3.6. (Code Quality Metric Definition). A quality metric is a function $\eta_j : c_\alpha \longrightarrow \mathbb{R}$. Each class c_α of the *SUA* has a set of metric values: $H_\alpha = \{\eta_1(c_\alpha), \eta_2(c_\alpha), \ldots, \eta_J(c_\alpha)\}$, where $\mathcal{J}$ is the total number of metrics and $H_\alpha \in \mathbb{R}^J$.

We utilized Refactoring Impact PrEdiction (RIPE) to predict the impact of 12 refactoring operations on 11 code metrics [2] because our solution set of refactoring recommendation is too expensive to perform in a SUA, yet an *estimation* of the refactoring on the metrics can be computed. RIPE implements 89 impact prediction functions that show developers the variation of code metrics before the application of a refactoring operation. For instance: $LOC_p(c_s) = LOC_b(c_s) - LOC(m_k)$ and $LOC_p(c_t) = LOC_b(c_t) + LOC(m_k)$. For each Refactoring Operation r_δ and quality metric η_j there is a prediction function $Prediction_{\delta,j}(c_\alpha) = \widetilde{\eta}_{\delta,j}(c_\alpha)$, which *estimates* the impact of the refactoring on the metric of a class (iff the refactoring affects the metric).

Definition 3.7. (Impacted Code Quality Metric Definition.) An impacted quality metric is a function $\widetilde{\eta_{\delta,j}} : c_\alpha \longrightarrow \mathbb{R}$. Each class $\widetilde{c_\alpha}$, impacted by a refactoring operation r_δ of the *SUA*, has a set of metric values: $\widetilde{H_\alpha} = \{\widetilde{\eta}_{(\delta,1)}(c_\alpha), \widetilde{\eta}_{(\delta,2)}(c_\alpha), \ldots, \widetilde{\eta}_{(\delta,J)}(c_\alpha)\}$, where $\mathcal{J}$ is the total number of metrics and $\widetilde{H_\alpha} \in \mathbb{R}^J$.

3.2 Refactoring Detection as a Combinatorial Problem

The Artificial Refactoring Generation (ARGen) considers all possible combinations of refactoring operations for a given SUA, which implies to operate in a large solution space. The purpose of the technique is to recommend massive artificial refactoring operations that fulfill the proposed parametric objective function.

The refactoring generation is a *NP-Complete combinatorial problem* [5]. The solution constitutes a *set* of refactoring operations instead of a *sequence* of refactoring operations. The refactoring operations from the same set are independent of each other. ARGen does not guarantee generated outcomes reduce the error-proneness of an inspected piece of code (or SUA). Nevertheless, ARGen is able to explore the search space in actionable areas, if and only if, the developers/researchers properly tune the objective function based on their interest in SUA.

3.2.1 Characterization of the Search Space. Figure 1 depicts the search space of the Artificial Refactoring Generation. The *feasible region* represents refactoring that fulfill defined constraints of the proposed objective function. The *actionable region* represents refactoring solutions that reduce error-proneness. The search space increases by the expression $(C^2 * A * M)^r$, where r is the number of refactoring operations in a set (e.g., a system with 10 classes, 10 attributes, and 10 methods would have a size of 10,000 if the set is composed of one refactoring).Our paper demonstrates that

[5]The ARGen was demonstrated through the Subset Sum problem computational complexity analysis (Subset-Sum-Problem $\propto$ ARGen)

Table 1: Comparison of Combinatorial Techniques Related to The Refactoring Problem

Authors	Google Scholar Citation Count	Optimization Technique	Objective(s)	Type of solution	Evaluation Method	Behavior Preservation	Reproducibility
Seng & Stammel (2006) [29]	232	Single Objective EA (non-described technique).	Maximize features such as complexity, stability, coupling and cohesion by means of a weighted sum of seven metrics values. The input is the source code.	A list of model refactorings. The list is ordered according to established pre/post conditions of refactorings.	Fitness convergence analysis and an example of the code development.	Domain specific preconditions based on the object oriented structure and design patterns.	Moderately hard
Keeffe, Mark O & Cinneide, Mel O (2007) [9, 20]	101	Multiple ascent hill climbing, simulated annealing and genetic algorithms.	An implementation of the unserstandability function from QMOOD model. The input is an Abstract Syntax Tree (AST).	The refactorings are applied to the AST, then the outputs are the refactored input code (unclear solution example).	Mean analysis on fitness values and execution time for each search technique.	Static program analysis on an unknown pre/post-conditions. There is no evidence of how the refactoring is executed on the AST.	Hard
Harman & Tratt (2007) [6]	173	Variant of hill climbing (non-described technique).	Two objectives: metric coupling between classes (CBO) and the standard deviation of methods per class (SDMPC).	Sequences of refactorings; although, it is unclear how the solution is configured.	Pareto front analysis; however, without concrete type solutions or outcome examples, the analysis is difficult to interpret.	NA	Hard
Jensen & Cheng (2010) [7]	52	Genetic programming.	A proposed fitness function composed of the QMOOD model, specific penalties,and number of modifications.	The individual is composed of a transformation tree and design graph (UML class diagram).	Exploratory analysis on fitness values; however, the analysis are very informal and no concrete solution is described.	NA	Hard
Ouni & Kessentini (2013) [22, 23, 25]	24	NSGA-II (it is unclear how the refactorings were executed).	Maximize design quality, semantic coherence and the reuse of the history of changes.	Vector-based representation. A set of refactorings is configured and sorted using dominance principle and a comparison operator based on crowding distance. The order of the operations inside the vector is important.	DCR (defect correction ratio) and RP (refactoring precision). It is unclear how they reproduced previous approaches. They used RefFinder (Eclipse Tool) for refactoring comparison.	Pre and post conditions described by Opdyke (it is unclear how authors performed behavior preservation).	Hard
Mkaouer & Kessentini (2014) [15, 16]	13	NSGA-II (it is unclear how authors executed the 23 refactorings).	Improve software quality, reduce the number of refactorings and increase semantic coherence.	A set of Non dominated refactoring solutions. The set of refactorings are ranked.	Execution time analysis and a manual validation of the refactorings. There are comparisons with other reported refactorings.	NA	Hard
Ouni & Kessentini (2015) [21, 26]	6	Chemical Reaction Optimization; although, there is no evidence or concrete example of how the refactorings are executed on the code	One objective that minimize the number of bad smells.	A sequence of refactorings in a vector representation (it is unclear how the appearance of the suggested refactoring is).	Execution time analysis and comparison to the number of bad code smells with an old version of the systems.	Opdyke's functions; however, these conditions are unclear in the implementation of the approach.	Hard
Mkaouer & Kessentini (2016) [18]	NA	NSGA-II (it is unclear how the refactorings were executed).	Maximize quality improvements and the respective uncertainties associated with severity and importance of refactorings opportunities.	Vector-based representation of refactoring operations. The order depends on the position of the refactoring inside the array.	Hypervolume, Inverse Generational Distance, Number of Fixed Code-Smells, Severity of Fixed Code-Smells, Correctness of the suggested refactorings and computational Time.	Opdyke's function (defining pre/post conditions): however, these conditions are unclear in the implementation of the approach.	Hard

the number of classes involved in the optimization affects the time complexity.

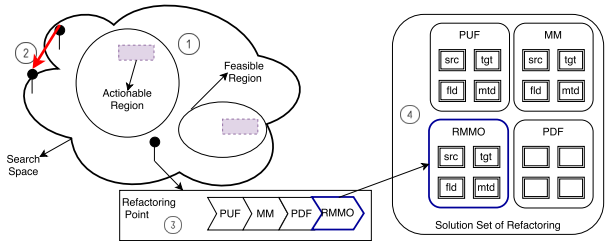


Figure 1: Refactoring Problem Search Space. The left part of the graphic is an overview of the search space. It includes two relevant regions: ① feasible and actionable and shows that the space is not convex ②. A *refactoring solution point* ③ that is a set of refactoring operations from the Fowler's Catalog. The solution point is amplified in the right part ④ where a source class (src), a target class (tgt), a field from the source class (fld) and a method from the source class (mtd) are included within each refactoring operation.

3.2.2 Objective Function. The objective function represents a ratio that measures the normalized difference between the actual metrics and the impacted metrics of the system under analysis. The inputs constitute a set of refactoring operations composed of real objects from the system (i.e target class, source class, fields, and methods) and a weighted vector w that captures the relevance of each metric.

Definition 3.8. (Code Quality Additive Metric Definition). The use of quality metrics seeks not only to identify refactoring opportunities but also to capture developers' interest. This formula computes $\Upsilon_H(j \in J) = \sum_{\alpha=1}^{k} \left(\eta_j(c_\alpha)\right)$ a general value for one exact metric on the processed *SUA*.

The parameter Φ_j of the impacted sum function $\Upsilon_{\widetilde{H}}(\Phi_j)$ is composed of a metric $j \in J$ and the solution set $S_i : \Phi_j = (j \in J, S_i) = (j, \{RI_1, RI_2, ..., RI_i, ..., RI_T\})$, where T is the total number of Refactoring Instances in S_i.

Definition 3.9. (Code Quality Additive Impacted Metric Definition). The following formula

$$\Upsilon_{\widetilde{H}}(\Phi_j) = \sum_{i=1}^{|S_i|} \sum_{\alpha=1}^{|c \in \Omega_i|} max_{1 \leq j \leq \widetilde{H_\alpha}} \left(\left(\widetilde{\eta_{\delta,j}(c_\alpha)} \right) \right)$$

estimates a general value for one specific metric on the processed *SUA* according to a solution set.

Definition 3.10. (The Bias Quality System Ratio). The Bias Quality System Ratio $BQR(\Phi_j) = \frac{\Upsilon_{\widetilde{H}}(\Phi_j)}{\Upsilon_H(j)}$ measures the ratio between the quality of the *SUA* and the predicted quality of the impacted classes following a refactoring recommendation S_i.

Definition 3.11. (The Objective Function Definition). The objective function is a parametric optimization function receives developer-defined goals. We use min-max normalization for BQR to put the metrics in a positive scale:

$$Obj(\Phi) = \frac{\sum_{j=1}^{J} \left(w_j \frac{\Upsilon_{\widetilde{H}}(\Phi_j) - min(\Upsilon_{\widetilde{H}}(\Phi_j))}{max(\Upsilon_{\widetilde{H}}(\Phi_j)) - min(\Upsilon_{\widetilde{H}}(\Phi_j))} \right)}{\sum_{j=1}^{J} \left(w_j \frac{\Upsilon_H(\eta_j) - min(\Upsilon_H(\eta_j))}{max(\Upsilon_H(\eta_j)) - min(\Upsilon_H(\eta_j))} \right)} + \rho(\Phi)$$

where w or vector of *weights* are numbers between $\{-1, 1\}$ and $\rho(\Phi)$ penalization. The vector of weights is the formalism where developers convey their interest in metrics to obtain meaningful suggestions of classes to be reconstructed [1]. When the numerator is less than the denominator, the predicted system improves its quality metrics. In other words, if the objective function is an improper fraction, then the predicted system is worse (in terms of quality) than the actual. The input of the objective function is a set Φ of refactoring recommendations; alternatively, an output is a value that represents the estimation of the metrics after applying the refactorings operations in Φ.

The objective function does not ensure refactoring opportunities in a SUA [1], yet an *estimation* of how the refactorings are impacted by the change of a quality metric [2]. The implementation of the objective function (Figure 2) required the design of computational techniques to adapt the Artificial Refactoring Generation and the definition of specific refactoring *data structures* (see figure [6] from the web page).

3.2.3 Refactoring Constraints. We pose a catalog of constraints that depend on the refactoring operation structure and the general object-oriented guidelines. Figure 3 depicts a typical configuration of a solution after applying a mutation, a variation's operator, or a new generation. Consider a class C1 a subclass of C2. If we try to apply a "Pull Up Field" on a solution where C1 constitutes the source and C2 the target, then that operation violates the constraint *SRCSubClassTGT*.

[6]Figure in: https://argeneration.github.io/assets/pdfs/RefDataStructure.pdf

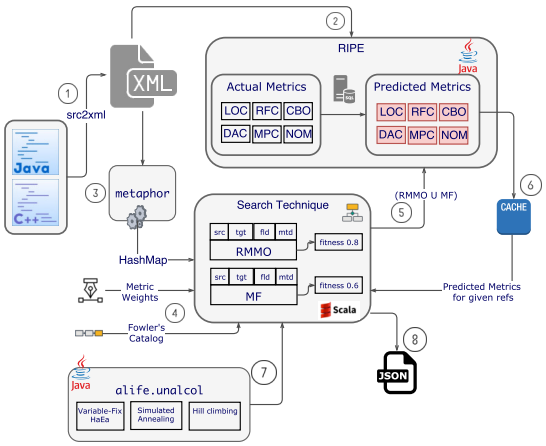


Figure 2: Refactoring Detection in the Objective Function. ① *Parser.* the source code must be in Java or C++ to be translated into an XML by means of src2xml. ② *Refactoring Impact PrEdiction.* RIPE is composed of a Java module that measures 6 metrics and another module that estimate the same metrics on the output provided by the search technique. ③ *Source Code Metaphor.* The metaphor keeps the information of the analyzed systems. ④ *The search technique.* This module is in Scala and process the vector of weights and the Fowler's Catalog using previous formalisms. ⑤ *Individual.* The search technique module proposes different refactoring configurations that RIPE must measure. ⑥ *Cache.* The outputs of RIPE are stored in a Cache to improve performance. ⑦ *alife.unalcol* This library is employed for the optimization algorithms. ⑧ *JSON recommendations.* The search technique module produces a final refactoring detection output with the best fitness mapped into a JSON format.

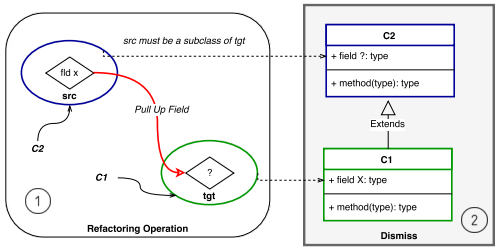


Figure 3: Refactoring Constraint Pull Up Field. In "Pull Up Field", the source class must be a subclass of the target.

3.3 Refactoring Detection as Evolutionary Approach

We employed the Hybrid Adaptive Evolutionary Algorithm (HaEa) that evolves the operator rates and the solutions at the same time [8]. Two versions of HaEa were considered: invariable and variable chromosome. A variable chromosome indicates that the individual allows modification in its size (varies between a minimum and maximum range) this paper refers to that algorithm as HaEa Var. In an invariable HaEa version, the number of refactoring recommendations are fixed.

In HaEa, each individual is evolved independently from other individuals in the population. In each generation, one genetic operator is selected according to dynamically learned operator rates that are encoded into the individual. If the selected operator requires another individual for mating, then the individual is selected from the population with replacement. HaEa replaces the parent and the operator rate is rewarded for improving the individual if the fitness of the offspring is better than the parent. Whereas, the operator rate is penalized if the individual diminishes the fitness. The source

code of this approach is available online [7]. The Hybrid Adaptive approach utilizes the repairing functions and six different genetic operators (Figure 4) that were specially made for exploring in spaces where the algorithm preserves the objected-oriented structure.

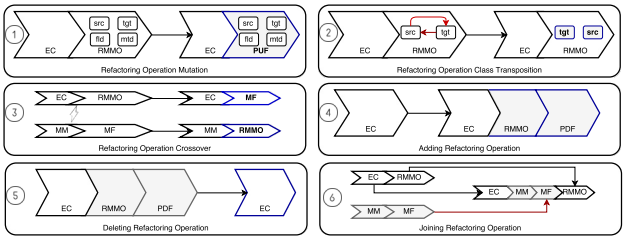


Figure 4: HaEa with invariable (fixed) chromosome uses: ① Refactoring Operation Mutation that changes a complete refactoring operation label preserving its internal parameters (src, tgt, fld and mtd); ② Refactoring Operation Class Transposition that interchanges the src with tgt parameters inside a specific gen (the gen is selected by following a Gaussian distribution); and ③ Refactoring Operation Crossover that crosses two refactoring operations in a random position (Gaussian distribution) of the chromosomes. HaEa with variable chromosome uses: ④ Adding Refactoring Operation that adds a specific number of refactorings to the individual; ⑤ Deleting Refactoring Operation that deletes a specific number of refactorings to the individual; and ⑥ Joining Refactoring that mixes two individuals into one.

4 EVALUATION

We evaluate the performance of the approach when generating massive artificial refactoring recommendations and to present the results of *baseline search techniques* through proper statistical analysis. We performed a *preliminary experiment* and a *formal experiment* from previous reports of automated refactoring [18, 29]. The preliminary experiment evaluated the *do-ability* and the formal experiment validated the performance of the models and the time complexity given a number of classes.

We employed three open software systems, which have been regularly reported in evolutionary computation approaches for refactoring [11, 12, 18, 26]: Commons Codec v.1.10 with 123 number of classes (CCODEC), Acra v.4.6.0 with 59 classes (ACRA) and JFreeChart v.1.0.9. with 558 classes. Two datasets were initially configured for algorithms performance analysis. The first dataset contains all source classes from CCODEC and the second dataset all source classes of ACRA. In addition, we employed 6 datasets from JFreeChart (classes inside each dataset are hierarchy related). These datasets were applied for time complexity analysis.

In the preliminary results, the Shapiro-Wilk test suggested the data featured no normal distribution for both datasets ACRA and CCODEC. The $p - values$ constitute less than 0.05 so that the alternative hypothesis is rejected [8]. Hill Climbing and Simulated Annealing were compared against HaEa with a Wilcoxon test. The null hypothesis H_0 states that the median of Hill Climbing fitness is the same as HaEa's and the alternative hypothesis (H_1) constitutes that the median of HaEa fitness is greater than the baseline algorithms'. The α value follows 0.05 level of significance.

Each dataset generated a sample of fitness values that were organized sequentially. The actual position was compared respect to the previous one by keeping the least value in each evaluation (steady state). Since all the experiments were executed 30 times, we organized the results in a matrix where rows constitute the evaluation (fitness or time) and the columns the experiments. The median, the median deviation, maximum and minimum values for each row were calculated. As far as the formal experiments are concerned, *Experiment I. Algorithm's Performance* seeks to validate which approach conveys the best fitness behavior results and *Experiment II. Time Complexity of the recommendations* assesses the relationship between the number of classes and the time spent in generating artificial refactorings for such datasets.

4.1 Performance Convergence Experiment I

The following research question narrowed the study: **RQ1. What is the technique's performance when generating feasible recommendations?** To answer **RQ1**, first, the source code datasets were organized according to heritage and design relationships and, second, baseline algorithms for convergence assessment were applied to the datasets.

4.1.1 Algorithm's Performance Results. We compared three curves that represent computed fitness values for HC and SA (best, median and worst). Figure 5 depicts for three algorithms the largest evaluations attempted during our experiment. The best fitness rates in (a) and (b) are relatively homogeneous during evaluations. In (a) the median and worst values became stable after 3,000 evaluations, however, in (b) the median values are variable though all evaluations (0.97775 ± 0.00633[5000], 0.97386 ± 0.00631[7000], 0.97167 ± 0.00354[10000]). We observed that HaEa (c) obtained the best values starting from 6,000 evaluations for HC ($p - value = 0.034\,96$) and 3,000 evaluations for SA (p-value = $0.041\,17$). By 10,000 evaluations, HaEa values are significantly lower than HC ($p - value = 0.000\,16$) and SA ($p - value = 0.000\,14$). Although, HaEa behavior (c) in large evaluations is better in all sample points (Table 2) with a remarkable p-value = 0.00023. Our results exhibit that HaEa presents good performance in large evaluations. HC and SA experience relative better results in few steps than HaEA, even though the evolutionary approach reaches the best values after certain evaluations. Previous studies pose combinatorial representations of refactorings [6, 10, 20, 29], but this paper establishes a benchmark problem contrasting baseline algorithms (HC and SA) versus HaEa with an extensive performance study.

Table 2: Performance Rates over Number of Evaluations for HC, SA and HaEa in Acra System

	Performance (median fitness)		
Evaluation	Hill Climbing	Simulated Annealing	HaEa
10000	0.97589	0.97145	0.96543
20000	0.97529	0.9695	0.96133
40000	0.97077	0.9684	0.95623
50000	0.96862	0.9684	0.9557
60000	0.96815	0.96819	0.95493

p - value <= 0.000 23 vs HeEa group.

The following plots Figure 6a and Figure 6b aim to compare four algorithms. In system CCODEC, the best behavior corresponds to

[7] https://github.com/danielrcardenas/argen
[8] Table in: https://github.com/argeneration/argeneration.github.io/blob/master/assets/pdfs/RefShapiro.pdf

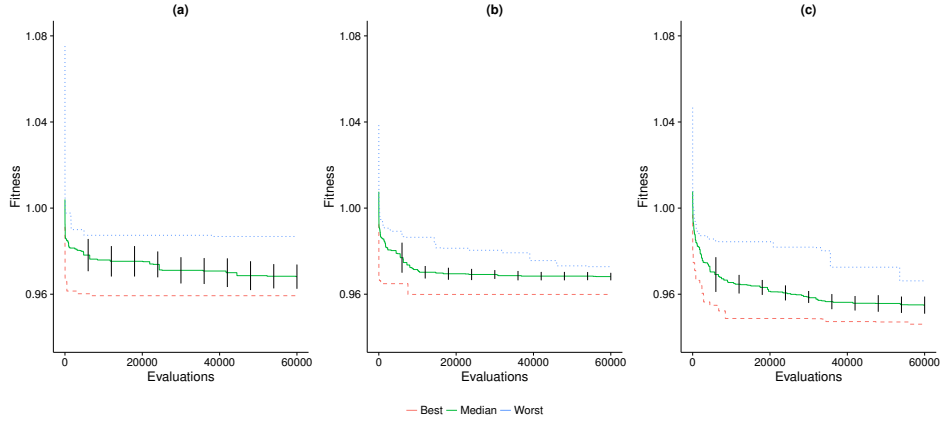


Figure 5: Acra System Fitness Performance up to 60000 Evaluations for four Combinatorial Algorithms: (a)Hill Climbing, (b)Simulated Annealing, (c)HaEa.

Simulated Annealing algorithm for 1000 evaluations, yet it presents the highest variability. While, in ACRA, HaEa experienced the best performance for 10000 and 60000 evaluations. HaEa presents good performance in large evaluations is supported by box-plot analysis. Ten thousand evaluations were executed on ACRA system, only HaEa obtains the best median value. Likewise, after 6,000 evaluations, HaEa was still searching for fewer values. Hill Climbing and Simulated Annealing induced better results in few evaluations like in CCODEC experiments with 1000 and 2000 evaluations respectively.

4.2 Time Complexity Experiment II

The JFreeChart datasets were employed to perform the time complexity evaluation. The following research question narrowed the study: **RQ2. How is the relation between the variables n classes and t time?** To answer **RQ2**, first, the source code datasets were organized according to design relationships and heritage and, second, the time analysis included both baseline and evolutionary techniques to establish the trend line between classes and time.

4.2.1 Time Complexity Results. Figure 7 depicts a comparison of the mean time complexity between baseline algorithms and HaEa search techniques for 30 independent experiments. The time complexity points forms an exponential patterns (HaEa with an equation model $y = 470.537e(0.024x)$).

The percentage increase in HC and SA techniques from 11 classes to 71 was 504.41% and 515.92% compared to 419.63% in HaEa's. The results show that each time the experiment increased the number of the classes, baseline algorithms took more time processing classes for refactoring than HaEa. HaEa seems to be a better model to estimate refactoring recommendations in less time.

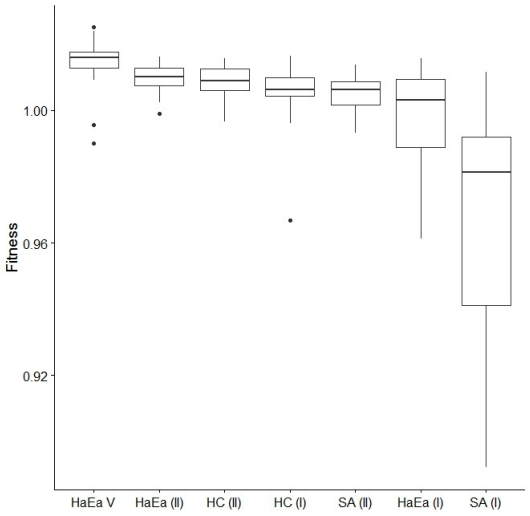
5 DISCUSSION

The following listing is a real JSON output from ACRA generated by ARGen, which recommends two types of refactorings. Such output allowed us to manually implement the refactoring [9]. Understanding software design artifacts are not enough for detecting refactoring regardless the size of the SUA. ARGen widely explores the search space to detect refactorings without any previous knowledge about the system. An automatic refactoring detection approach by employing search techniques is an open research problem. However, this research considers the single-objective case and the main goal embodies to correctly insert the assessment techniques (convergence and time complexity) into the empirical studies and not to solve the RDP.

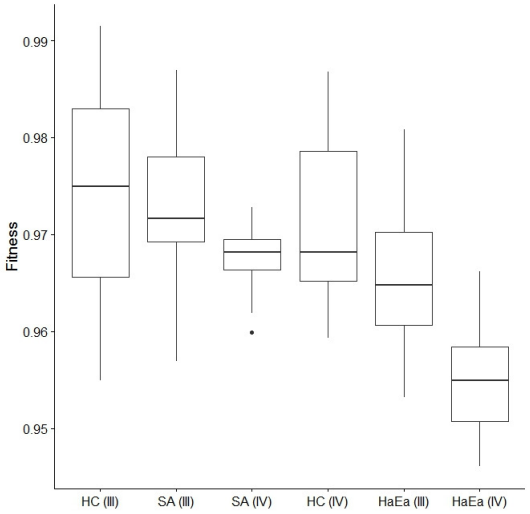
Our main finding that the combinatorial techniques are validated in terms of performance and time complexity is supported by the experiments performed in section 4. Answering **RQ1**, for dataset CCODEC, three defined curves (best, median and worst) diverge for HC and SA after 2,000 evaluations; nonetheless, for ACRA, both algorithms converged in a mean value of 0.96819 with a standard deviation of 0.00174 after 60,000 fitness evaluations. Conversely, answering **RQ2**, the relation between a number of classes and time is exponential for both HC and SA.

Seng, et al. [29] research contains the most relevant convergence study to compare with. They proposed an evolutionary algorithm for optimizing class structure in an open source system to keep class level refactoring. The outcome is a refactoring developed by the value of several quality metrics and the number of violations of object-oriented design guidelines. Seng's study design considered only one refactoring (move method) and restricted number of classes; thus, the search space was significantly reduced. Whereas, ARGen explores all the possible search space for the given SUA with 12 refactoring operations. Ignoring convergence and time complexity in software empirical analysis implies that search-based approaches do not guarantee a reasonable number of evaluations to obtain proper solutions on time. We summarize the limitations of this research as below: our research did not develop a refactoring

[9] https://github.com/argeneration/argeneration.github.io/blob/master/assets/pdfs/RefSRC.pdf and https://github.com/argeneration/argeneration.github.io/blob/master/assets/pdfs/RefTGT.pdf



(a) Commons Codec System



(b) Acra System

Figure 6: Comparative Box Plot: (I)ccodec 1000, (II)ccodec 2000, (III)acra 10000, (IV)acra 60000.

tool to implement recommendations. ARGen does not assist developers in *what software properties need to change* (e.g., reduce cyclomatic complexity by 10% and increase cohesion by 0.2 in class C), but *how to change the software* (e.g., a Replace Method with Method Object is a suggested refactoring operation for this source "src" and target "tgt" classes according to some fixed weights in the objective function). The generated sets of refactorings do not exclusively represent actionable recommendations, this depends upon developers' criteria; nonetheless, the sets do represent feasible individuals that fulfill fitness parameters and object-oriented guidelines. Our approach does not analyze the convergence-complexity limitations of multi-objective optimization approaches, however, interpreting the results of the reproducibility section, we intuit that RDP based

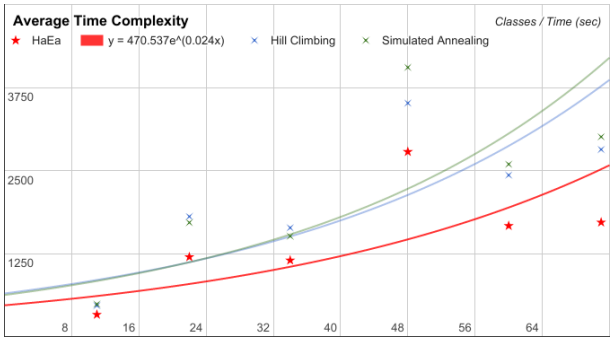


Figure 7: Average Time Complexity Hill Climbing vs Haea. In blue and green, the hill climbing and simulated annealing trendline. In red, HaEa trendline with the exponential model. Y-axis is the time in seconds and X-axis is the number of classes for JFreeChart System.

on quality metrics can be handled without multi-objective optimization. Finally, the empirical evaluations concentrated on algorithm performance and refactoring structure coherence.

Regardless of reported empirical studies [15, 16, 18, 21, 23, 24, 26], which exhibits precision and recall measures, the refactoring process highly depends on *human factor* that comprises specific domain knowledge and expertise when designing reconstructions on the code. Furthermore, the proposed set theory refactoring formalization is a first step to envision the refactoring recommendation as a graph instead of a sequence [30]. A graph representation of refactorings would be powerful and flexible.

We, indeed, accomplished to generate massive artificial refactorings, yet we cannot guarantee that none of those refactorings were actionable. Actionability implies not only developers' interest in quality metric but also developers' criteria (how to design a SUA) that should be extracted from their minds. Consequently, search techniques or combinatorial analysis are insufficient approximations to tackle the RDP because those techniques cannot acknowledge how the developer's mind is simulated, employed, or mapped to make the exact context (human design process) for recommending refactorings. Moreover, notice that assessing convergence requires a large number of evaluations and the time complexity of applying search techniques in just detecting a refactoring -with as possible fewer constraints- is exponential.

```
{ "MM": [
    {
      "tgt": "[TypeDeclaration [name=
            NewCrashReporterPersister]]",
      "src": "[TypeDeclaration [name=CrashReporterPersister]
            ]",
      "mtd": "[Store|EXISTS]"
    }
  ],
  "MF": [
      {
      "tgt": "[TypeDeclaration [name=
            NewCrashReporterPersister]]",
      "src": "[TypeDeclaration [name=CrashReporterPersister]
            ]",
      "fld": "[Context|EXISTS]"
    }
  ]
}
```

6 CONCLUSION AND FUTURE WORK

We introduce a systematic formal approach to generate artificial refactoring recommendations based on quality metrics, which is used for assessing convergence and time complexity. We contributed to reducing the gap between the software empirical analysis (in the context of refactoring detection process) and search based techniques by establishing the fundamentals of performance convergence and time complexity in a unified mathematical theory toward a combinatorial optimization model. Our paper propounded a definition, an implementation, and an evaluation of a systematic approach that empirically analyses the convergence of a set of artificial refactoring operations. We plan to research on how our approach automatically adjusts the objective function weights to reduce the error-proneness in order to avoid multi-objective optimization since the problem does not exhibit strongly opposite objectives.

ACKNOWLEDGMENTS

The authors are grateful to Carlos Sierra, Oscar Chaparro, and Miguel Ramirez for their valuable comments and helpful suggestions during the model implementation and statistical analysis. The authors would also like to thank the former members of ALIFE and SEMERU research groups and the anonymous reviewers for their valuable feedback.

REFERENCES

[1] Gabriele Bavota, Andrea De Lucia, Massimiliano Di Penta, Rocco Oliveto, and Fabio Palomba. 2015. An experimental investigation on the innate relationship between quality and refactoring. *Journal of Systems and Software* (2015). https://doi.org/10.1016/j.jss.2015.05.024

[2] Oscar Chaparro, Gabriele Bavota, Andrian Marcus, and Massimiliano Di Penta. 2014. On the Impact of Refactoring Operations on Code Quality Metrics. In *2014 IEEE International Conference on Software Maintenance and Evolution*. https://doi.org/10.1109/ICSME.2014.73

[3] Xianshun Chen, Yew Soon Ong, Meng Hiot Lim, and Kay Chen Tan. 2011. A multi-facet survey on memetic computation. *IEEE Transactions on Evolutionary Computation* 15, 5 (2011), 591–607. https://doi.org/10.1109/TEVC.2011.2132725

[4] Martin Fowler and Kent Beck. 1999. *Refactoring: improving the design of existing code*. Addison-Wesley Professional.

[5] Jonatan Gomez. 2017. Stochastic Global Optimization Algorithms: A Systematic Formal Approach. (2017).

[6] M Harman and L Tratt. 2007. Pareto Optimal Search Based Refactoring at the Design Level. In *Proceeding of the Genetic and Evolutionary Computation Conference GECCO 2007*. https://doi.org/10.1145/1276958.1277176

[7] Adam C Jensen and Betty H C Cheng. 2010. On the use of genetic programming for automated refactoring and the introduction of design patterns. *Proceedings of the 12th Annual Conference on Genetic and Evolutionary Computation (GECCO'10)* (2010), 1341–1348. https://doi.org/10.1145/1830483.1830731

[8] Jonatan Gomez. 2004. Self adaptation in evolutionary algorithms. In *GECCO 2004, LNCS*. Universidad Nacional de Colombia, 12. https://doi.org/10.1109/CEC.2004.1331103

[9] Mark O Keeffe and Mel Ó Cinnéide. 2007. Search-Based Refactoring : an empirical study. *Journal of Software Maintenance and Evolution: Research and Practice* August 2007 (2007), 1–7.

[10] Marouane Kessentini, Wael Kessentini, Houari Sahraoui, Mounir Boukadoum, and Ali Ouni. 2011. Design defects detection and correction by example. In *IEEE International Conference on Program Comprehension*. https://doi.org/10.1109/ICPC.2011.22

[11] Wael Kessentini, Marouane Kessentini, Houari Sahraoui, Slim Bechikh, and Ali Ouni. 2014. A Cooperative Parallel Search-Based Software Engineering Approach for Code-Smells Detection. *IEEE Transactions on Software Engineering* (2014). https://doi.org/10.1109/TSE.2014.2331057

[12] Thainá Mariani and Silvia Regina Vergilio. 2017. A systematic review on search-based refactoring. *Information and Software Technology* 83 (2017), 14–34. https://doi.org/10.1016/j.infsof.2016.11.009

[13] Tom Mens, Serge Demeyer, Bart Du Bois, Hans Stenten, and Pieter Van Gorp. 2003. Refactoring: Current research and future trends. *Electronic Notes in Theoretical Computer Science* 82, 3 (2003), 483–499. https://doi.org/10.1016/S1571-0661(05)82624-6

[14] Tom Mens and Tom Tourwé. 2004. A survey of software refactoring. *IEEE Transactions on Software Engineering* 30, 2 (2004), 126–139. https://doi.org/10.1109/TSE.2004.1265817

[15] Mohamed Wiem Mkaouer, Marouane Kessentini, Slim Bechikh, and Mel Ó Cinnéide. 2014. A robust multi-objective approach for software refactoring under uncertainty. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. https://doi.org/10.1007/978-3-319-09940-8-12

[16] Mohamed Wiem Mkaouer, Marouane Kessentini, Slim Bechikh, Kalyanmoy Deb, and Mel Ó Cinnéide. 2014. Recommendation system for software refactoring using innovization and interactive dynamic optimization. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering - ASE '14*. https://doi.org/10.1145/2642937.2642965

[17] Mohamed Wiem Mkaouer, Marouane Kessentini, Slim Bechikh, Mel Ó Cinnéide, and Kalyanmoy Deb. 2015. On the use of many quality attributes for software refactoring: a many-objective search-based software engineering approach. (2015), 43 pages. https://doi.org/10.1007/s10664-015-9414-4

[18] Mohamed Wiem Mkaouer, Marouane Kessentini, Mel Cinnéide, Shinpei Hayashi, and Kalyanmoy Deb. 2016. A robust multi-objective approach to balance severity and importance of refactoring opportunities. *Empirical Software Engineering* (2016), 1–34. https://doi.org/10.1007/s10664-016-9426-8

[19] Martin Odersky and Tiark Rompf. 2014. Scala unifies traditionally disparate programming-language philosophies to develop new components and component systems. *COMMUNICATIONS OF THE ACM* 57 (2014), 76–86. https://doi.org/10.1145/2591013

[20] Mark O'Keeffe and Mel Ó Cinnéide. 2008. Search-based refactoring for software maintenance. *Journal of Systems and Software* 81, November 2006 (2008), 502–516. https://doi.org/10.1016/j.jss.2007.06.003

[21] Ali Ouni, Marouane Kessentini, Slim Bechikh, and Houari Sahraoui. 2015. Prioritizing code-smells correction tasks using chemical reaction optimization. *Software Quality Journal* (2015). https://doi.org/10.1007/s11219-014-9233-7

[22] Ali Ouni, Marouane Kessentini, and Houari Sahraoui. 2013. Search-based refactoring using recorded code changes. *Proceedings of the European Conference on Software Maintenance and Reengineering, CSMR* (2013), 221–230. https://doi.org/10.1109/CSMR.2013.31

[23] Ali Ouni, Marouane Kessentini, Houari Sahraoui, and Mounir Boukadoum. 2013. Maintainability defects detection and correction: A multi-objective approach. *Automated Software Engineering* 20 (2013), 47–79. https://doi.org/10.1007/s10515-011-0098-8

[24] Ali Ouni, Marouane Kessentini, Houari Sahraoui, and Mohamed Salah Hamdi. 2012. Search-based refactoring: Towards semantics preservation. *IEEE International Conference on Software Maintenance, ICSM* (2012), 347–356. https://doi.org/10.1109/ICSM.2012.6405292

[25] Ali Ouni, Marouane Kessentini, Houari Sahraoui, and Mohamed Salah Hamdi. 2013. The use of development history in software refactoring using a multi-objective evolutionary algorithm. *Proceeding of the fifteenth annual conference on Genetic and evolutionary computation conference - GECCO '13* (2013), 1461. https://doi.org/10.1145/2463372.2463554

[26] Ali Ouni, Marouane Kessentini, Houari Sahraoui, Katsuro Inoue, and Mohamed Salah Hamdi. 2015. Improving multi-objective code-smells correction using development history. *Journal of Systems and Software* 105 (2015), 18–39. https://doi.org/10.1016/j.jss.2015.03.040

[27] Eduardo Gurgel Pinho and Francisco Heron De Carvalho. 2014. An object-oriented parallel programming language for distributed-memory parallel computing platforms. *Science of Computer Programming* 80, PART A (2014), 65–90. https://doi.org/10.1016/j.scico.2013.03.014

[28] Edmund Burke; Michel Gendreau; Matthew Hyde; Graham Kendall; Gabriela Ochoa; Ender Ozcan; Rong Qu. 2012. Hyper-heuristics: A Survey of the State of the Art. *Journal of the Operational Research Society* (2012), 1695–1724. https://doi.org/10.1057/jors.2013.71

[29] Olaf Seng, Johannes Stammel, and David Burkhart. 2006. Search-Based Determination of Refactorings for Improving the Class Structure of Object-Oriented Systems Categories and Subject Descriptors. *GECCO 2006 - Genetic and Evolutionary Computation Conference* (2006), 1909–1916. https://doi.org/10.1145/1143997.1144315

[30] A.V. Zarras, T. Vartziotis, and P. Vassiliadis. 2015. Navigating through the archipelago of refactorings. In *2015 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE 2015 - Proceedings*. https://doi.org/10.1145/2786805.2803203