Research Square

Preprints are preliminary reports that have not undergone peer review.
They should not be considered conclusive, used to inform clinical practice,
or referenced by the media as validated information.

Multi-objective Integer Linear Programming approach for Automatic Software Cognitive Complexity Reduction

Adriana Novoa-Hurtado

novoa@uma.es

University of Malaga

Rubén Saborido

University of Malaga

Francisco Chicano

University of Malaga

Manuel Giménez-Medina

Ayesa Advanced Digital Services Limited (ASDA)

Research Article

Keywords: cognitive complexity, code refactoring, software maintenance, multi-objective optimization

Posted Date: April 13th, 2026

DOI: https://doi.org/10.21203/rs.3.rs-9281346/v1

License: This work is licensed under a Creative Commons Attribution 4.0 International License.
Read Full License

Additional Declarations: No competing interests reported.

Multi-objective Integer Linear Programming approach for Automatic Software Cognitive Complexity Reduction

Adriana Novoa-Hurtado[1*], Rubén Saborido[1†], Francisco Chicano[1†], Manuel Giménez-Medina[2†]

[1]ITIS Software, Universidad de Málaga.
[2]Ayesa Advanced Digital Services Limited (ASDA).

*Corresponding author(s). E-mail(s): novoa@uma.es;
Contributing authors: rsain@uma.es; chicano@uma.es;
mgimenezm@ayesa.com;
†These authors contributed equally to this work.

Abstract

Clear and concise code is necessary to ensure maintainability, so it is crucial that the software is as simple as possible to understand, to avoid bugs and, above all, vulnerabilities. One metric of software quality is cognitive complexity, which measures the relative effort a developer must expend to understand a piece of code. In this work, we will use the cognitive complexity measure defined by SonarSource —the company behind products such as SonarQube— as a reference to assess this difficulty in understanding code at the method level. This measure considers control flow statements and their nesting level. There are many ways to enhance software without changing its functionality, with refactoring as the primary approach. One refactoring that reduces cognitive complexity is the extract method refactoring. The decision of which extraction operations to apply has been modeled as a combinatorial optimization problem in recent years, aiming to minimize the number of operations. This goal does not always align with the developer's preferences. Motivated by a survey conducted with our industrial partner, this work models the problem of reducing cognitive complexity as a multi-objective integer linear program. In addition to minimizing the number of extractions, our new formulation seeks to balance cognitive complexity and the number of lines of code between the original and extracted methods. We implemented two algorithms to solve the multi-objective problem and evaluated them

on a corporate software project and 10 open-source projects. Experimental evaluation shows that one of the two algorithms finds optimal solutions for 70.47% of the analyzed issues, with a median execution time of 24.5 seconds, and the other finds optimal solutions for 73.77% of the analyzed issues, with a median execution time of 10.8 seconds. This new approach attains substantial improvements in refactoring quality, in terms of cognitive complexity balance and number of lines of code balance, in exchange for a very acceptable increase in the average number of extractions. We achieved a total savings of 275.7 hours of development effort. This translates to approximately 34 full workdays that the engineering team can devote to higher-value tasks.

Keywords: cognitive complexity, code refactoring, software maintenance, multi-objective optimization

1 Introduction

Software systems become more complex as they grow, making it difficult for developers to fully understand all their details. That is why it is essential to develop clear code, not only to understand it, but also because other developers will likely use or modify it. Reducing code complexity has been a constant concern in software engineering, from classic structural metrics such as McCabe's Cyclomatic Complexity [1] to more recent proposals explicitly aimed at modeling human cognitive load.

Recently, a new cognitive complexity metric has been proposed and integrated into the well-known static code analysis tools SonarCloud[1] and SonarQube[2], which are, respectively, a service and an open-source platform for continuous code quality inspection. This cognitive complexity metric, which we will refer to from now on as **CC** (for Cognitive Complexity), reflects the relative difficulty of understanding and, therefore, maintaining methods, classes, and applications. The CC of a method within a class is defined as a positive integer that increases each time a control statement is encountered. The nesting level of these control statements within the method also contributes to the CC of the code. Although SonarSource suggests keeping the code's CC below a certain threshold, software developers have no support for optimally reducing it.

Refactoring is a transformation that preserves the original functionality of the code. One way to reduce the CC associated with a method is to extract certain lines of code into new methods. This strategy is known as the *Extract Method* refactoring. Reducing the CC of the code involves searching for sequences of method extractions to reduce it below the established limit [2].

Saborido et al. [3] modeled the reduction of the CC to a given threshold as an Integer Linear Programming (ILP) problem, minimizing the number of code extractions to reduce the CC of Java methods to or below a given threshold, enabling exact solution by solvers. Furthermore, this approach is beneficial because it does not introduce too many methods that distort the code written before refactoring. However,

[1] https://sonarcloud.io (last accessed March 2026)
[2] https://www.sonarqube.org (last accessed March 2026)

a survey[3] conducted with a software factory called Ayesa Servicios Digitales Avanzados SL (ASDA)[4] revealed that just minimizing the number of extractions could result in highly unbalanced methods, both in terms of CC and lines of code (hereafter referred to as **LOC**). Consequently, it seems natural to add multiple objectives to the problem model. Therefore, in this work, the problem of optimizing CC reduction in a method is modeled as a multi-objective integer linear programming (MO-ILP) problem. The experimental evaluation is performed on existing methods from 10 open-source projects and an industrial project.

The contributions of this work are:

1. Formulating a multi-objective ILP variant of the software CC reduction problem, where the three ILP objectives available with which we work are to minimize the number of extractions performed to a given method and equilibrate the CC and LOC of all resultant methods (both the refactored method and its extractions).
2. Designing a tool, called "ILP CC reducer tool", that integrates several multi-objective ILP problem-solving algorithms that obtain a set of solutions aimed at reducing the CC of a given method. This tool can combine one, two, or three of the available objectives to model and solve the chosen MO-ILP problem.
3. Carrying out experimental validation of the proposed multi-objective ILP model and the "ILP CC reducer tool" using a benchmark of methods from open-source and private industrial software projects in Java. CC reduction will be achieved by working at the method level and ensuring that the CC limit is not exceeded in each method. SonarSource sets a maximum CC of 15 for Java methods[5]. Above this value, the source code for that method is considered to have maintainability issues.

This new multi-objective approach gives developers greater operational leeway to continue editing the code after refactoring. This contribution is particularly relevant to collaborative, long-term projects, where code maintainability and readability are critical. If none of the resulting methods have high CC and none are significantly longer than others, then the refactored code becomes more uniform and manageable. This approach facilitates future modifications and extensions without compromising quality or exceeding the recommended CC limits again. In terms of human effort, this new approach achieved a total savings of 275.7 hours of development effort. This translates to approximately 34 full workdays that the engineering team can devote to higher-value tasks.

[3]The survey is included in a previous work which is currently under review at another journal.

[4]Ayesa Servicios Digitales Avanzados SL (ASDA) is a global provider of technological and engineering services, with 13,000 professionals and a direct presence in 24 countries across Europe, the Americas, Africa, Asia, and Oceania. They develop and implement digital solutions for companies and public administrations, applying the latest technologies to design and supervise infrastructures. They have teams specialized in more than 70 disciplines and certified in market-leading technologies, working across digital administration, health, industry, consumer, banking, insurance, telecommunications, media, energy and utilities, transportation, building and urban planning, resources, and the environment. In its vocation to be a global, creative, technology-driven, and human-centric company, Ayesa is committed to talent, through diversity and inclusion, as well as sustainability, as hallmarks and levers for innovation.

[5]S3776 - Reason for the current default value of 15: https://community.sonarsource.com/t/s3776-reason-for-the-current-default-value-of-15/127103 (last accessed March 2026)

The remainder of this paper is organized as follows. Section 2 motivates the challenge of reducing code's CC. Section 3 sets the background and introduces the concepts and notation used throughout the rest of the document. Section 4 presents the formulation of the proposed multi-objective ILP model. This section introduces two algorithms for solving the multi-objective model. Section 5 shows the implementation of the "ILP CC reducer tool" at the method level. It describes the tool's ILP module, its architecture, and its implementation. Sections 6 and 7 present the experimental analysis and the results obtained for the MO-ILP model using the different algorithms developed, respectively. Section 8 revises the related work about refactoring and CC, as well as the contributions of this paper. Section 9 presents the conclusions derived from the experimental results and proposes future work, including potential extensions and improvements.

1.1 Replication package

The complete tool resulting from this paper can be found on GitHub[6]. Similarly, the full replication package can be found on Zenodo[7]. This full replication package includes (i) all the source code developed, (ii) an interactive full p-split example's graph, (iii) results, and (iv) the results' statistics. All the studies presented in this paper can be replicated using the materials provided in this replication package.

2 Motivation

To ensure clarity when illustrating the difficulties developers face when reducing the CC of code, we use the `routeAPacketTo` method in the `Host` class of the `cybercaptor-server`[8] open-source Java project. It is shown in Figure 1, and it has CC = 20, which is above the maximum recommended by SonarQube (CC = 15).

The number of different extract method refactoring opportunities of a method with n sentences is bounded by $CR_2^n = \binom{n+2-1}{2} = \binom{n+1}{2}$, given that each extraction is delimited by its first and last sentences in the code. The running example has 28 lines of code, so an upper bound for the number of feasible code extractions is $\binom{29}{2} = 406$. Thus, analyzing all possible refactoring opportunities to identify those that reduce a method's CC to a given threshold becomes unmanageable for developers [2]. Furthermore, not all extractions are applicable, as they could break a control flow sentence (for instance, Lines 11 and 12 in Figure 1 cannot be extracted as a new method) or they do not contribute to the CC of the method (see Lines 14 and 16 in Figure 1). For the running example, we computationally checked that there are 23 applicable code extractions that reduce the initial CC of the method. This means that the upper bound on all possible sequences of extract method refactoring operations is $2^{23} - 1 = 8,388,607$.

As explained in the introduction, we previously modeled CC reduction as a single-objective integer linear programming problem that minimizes the number of extractions. After surveying 38 professional developers, they found that this type of

[6] https://github.com/anhe2p9/M2I-TFM-Adriana.git
[7] https://doi.org/10.5281/zenodo.15682834
[8] https://github.com/fiware-cybercaptor/cybercaptor-server.git (last accessed March 2026)

```
1  public void routeAPacketTo(IPAddress ip, int ttl, List<Host> usedHosts) throws Exception {
2      usedHosts.add(this);
3      if (ttl == 0) {//Problem in routing
4          throw new Exception("Routing problem...");
5      }
6      if (!hasIP(ip)) { //Packet not arrived
7          Host nextHost = null;
8          List<Host> directlyAccessibleHosts = getDirectlyAccessibleHosts();
9          for (Host directlyAccessibleHost: directlyAccessibleHosts) {
10             if (...) // If the packet is for a neighbour, we send it to him
11                 nextHost = directlyAccessibleHost;
12         }
13         if (nextHost != null) {
14             nextHost.routeAPacketTo(ip, ttl - 1, usedHosts);
15         } else {//We have to look in the routing table
16             List<Host> directlyAccessible = getDirectlyAccessibleHosts();
17             IPAddress nextIP = this.getRoutingTable().getNextHop(ip);
18             boolean nextHostFound = false;
19             for (Host aDirectlyAccessible: directlyAccessible) {
20                 if (...) { //Search the nextHop host object
21                     aDirectlyAccessible.routeAPacketTo(ip, ttl - 1, usedHosts);
22                     nextHostFound = true;
23                 }
24             }
25             if (!nextHostFound) { //Routing problem
26                 throw new Exception("Routing problem...");
27             }
28         }
29     }
30 }
```

Fig. 1: Method `routeAPacketTo` in the `Host` class of the open-source project `cybercaptor-server`, used as running example in the project.

solution was not adequate for improving code comprehension in some specific cases; in those cases, developers preferred code that was balanced in terms of CC and LOC. It seemed natural, then, to incorporate these objectives into the problem while keeping the number of extractions in mind. Solving this problem yields solutions that minimize code extractions while balancing the CC and LOC of extracted methods, thereby enabling clean, readable, and maintainable code.

3 Background

The concepts used throughout the rest of the document are outlined in this section. Cognitive Complexity, linear programming, and multi-objective integer linear programming are introduced. The notation used for the rest of the document is defined in the section.

3.1 Cognitive complexity (CC) of a code

Cognitive Complexity is a relatively novel measure of how difficult a piece of code is to understand and handle intuitively. Unlike the more traditional Cyclomatic Complexity, which focuses on the number of possible execution paths in a program, CC shifts the attention toward human factors. This new concept focuses on understandability, i.e., how long it takes a developer to understand the code. Different metrics have been defined over the years to reduce it as much as possible. By minimizing CC, teams can improve collaboration, reduce onboarding time for new developers, and decrease the likelihood of introducing errors during future modifications.

The software development community has long sought to predict the effort required for software development or quality assessment, leading to the proposal of various

metrics. In 1983, Basili et al. conducted an empirical study to investigate the correlation between different measures and actual data [4]. Later in 1988, Weyuker presented abstract properties to formally compare software complexity models [5]. These properties allow developers to evaluate a large number of existing software metrics. Most metrics focus on method complexity, and then the complexities of each method are added. Weyuker states that the complexity of a program, as measured by data flow, depends directly on the placement of statements and on how components interact via the potential flow of data. Kaner and Bond conducted an evaluation to validate proposed metrics until 2004, during which they criticized the way metrics programs are used. They attempted to evaluate measurements by answering 10 questions, such as their purpose, scope, and scale, among others [6]. Misra et al. proposed a suite of cognitive metrics for evaluating complexity for object-oriented code [7]. The progression observed over the years is that researchers have sought to define a formal metric to measure something subjective. Finally, it was possible to validate a measure for determining the effort a developer spends understanding code [8]. This metric is available for most programming languages[9].

Once the CC metric is defined, one would like to reduce it to a given threshold, resulting in a clean, easy-to-understand, and easy-to-manage code. CC reduction can be explained both qualitatively and quantitatively. Qualitatively, it is easier to understand a compact code by substituting a piece of code with a line that explains its functionality, i.e., the extracted method name. Recio-Abad et al. found that the names offered by LLMs are highly acceptable [9]. On the other hand, CC reduction can be explained quantitatively. If one subtracts the metrics for a target method, not only is the CC of that method reduced, but the sum of the CCs of the refactored method and its extractions is usually lower than the initial CC.

3.2 Linear programming

Linear Programming aims to optimize a linear function subject to a set of inequalities [10]. Given a set of real numbers $a_1, a_2, ..., a_n$ and a set of variables $x_1, x_2, ..., x_n$, we define a linear function f on those variables by

$$f(x_1, x_2, ..., x_n) = a_1x_1 + a_2x_2 + ... + a_nx_n = \sum_{j=1}^{n} a_jx_j. \quad (1)$$

If b is a real number and f is a linear function, the equation $f(x_1, x_2, ..., x_n) = b$ is a linear equality, while $f(x_1, x_2, ..., x_n) \leq b$ and $f(x_1, x_2, ..., x_n) \geq b$ are linear inequalities. We use the general term linear constraints to denote either linear equalities or inequalities. In linear programming, we do not allow strict inequalities, but they can usually be transformed into non-strict inequalities. A linear programming problem consists of minimizing or maximizing a linear function subject to a finite set of linear constraints. If we are minimizing, we refer to the linear program as a minimization linear program. If we are to maximize, then we call the linear program a maximization linear program.

[9] Webinar refactoring with cognitive complexity: https://community.sonarsource.com/t/webinar-refactoring-with-cognitive-complexity/45331/2

An integer linear programming problem (ILP) is a linear programming problem with the additional constraint that the variables x must take integral values. Determining whether an integer linear program has a feasible solution is NP-hard, meaning there is no known polynomial-time algorithm to solve it. A mixed-integer linear programming (MILP) problem is a linear programming problem that incorporates both integer and continuous variables.

3.3 Multi-objective optimization

It is common to encounter real-world problems where we need to optimize multiple objectives simultaneously. In these cases, the objectives often conflict, as optimizing one typically results in the rest deteriorating. The goal then is to find a set of solutions that satisfy every constraint, where no objective can be improved without worsening at least one of the others. A multi-objective optimization problem can be formulated as follows [11]:

$$\text{minimize} \quad \mathbf{f}(\mathbf{x}) = (f_1(\mathbf{x}), f_2(\mathbf{x}), ..., f_k(\mathbf{x}))^T \tag{2}$$

$$\text{s.t} \quad \mathbf{x} \in S, \tag{3}$$

where:

- $\mathbf{x} = (x_1, x_2, ..., x_n)^T \in S$ is the vector of decision variables.
- $S \subset \mathbb{R}^n$ is the decision space of feasible solutions.
- $f_i : S \to \mathbb{R}, \quad i = 1, ..., k$, with $k \geq 2$ are the objective functions to be optimized.
- The image of S in $\mathbb{R}^k$, i.e. $Z = \mathbf{f}(S) \subset \mathbb{R}^k$, is the objective space.
  - The objective space is composed of the objective vectors as $\mathbf{f}(\mathbf{x})$, for any $\mathbf{x} \in S$.

Since the objectives are often conflicting, finding a single decision vector (solution) that optimizes all the objectives simultaneously is usually impossible. Thus, much attention is given to the so-called Pareto-optimal solutions, for which none of the objectives can be improved without worsening at least one of the others. The set of Pareto optimal solutions in the decision space is called the Pareto optimal set, and its image in the objective space is the Pareto front (PF) [11]. Each element of the Pareto front is a point, but it is usually referred to as a "solution" due to a common abuse of language.

Now we define some important multi-objective concepts using Figure 2, which is an example of a PF in the objective space. For $\mathbf{z}, \overline{\mathbf{z}} \in \mathbb{R}^k$:

- **Domination relation**: $\mathbf{z}$ *dominates* $\overline{\mathbf{z}}$ if and only if $z_i \leq \overline{z}_i$ for all $i = 1, ..., k$, and $z_j < \overline{z}_j$ for, at least, one index j. The left-most blue solution in Figure 2 dominates the brown solution (the brown solution is a *dominated* solution).
- **Efficient solution** (see the blue solutions in Figure 2): one solution is efficient if there is no other solution in the search space that dominates it.

- **Unsupported efficient solution** (see the green solution in Figure 2): $\mathbf{z}$ is an unsupported efficient solution if it cannot be obtained by minimizing a linear combination of the objectives. Therefore, unsupported solutions, although non-dominated, are not contained in the convex envelope of the PF.
- **Weak domination relation**: $\mathbf{z}$ *weakly dominates* $\overline{\mathbf{z}}$ if $z_i \leq \overline{z}_i$ for all $i = 1, ..., k$.
- **Strict domination relation**: $\mathbf{z}$ *strictly dominates* $\overline{\mathbf{z}}$ if $z_i < \overline{z}_i$ for all $i = 1, ..., k$.
- **Weakly efficient solution**: a solution z is *weakly efficient* (red and blue solutions in Figure 2) if it is not strictly dominated by any other solution in the search space. All the efficient solutions are also weakly efficient.

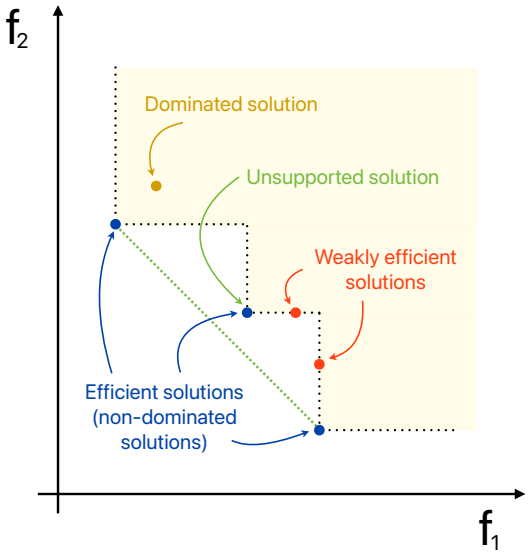


Fig. 2: Objective space example and types of solutions that may appear in the objective space. Dominated solutions do not belong to the PF.

A non-dominated solution set is a set of solutions whose objective vectors are not dominated by any other objective vector of a solution in the set. For two non-dominated solution sets A and B (in the objective space $\mathbb{R}^k$), we say that A *strictly dominates* B (or that A is *better* than B) if, for every $\mathbf{z} \in B$, there is at least one $\mathbf{y} \in A$ such that $\mathbf{y}$ weakly dominates $\mathbf{z}$ (and this means that A *weakly dominates* B), but there exists at least one solution in A that is not weakly dominated by any solution in B (i.e. B does not weakly dominate A).

3.4 Notation

CC can be computed as the sum of two components, the *inherent component* and the *nesting component*. The inherent component adds +1 for each occurrence of certain control flow structures and complex expressions, such as logical operators in conditionals and loops. The nesting component depends on the depth at which a specific control flow structure appears in the code related to the method declaration. Each possible extraction will be characterized by the range of positions in the file, that is, the start and end characters (offsets) that define where the code extraction begins and ends.

Let s_i, s_j and e_i, e_j be the start and end offsets of the ith and jth code extractions, respectively. We consider the jth extraction is nested in the ith extraction, denoted with $j \to i$, when $[s_j, e_j] \subset [s_i, e_i]$. In that case, we say that the jth extraction is the *descendant* of the ith extraction, and that the ith extraction is the *ancestor* of the jth extraction. Code that is nested inside another, which in turn is nested inside a third, is ultimately also nested within the third. Regarding the running example in Figure 1, the first `for` loop in Lines 8–10 is nested in the second conditional in Lines 5–22. As we are working at method level, the code associated with the body of the original method is the 0th extraction. Thus, $i \to 0$ can be written for all $i \geq 1$, which means that all extractions are nested in the body of the original method.

We consider that the ith extraction is in conflict with the jth extraction, denoted with $i \not\leftrightarrow j$ when i and j are not nested one into the other, and $[s_j, e_j] \cap [s_i, e_i] \neq \varnothing$. Two extractions cannot be extracted simultaneously if they are in conflict. In the running example, the extraction between Lines 16–24 conflicts with the extraction between Lines 17–27, and therefore, these two extractions cannot be extracted simultaneously.

Next, we define new variables to facilitate formal CC reduction. To better understand the use of each variable, Figure 3 shows them as comments in the running example presented in Figure 1. We only consider the variables of those feasible extractions that contribute to the CC of the method. With this notation, the 0th extraction is the complete method that we want to refactor. Those variables are defined as follows.

The nesting component of the ith feasible extraction is denoted by λ_i. Note that $\lambda_i = 0$ when no nesting exists in extraction i. $\lambda_0 = 0$ is the depth of the complete method. Referring to the running example, the conditional sentence in Line 5 of Figure 3 has $\lambda = 0$ because its nesting level is zero, but the conditional sentence in Line 16 has $\lambda = 2$ because it is nested in the `for` loop in Line 14, and that `for` loop is in turn nested in the conditional sentence in Line 10.

The accumulated inherent component of the ith extraction and the ones contained by it are denoted by ι_i. Referring to the running example, the conditional sentence in Line 37 found in Figure A2 (in the appendix) has $\iota = 1$ because the unique contribution to the inherent component of the sentence is provided by the sentence itself. But the conditional sentence in Line 21 has $\iota = 5$ because inside that sentence there are the `else` sentence of Line 24 (with $\iota = 4$), the `for` loop in Line 29 (with $\iota = 2$), the conditional sentence in Line 31 (with $\iota = 1$), and the conditional sentence in Line 37 (with $\iota = 1$).

The accumulated nesting component of the extractions contained by the ith extraction (considering nesting 0 for the ith extraction) is denoted as ν_i. The total

accumulated nesting component of the complete method, ν_0, is obtained by adding every contribution ν_i to the zero-th extraction (that is, the complete method). Referring to the running example, the `for` loop in Line 29 found in Figure A3 (in the appendix) contributes to the accumulated nesting component of the method with +2 because its nesting level is $\lambda = 2$. On the other hand, the conditional sentence in Line 31 contributes to the accumulated nesting component of the method in the running example with +3 because its nesting level is $\lambda = 3$. Both sentences contribute to the accumulated nesting component of the method with $\nu_i = +2 + 3 = +5$.

The number of extractions that contribute to the accumulated nesting component of the ith feasible extraction, including itself when its nesting component is not 0, is denoted by μ_i. Referring to the running example, the `for` loop in Line 29 found in Figure A4 (in the appendix) has $\mu = 2$ because both the conditional sentence inside it (Line 31) and the `for` loop itself contribute to the nesting level. Hence, they are two sentences that contribute, so $\mu = 2$. In the case of the `else` sentence in Line 24 in Figure A4 (in the appendix), $\mu = 3$ because it has three sentences that contribute to the nesting level. They are the `for` loop in Line 29, the conditional sentence in Line 31, and the conditional sentence in Line 37. Note that the `else` sentence does not contribute to the nesting level [8], even though this sentence nesting level is different from 0. That is why there is no contribution to μ in this `else` sentence, as it can be observed in Figure A4. In the case of the conditional sentence in Line 5, its μ value is 0 because it does not contribute to the nesting level, and it does not have any feasible extractions that contribute to the nesting level inside it.

Once the parameters λ_i, ι_i, ν_i, and μ_i are defined, they are combined to define $CCR_{j\to i} = \iota_j + \nu_j + (\lambda_j - \lambda_i)\mu_j$ and $NMCC_i = \iota_i + \nu_i$. In plain English, $CCR_{j\to i}$ is the cognitive complexity reduction of sequence i when extracting sequence j (we should have $j \to i$). On the other hand, $NMCC_i$ is the cognitive complexity of the new method when extracting sequence i. Note that if one extraction is not nested, then $NMCC_j = CCR_{j\to i}$, since $\lambda_j = \lambda_i = 0$ in that case. That is why one would like to extract a sequence of lines of code that is deeply nested but also has a high $NMCC$, because in that case, both the original method and the extracted method would experience a large CC reduction.

4 Multi-objective ILP model and optimization algorithms

The formulation of the multi-objective ILP model is inspired by the model developed by Saborido et al. [3]. They use a zero-one program, that is, an integer program in which the variables can only take values 0 or 1. This model minimizes the number of extractions, ensuring that all extractions made to the code, as well as to the original method, have CC below or equal to a given limit, τ. New objectives and constraints have been added to this ILP problem to balance the CC and LOC of the extracted methods. The new model is formalized as follows:

```
1  // Cognitive complexity 20
2  public void routeAPacketTo(IPAddress ip, int ttl, List<Host> usedHosts) throws Exception {
3      usedHosts.add(this);
4      // [λ=0, ι=1, ν=0, μ=0, CCR=1, NMCC=1]
5      if (ttl == 0) {//Problem in routing
6          throw new Exception("Routing problem, TTL is null : packet to "
7                              + ip + " deleted on host " + this.getName());
8      }
9      // [λ=0, ι=8, ν=11, μ=6, CCR=19, NMCC=19]
10     if (!hasIP(ip)) { //Packet not arrived
11         Host nextHost = null;
12         List<Host> directlyAccessibleHosts = getDirectlyAccessibleHosts();
13         // [λ=1, ι=2, ν=1, μ=2, CCR=5, NMCC=3]
14         for (Host directlyAccessibleHost: directlyAccessibleHosts) {
15             // [λ=2, ι=1, ν=0, μ=1, CCR=3, NMCC=1]
16             if (directlyAccessibleHost.hasIP(ip)) // If the packet is for a neighbour,
17                                                   // We send it to him
18                 nextHost = directlyAccessibleHost;
19         }
20         // [λ=1, ι=5, ν=4, μ=4, CCR=13, NMCC=9]
21         if (nextHost != null) {
22             nextHost.routeAPacketTo(ip, ttl − 1, usedHosts);
23         // [λ=1, ι=4, ν=4, μ=4, CCR=13, NMCC=9]
24         } else {//We have to look in the routing table
25             List<Host> directlyAccessible = getDirectlyAccessibleHosts();
26             IPAddress nextIP = this.getRoutingTable().getNextHop(ip);
27             boolean nextHostFound = false;
28             // [λ=2, ι=2, ν=1, μ=2, CCR=7, NMCC=3]
29             for (Host aDirectlyAccessible: directlyAccessible) {
30                 // [λ=3, ι=1, ν=0, μ=1, CCR=4, NMCC=1]
31                 if (aDirectlyAccessible.hasIP(nextIP)) { //Search the nextHop host object
32                     aDirectlyAccessible.routeAPacketTo(ip, ttl − 1, usedHosts);
33                     nextHostFound = true;
34                 }
35             }
36             // [λ=2, ι=1, ν=0, μ=1, CCR=3, NMCC=1]
37             if (!nextHostFound) { //Routing problem
38                 throw new Exception("Routing problem, there is no route corresponding
39                                     to the packet or the destination host is on the internet");
40             }
41         }
42     }
43 }
```

Fig. 3: Method `routeAPacketTo` in the `Host` class of the open-source project `cybercaptor-server`, used as running example in the project. Comments show the CC metrics for those statements that contribute to the method's cognitive complexity.

$$\min \sum_{i=1}^{n} x_i \tag{4}$$

$$\min c_M - c_m \tag{5}$$

$$\min t_M - t_m \tag{6}$$

subject to:

$$x_i + x_j \leq 1 \;\; \forall i \nleftrightarrow j \tag{7}$$

$$\underbrace{NMCC_i \cdot x_i}_{\text{Term A}} - \underbrace{\sum_{j, j \to i} CCR_{j \to i} \cdot z_{ji}}_{\text{Term B}} \leq \tau \;\; \forall i = 0, 1, \ldots, n \tag{8}$$

$$z_{ji} + |\{l | j \to l \to i\}| \, (z_{ji} - 1) \leq x_j - \sum_{l, j \to l \to i} x_l \;\; \forall j \to i \tag{9}$$

$$c_M \geq NMCC_i \cdot x_i - \sum_{j,j \to i} CCR_{j \to i} \cdot z_{ji} \;\; \forall i = 0, 1, \ldots, n \tag{10}$$

$$c_m \leq \tau(1 - x_i) + NMCC_i \cdot x_i - \sum_{j,j \to i} CCR_{j \to i} \cdot z_{ji} \;\; \forall i = 0, 1, \ldots, n \tag{11}$$

$$t_M \geq LOC_i \cdot x_i - \sum_{j,j \to i} LOC_j \cdot z_{ji} \;\; \forall i = 0, 1, \ldots, n \tag{12}$$

$$t_m \leq LOC_0(1 - x_i) + LOC_i \cdot x_i - \sum_{j,j \to i} LOC_j \cdot z_{ji} \;\; \forall i = 0, 1, \ldots, n \tag{13}$$

$$x_0 = 1 \tag{14}$$

$$x_i \in \{0, 1\} \;\; \forall i = 0, 1, 2, \ldots, n \tag{15}$$

$$z_{ji} \in \{0, 1\} \;\; \forall j \to i \tag{16}$$

$$t_M, t_m, c_M, c_m \in \mathbb{R}^+ \tag{17}$$

where the goal is to minimize the objective functions defined by Eq. (4), that is, the number of extractions, Eq. (5), which represents the difference between the highest and the lowest CC of all the extracted methods, and Eq. (6), which represents the difference between the highest and the lowest LOC of all the extracted methods. For Eqs. (5) and (6), c_M, t_M (respectively c_m, t_m), are the maximum (respectively minimum) CC and LOC reached by all the extracted methods, respectively. These two new objectives ensure that both the amount of CC and LOC are balanced after the extract method refactoring.

From the base model [3], we know that n is the number of feasible extractions and x_i is a binary variable that takes the value 1 if the i-th extraction is extracted, and 0 otherwise. Eq. (7) ensures that at most one of two feasible extractions in conflict will be selected in the final solution. It avoids choosing two feasible extractions that are in conflict simultaneously. Eq. (8) limits the CC of all the extractions in the code after applying the refactoring operations to be lower than or equal to τ. On the left-hand side, we find the CC of the ith extraction when extracted as a new function/method (Term A) minus the CC removed from extraction i due to code extractions j contained in i (Term B). If there were two feasible extractions k and j with $k \to j \to i$ that are extracted, one should only subtract the jth extraction CC contribution, and not k. The reason is that the contribution of extraction k is already considered in the contribution of j. For this reason, x variables are not used in the expression of the CC reduction (Term B). Instead, new variables z_{ji} are defined for $j \to i$ that are one if and only if the jth feasible extraction is selected and no other extraction strictly between j and i is selected. If $z_{ji} = 1$, then one should subtract the CC of the jth extraction from the CC of the ith extraction. This is what Eq. (8) does. Variables z are completely determined by the values of the x variables. Eq. (9) expresses the relationship between the z variables and the x variables. If $z_{ji} = 1$, then $x_j = 1$ and $x_l = 0$ for all l with $j \to l \to i$. If $z_{ji} = 0$, then there is no constraint for the x variables. Eq. (14) forces the CC reduction of the original function/method (represented with the 0th extraction). Finally, Eqs. (15) and (16) define the binary domain of the variables.

Eqs. (5), (6), (10), (11), (12), (13), and (17) were added to the single-objective model to add the new objectives. Eq. (5) is the objective of balancing the CC throughout all the extractions after the extract method refactoring is performed, and Eq. (6) does the same for balancing the LOC. Constraints (10) and (11) make c_M the upper bound of the CC of the extracted methods and c_m the lower bound. Similarly, Constraints (12) and (13) make t_M and t_m the upper and lower bounds of the LOC of the extracted methods, respectively. Minimizing the difference between the two variables, for both CC and LOC, means that, in an optimal solution, these variables attain maximum and minimum values, respectively.

4.1 Multi-objective optimization algorithms

In this section, we describe the two exact multi-objective optimization algorithms used in the experimental section to solve the ILP problem. The first algorithm is *augmented ε-constraint* or AUGMECON, inspired by the one proposed by Mavrotas [12], and is shown in Algorithm 1. This algorithm solves one sub-problem in each iteration, and each solution obtained is guaranteed to be efficient. The algorithm begins by creating the payoff table, which contains the lexicographic optima for each objective function. In general, the lexicographic optimization of a series of objective functions is to optimize the first objective function and then, among the possible alternative optima, optimize the second objective function and so on [12]. Practically, the lexicographic optimization is performed as follows: we optimize the first objective function, obtaining $\min f_1(x) = f_1(z)$. Then, we optimize the second objective function, $\min f_2(x) = f_2(z)$, subject to $f_1(x) \leq f_1(z)$, to maintain the optimal solution of the first optimization. Assume that we obtain $\min f_2(x)$. Finally, we optimize the third objective function $\min f_3(x) = f_3(z)$, subject to $f_1(x) \leq f_1(z)$, $f_2(x) \leq f_2(z)$ to preserve the previous optimal solutions. For each iteration, Algorithm 1 sets a positive value for the λ constant, which must be small enough to prevent the algorithm from omitting some of the efficient solutions, and large enough to avoid numerical problems. In general, it is sufficient to take a value of λ in $[10^{-6}, 10^{-3}]$. The main limitation of the AUGMECON algorithm is that it does not have anytime convergence; therefore, if execution time is limited, the solutions obtained tend to be concentrated in a single region of the Pareto front, and they also depend on the correct tuning of the λ parameter.

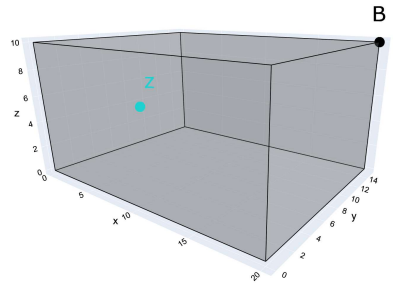
To compare the performance of the AUGMECON algorithm to an anytime algorithm (where the solutions obtained are well distributed at any time of execution), the second algorithm we propose to solve the multi-objective ILP problem is a simplification of the MOCO problem-solving algorithm using full p-split [13], see Algorithms 2 and 3. Full p-split is a method for decomposing the search space based on the objective space, utilizing an algorithm called the *Hybrid Method* [14].

Algorithm 2, which we will refer to as Hybrid Method, divides the objective space into boxes and explores them one at a time. Each time it finds a new solution within a box, all boxes containing that solution are divided, and those containing only dominated solutions are discarded. The remaining boxes may overlap, but their union occupies a lower volume than the original one. Figure 4 shows an example of the first

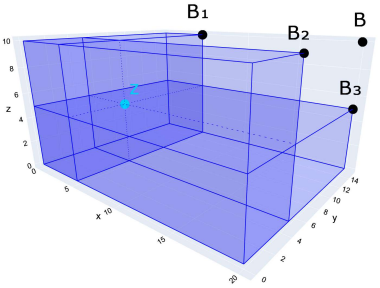
Algorithm 1: Augmented ε-constraint (AUGMECON)

Output: PF
Create payoff table (lexmin $f_k(x)\ \forall k = 1, ..., 3$)
Set lower bounds from the payoff table $lb_k\ \forall k = 2, 3$
Set upper bounds $ub_k\ \forall k = 2, 3$
Calculate ranges $r_k = ub_k - ul_k\ \forall k \in \{2, 3\}$
for $i \in ub_2, ..., ul_2$ **do**
  $j \leftarrow ub_3$
  while $\exists x \in X, j \geq ul_3$ **do**
    $z \leftarrow$ solve $\left\{\min f_1(x) - \lambda \cdot \frac{s_k}{r_k},\ \text{s.t. } f_k(x) + s = \varepsilon\ \forall k \in \{2, 3\}, x \in X\right\}$
    if *exists(z)* **then**
      $PF = PF \cup \{z\}$
      $j \leftarrow j - 1$
    else
      $j \leftarrow ul_3 - 1$
    end
  end
end

partition of full p-split.[10] If the algorithm ends before any other stopping criterion (for example, a time limit), then the complete PF is obtained, including unsupported solutions.



(a) Box $\mathbf{B} = [(0, 0, 0), (20, 15, 10)]$, with non-dominated solution $z = (5, 5, 5)$.



(b) After applying *full-p-split*: $\mathbf{B_1} = [(5, 15, 10)]$, $\mathbf{B_2} = [(20, 5, 10)]$, $\mathbf{B_3} = [(20, 15, 5)]$. The lower bounds of the boxes are all $ID = [(0, 0, 0)]$.

Fig. 4: Partitioning a box using *full-p-split*.

To solve the multi-objective problem, Algorithm 2 takes as input the initial box, i.e., the entire objective space. One example of an initial box $B = [(0, 0, 0), (20, 15, 10)]$ is depicted in Figure 4a. The Hybrid Method algorithm finds the complete PF by applying the multi-objective ILP model to each box obtained. An example of the first iteration of full p-split is depicted in Figure 4b. As a result of the full p-split, one box

[10] An interactive version of the example can be found at https://doi.org/10.5281/zenodo.15682834

may contain another one, so Lines $4-18$ of Algorithm 3 iterate over all pairs of boxes and eliminate those boxes that are contained in larger boxes.

Algorithm 2: Hybrid Method using full p-split

Input: B_0
Output: PF

$PF \leftarrow \varnothing$
$\mathcal{U} \leftarrow [B_0]$
$P(u) \equiv \left\{ \min \left(\sum_{i=1}^{p} f_i(x) \right), \text{ subject to } f(x) \leq u, x \in \mathcal{X} \right\}$
while $(\mathcal{U} \neq \varnothing)$ **do**
  $B \leftarrow \text{argmax}_{B \in \mathcal{U}}(v(B))$ `// v(B) denotes the volume of box B`
  if $(P(B)$ *has solution* $)$ **then**
    $x^* \leftarrow$ Optimal solution of $P(B)$
    $PF \leftarrow PF \cup \{f(x^*)\}$
    $\mathcal{U} \leftarrow \text{RE}(\mathcal{U}, f(x^*))$ `// See Algorithm 3`
  else
    $\mathcal{U} \leftarrow \mathcal{U} - B$
  end
end

Algorithm 3: RE (Redundancy elimination)

Input: $\mathcal{U}, z$ `// Set of boxes and solution found`
Output: $\mathcal{U}$ `// New boxes set`

$\mathfrak{R} \leftarrow \{u \in \mathcal{U} : z < u\}$ `// Boxes that contain z`
for $j \in 1, ..., p$ **do**
  $\mathfrak{D}_j \leftarrow \{u \in \mathcal{U} : z_j = u_j \wedge z_k < u_k, \forall k \neq j\}$
  $\mathfrak{P}_j \leftarrow \varnothing$
end
for $u \in \mathfrak{R}$ **do**
  for $j \in 1, ..., p$ **do**
    $\mathfrak{P}_j \leftarrow \mathfrak{P}_j \bigcup \{(u_1, ..., u_{j-1}, z_j, u_{j+1}, ..., u_p)\}$
  end
end
for $j \in 1, ..., p$ **do**
  $\mathfrak{P}_j \leftarrow \{z' \in \mathfrak{P}_j : z' \not\leq u', \forall u' \in \mathfrak{P}_j \cup \mathfrak{D}_j - \{z'\}\}$ `// Filter redundant boxes`
end
$\mathcal{U} \leftarrow (\mathcal{U} - \mathfrak{R}) \bigcup \left(\bigcup_{j=1}^{p} \mathfrak{P}_j \right)$

The main limitation of the Hybrid Method lies in the computational overhead introduced by partitioning the objective space, as the number of boxes and their overlaps grow rapidly as solutions are found, requiring a computationally intensive, ongoing process of filtering out redundancies.

5 ILP CC reducer approach

We propose an "ILP CC reducer tool". It is outlined in Figure 5. It first takes a CSV file containing the *refactoring cache of a method*. Then, using the refactoring cache processing module, the intermediate files needed to solve each MO-ILP problem are generated. Finally, the ILP CC reducer module solves the problem based on the user's specifications and generates the results, including statistics and some visuals.

The refactoring cache of a method is a list of extract-method refactoring opportunities to reduce its CC. Thus, the refactoring cache contains relevant information about candidate code extractions over the original method. This file is generated from the Java tool developed by Saborido et al. [2].

Second, the `refactoring cache processing module` automatically generates the following four CSV files, which are needed for the `ILP CC reducer module`:

(i) `MethodName`[11]`_extractions.csv` stores the index of each feasible extraction (i), the LOC_i of the extraction, and the $NMCC_i$ of each extraction. There is a fourth column that stores the number of parameters that each feasible extraction would need when called (currently unused).
(ii) `MethodName`[11]`_nested.csv` stores the indices of the child extractions (j), the indices of their parents (i), and the $CCR_{j \to i}$ for each relationship.
(iii) `MethodName`[11]`_conflict.csv` stores the extractions in conflict. The conflict extractions are paired in the two columns of the file.
(iv) `MethodName`[11]`_feasible_extractions_offsets.csv` stores the index of each extraction (i), its start offset, and its end offset. These offsets indicate the start and end positions of code extractions within the original file.

Thirdly, using the generated CSV files, the "ILP CC reducer tool" models the ILP optimization problem and solves it. This module takes as inputs from the user:

(i) The number of objectives, which sets the number of objectives to consider (from one to three).
(ii) The algorithms selected for solving the ILP problem, that is, `EpsilonConstraintAlgorithm` or `HybridMethodAlgorithm`[12].
(iii) The threshold, i.e., the maximum CC that all extractions must have after the extraction has been performed, including the original method.
(iv) An objectives list, which is an optional input, and that is an ordered list of the objectives, between $EXTRACTIONS$, CC and LOC (the default order is `extractions`, `extractions` and `CC`, or `extractions`, `CC` and `LOC`, for one, two or three objectives, respectively).

The tool generates the corresponding model based on the parameters specified in the ILP CC reducer module's input. As shown in Figure 5, the output is a CSV file containing the set of optimal solutions, statistics for each solution, and various visual content depending on the selected algorithm. One optimal solution to the MO-ILP

[11] `MethodName` is the name of the method under processing and not the literal "MethodName".
[12] The current version of the tool also supports a weighted-sum algorithm and a single-objective algorithm.

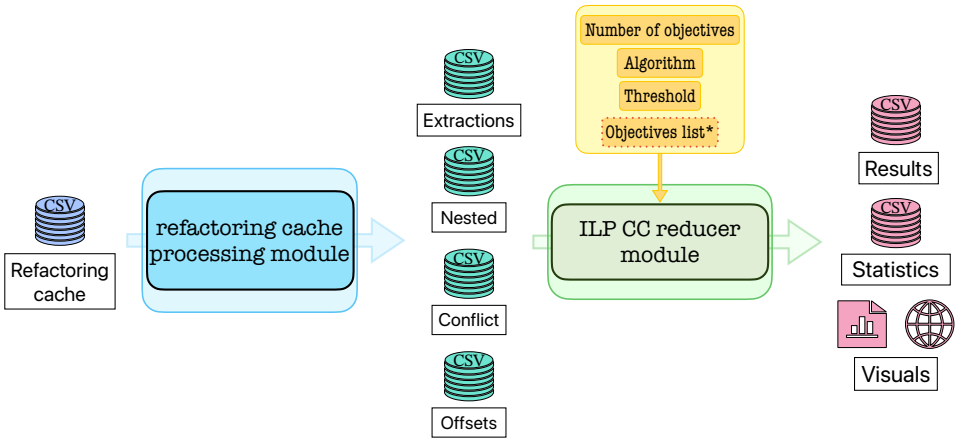


Fig. 5: Overview of the proposed "ILP CC reducer tool". *`Objectives list` is optional.

problem is a sequence of extractions that has a given impact on the user-specified objectives and that reduces the CC of the initial method to or below the given threshold.

The "ILP CC reducer tool" is developed in Python, using Pyomo to model the ILP problem. The ILP solver used in the tool is CPLEX. Pyomo[13] is an open-source Algebraic Modeling Language (AML) used for mathematical optimization. An AML is a language for expressing optimization models concisely and naturally, avoiding implementation details. It aims to standardize how optimization problems are modeled, making it easier to exchange models across users and communities. CPLEX[14] is a prescriptive analytics solution that accelerates the development and implementation of decision optimization models using mathematical and constraint programming. It has excellent compatibility with Python. Moreover, it is a software with a long history of development, which has enabled it to achieve a high degree of maturity, robustness, and efficiency in solving optimization problems.

The core of the "ILP CC reducer tool" is the `ILP CC reducer module`. Its architecture is shown in Figure 6, which is organized into three main modules that are (i) model, defining the corresponding combination of objective or objectives and constraints, (ii) ILPengine, that is, the main component that connects models and algorithms and manages the execution flow, and (iii) algorithms, implementing the optimization strategies explained in Subsection 4.1. The rest of the modules presented in Figure 6 are (i) utils, i.e., auxiliary tools for managing input and output data, (ii) `original_method`/`refactored_method` contain the original and refactored methods from a functional perspective, and are the input and output of the module, respectively, (iii) Pyomo, which is the modeling framework used to build the mathematical models, and (iv) CPLEX, used as a solver for ILP problems.

[13] https://pyomo.readthedocs.io/en/stable/ (last accessed March 2026)
[14] https://www.ibm.com/es-es/products/ilog-cplex-optimization-studio (last accessed March 2026)

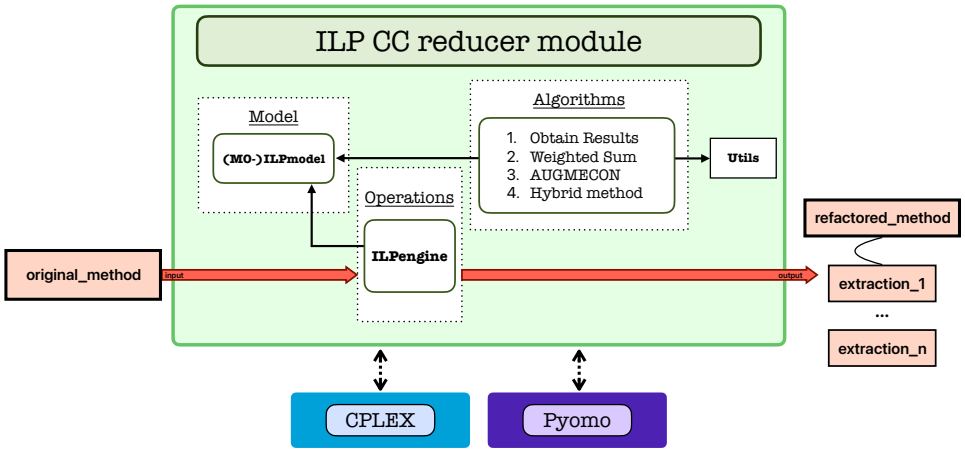


Fig. 6: Architecture of the ILP CC reducer module.

6 Empirical validation

This section presents the results for the AUGMECON and the Hybrid Method algorithm. The empirical comparison will help assess how execution time limits affect the performance of both methods, contrasting the Hybrid Method's rapid solution convergence with AUGMECON's lower computational cost per iteration. They were used to solve the multi-objective ILP problem for 14 methods from an industry project at Ayesa Advanced Digital Services Limited (ASDA) and 1,046 methods from 10 open-source projects.

6.1 Experimental setup

We conducted the experiments on an Intel© Xeon© Gold 6240R CPU machine with 220 GB of RAM and 96 cores, running Ubuntu 20.04.6. We set the cognitive complexity threshold to the default value proposed by SonarQube ($\tau = 15$). AST processing and Extract Method refactorings were performed through Eclipse JDT version 3.31.0 and Eclipse LTK version 3.10.0. To solve ILP optimization problems, we used Python 3.12.0 and the Pyomo library for modeling, and IBM ILOG CPLEX Optimization Studio V22.1.1, accessed through the IBM Academic Initiative, to solve each ILP model. To preserve the integrity of the runtime environment and ensure isolated resource management, the process has been implemented using Docker container technology. In this Dockerized environment, an overall time limit of 3600 seconds was set for solving the problem, so that in each iteration, the remaining time is provided to CPLEX as an updated stopping criterion. To avoid excessive computation or getting stuck in specific branches of the search tree, even after restricting execution to a single processing thread, each container was programmed to delete itself after 4 hours of execution automatically. If the algorithm used ends, then the whole PF is obtained.

6.2 Objects of study

To evaluate the performance of the proposed approach, several methods, both from open-source and industry projects at Ayesa Advanced Digital Services Limited (ASDA), have been selected. Fourteen methods from an industry project from Ayesa are studied. We utilize a diverse set of 10 open-source projects, selected from the benchmark by Saborido et al. [2]. These projects comprise two widely used frameworks for multi-objective optimization, five platform components designed to accelerate smart-solution development, and three popular open-source projects, each with over 12,000 stars and more than 1,000 forks. These open-source projects are shown in Table 1. It is also specified the commit hash for each open-source project. The entire study focuses on Java projects because our industrial partner, ASDA, required it, but the tool developed for this paper can be used with any other programming language.

Table 1: Open-source projects, their respective commits used for experimentation, and the number os issues considered for each project.

Project	Commit	#Issues considered
ByteCode	55bfc32	57
Cybercaptor	b6b1f10	35
FastJson	93d8c01	230
Fiware-Commons	f83b342	4
IoTBroker	98eeceb	10
Jedis	cfc227f	3
jMetal	e6baf75	62
Knowage-core	dfed28a	557
MOEA-Framework	223393f	82
QueryExecution	c032e5a	6

7 Results

This section presents the results obtained with the AUGMECON and Hybrid Method algorithms for both the ASDA projects and the ten open-source projects under study.

Figure 7 shows the number of issues (methods) that each algorithm manages to solve compared to the total number of instances studied for each project, both open-source and ASDA's industrial project. The main takeaway from this plot is clear: the Hybrid Method consistently solves more instances than or as many as AUGMECON in all scenarios. In projects with few instances, such as Fiware-Commons, Jedis, or Query-Execution, both algorithms solve all the instances. The advantage of the Hybrid method is evident in larger-scale projects, such as Knowage-core or FastJson. In total, AUGMECON solved 747 (70.47%) instances, and the Hybrid Method solved 782 (73.77%) instances out of the 1,060 instances studied.

Figure 8 presents a series of statistics regarding the solutions obtained, including the number of solutions found for each method, the runtime of the respective instance, the number of extractions (equation (4)) performed on each method, the difference

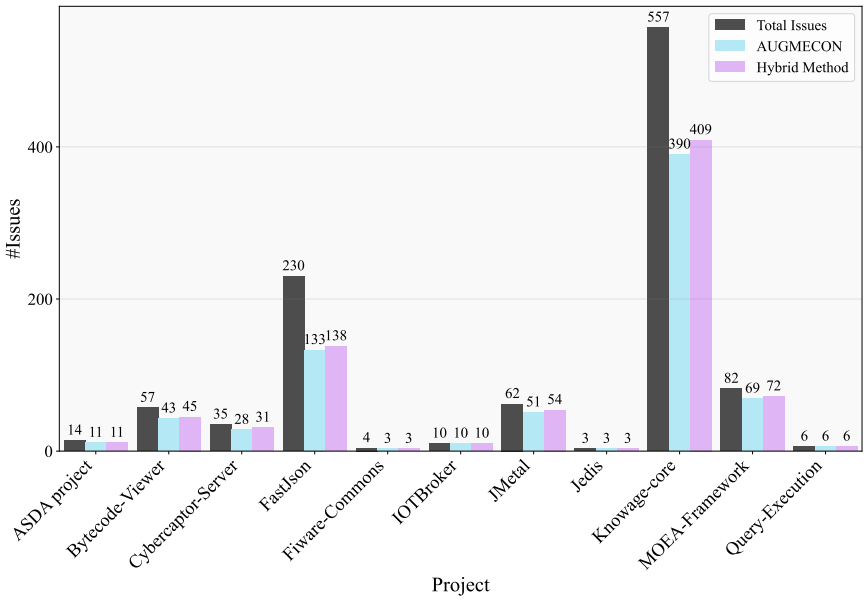


Fig. 7: Number of issues solved for each algorithm.

in CC (equation (5)) between the method with the lowest and highest CC for each method obtained in a solution, and the difference in LOC (equation (6)) between the method with the lowest and the highest LOC for each method obtained throughout each solution, always taking into account the refactored original method and all extractions for each solution. These metrics are shown for both algorithms considered.

In principle, the values associated with the solutions (number of solutions, extractions, CC_{diff}, and LOC_{diff}) should be equivalent for both algorithms, since they describe properties of the problem's Pareto front. Therefore, the only expected difference between the two algorithms should be the runtime. However, slight discrepancies are observed between the solution metrics, which is because, in some instances, the complete Pareto front is not obtained because of the 3,600-second global time limit set in CPLEX for each algorithm across all iterations. Since both algorithms exhibit different performance, they may explore the solution space unevenly within that time limit, leading to differences in the results. The number of solutions ranges from 1 to 28 for AUGMECON and from one to 26 for the Hybrid Method.

Regarding runtime, 50% of the instances are solved in less than 24.47 seconds for AUGMECON and in less than 10.88 seconds for the Hybrid Method, although in both cases some runs exceed the maximum allowed time. The maximum execution time for each algorithm is 3,622.39 seconds for AUGMECON and 3,642.59 seconds for the Hybrid Method algorithm. They both exceed the maximum stopping criterion because CPLEX only checks the clock after completing indivisible tasks (such as solving a node), so it will finish that operation before stopping. In addition, the program requires a few extra seconds of post-processing to organize the best solution and free up the memory it has used. The total runtime for each algorithm is 171.11

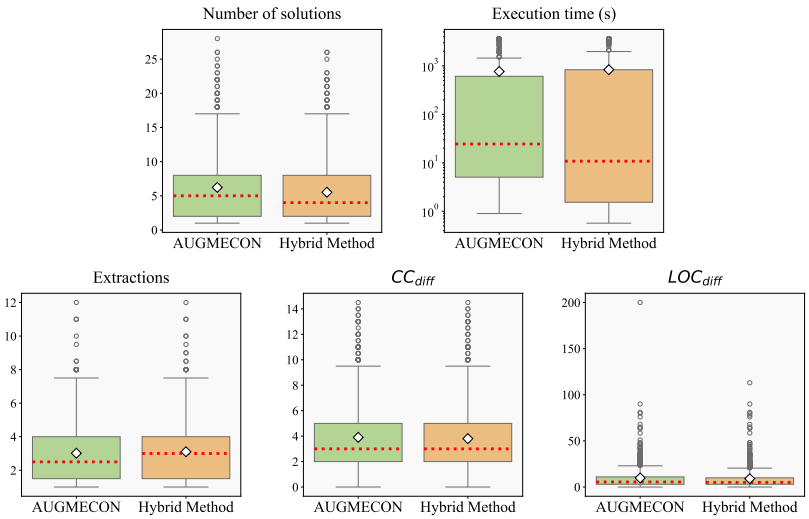


Fig. 8: Statistics for different metrics obtained after solving the MO-ILP problem. The average is represented with a white diamond.

hours for AUGMECON and 194.02 hours for the Hybrid Method algorithm. It is important to clarify an apparent contradiction in overall execution times: the Hybrid Method algorithm recorded a longer total execution time. This overall increase is not due to the algorithm being slower or encountering problems in other instances, but quite the opposite: the Hybrid Method successfully solves a greater number of instances than AUGMECON (see Figure 7).

The number of extractions per method ranges from 1 to 12 in both approaches. The difference in CC between the methods with the lowest and highest CC for each resulting method and solution has a median of 3 in both cases, with maximum values of up to 14.5. Meanwhile, the difference between the methods with the lowest and highest LOC has a median of 5.5 for AUGMECON and 5 for the Hybrid Method, reaching maximum values of 200 and 113, respectively. Overall, these results show consistent behavior across both approaches, with the observed differences arising from the algorithms' differing performance under the imposed time limit.

Figure 9 shows a comparison of the total execution time for each algorithm across all the projects under study. On a logarithmic scale, it can be seen that the Hybrid Method has a median execution time that is clearly lower than that of AUGMECON, indicating that, in general, it solves common issues more quickly. A notable aspect is that, in most cases, such as FastJson, Fiware-Commons, or IoTBroker, the Hybrid Method algorithm tends to exhibit lower median times, indicating faster execution for the problems solved by both algorithms. It is also observed that AUGMECON tends to show greater variability, with higher boxes or the presence of outliers in several projects. However, in some specific cases (such as Query-Execution or Knowage-core

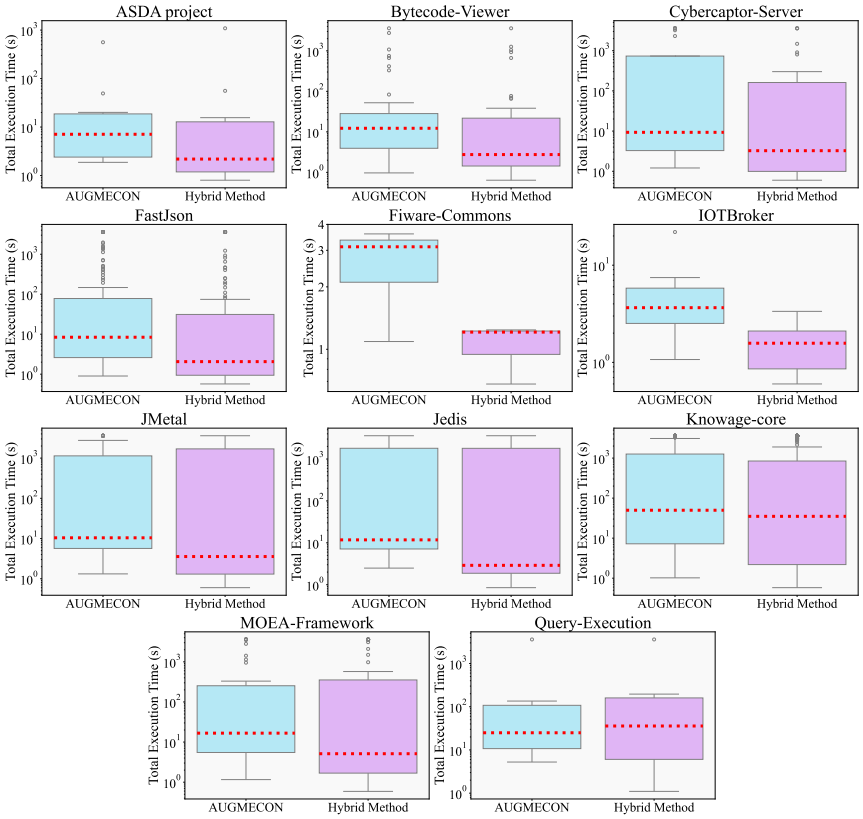


Fig. 9: Total execution time per project (Y-axis on a logarithmic scale).

projects), the differences between algorithms are smaller or even slightly in favor of AUGMECON. Overall, the graphs suggest that the Hybrid Method maintains more consistent performance and is generally more time-efficient across the different projects studied.

We compute the hypervolume (HV) as a reference for future applications that attempt to solve the same problem. This will ease comparisons among different algorithms. HV is the volume of the n-dimensional space that is dominated by a set of points [15], and that dominates a given reference point. It satisfies numerous mathematical properties and provides a single scalar value that simultaneously reflects both the diversity of solutions along the PF and their proximity to the Pareto-optimal front (i.e., convergence). The HV has been normalized to ensure comparability across different instances and objective scales. As a result, the reported values are bounded between 0 and 1. The normalization process consists of dividing the absolute HV obtained with the `Pymoo` library by the total volume of the box defined by the ideal (i.e., the bests values for each objective taking into account all the solutions generated) and the reference point $Reference = (na_1 + 1, ..., na_n + 1)$, where na is the nadir (i.e., the worsts values for each objective taking into account all the solutions found).

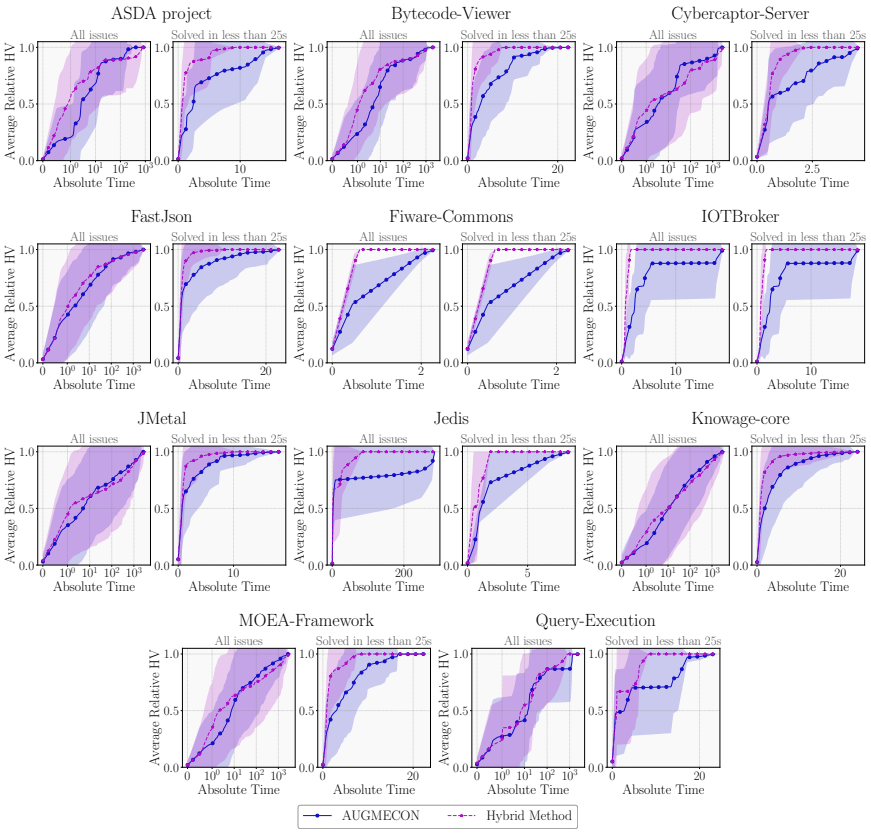


Fig. 10: Comparing the performance of the AUGMECON and Hybrid Method algorithms using the relative HV over time.

To evaluate the performance of both algorithms, we analyzed the trend of the relative HV over absolute execution time in Figure 10. Higher values indicate a better approximation to the PF. To ensure a fair comparison, a matching filter was applied during data selection: only instances successfully resolved by both algorithms were included. Thus, we excluded runs in which only one algorithm completed the task. The visual evaluation has been divided into two scenarios to study the impact of instance complexity: (i) an unrestricted overview (All issues) and (ii) a view restricted to the fastest runs (Solved in less than 25 seconds). The 25-second threshold is not arbitrary; rather, it was established based on the higher median execution time between the two algorithms, that is, the median for AUGMECON, which is 24.47. Once again, the matching filter was applied, ensuring the same issues were accounted for in both algorithms. When examining instances solved within this 25-second limit, it becomes evident that the Hybrid Method can find the complete Pareto front significantly faster than AUGMECON. This advantage is particularly noticeable for Pareto fronts that require little computation time, typically associated with fewer solutions. However, when analyzing the overall scenario without time constraints (where a logarithmic

scale is used to represent the largest and most computationally intensive instances), both algorithms ultimately perform similarly. As the resolution time increases, the initial gap disappears, and AUGMECON matches the convergence rate of the Hybrid Method algorithm. The most notable aspect of this multiple comparison is the significant differences observed in projects with a high issue density and very long execution times, such as Jmetal, Knowage-core, and MOEA-Framework.

7.1 Impact on software

Table 2 shows, for each project, the number of cognitive complexity issues detected (#Issues), the total time required to fill the refactoring cache and run CPLEX (Runtime (h)), as well as the initial technical debt (In. debt (h)) and final technical debt (Final debt (h)) calculated from the values provided by SonarQube. Technical debt represents the estimated effort required by developers to fix code issues; it reflects the cumulative cost of shortcuts and compromises in code quality that must eventually be addressed. According to SonarQube's definition of the metric, the time unit in days refers to an 8-hour workday, so a reported effort of 1d 7h translates to 15 hours of developer work. The Runtime and Final debt columns include two variants, indicated by the letters "AUGMECON" and "HM", which reflect the results obtained by the two applied algorithms. Runtime shows the computational effort required to solve the MO-ILP model. At the same time, initial technical debt represents the estimated effort required by developers to fix the issues before applying the multi-objective model. Final technical debt indicates the effort required to address issues that the MO-ILP approach cannot resolve. Therefore, the actual reduction in technical debt, that is, the time saved by the development team, is calculated by subtracting the final debt from the initial debt. Taken together, the table allows a comparison, by project, between the computational load and the effectiveness in reducing technical debt, providing a clear view of the multi-objective approach's performance across different scenarios.

From a software engineering perspective, the results have a direct, highly positive impact on maintenance. Traditionally, refactoring to reduce cognitive complexity consumes a massive amount of human resources, diverting developers from creating new features. Automating this process using the proposed tool results in significant time savings. Globally, the application of AUGMECON saves a total of 259.48 hours of manual work, while the Hybrid Method (HM) algorithm exceeds this figure, achieving a total savings of 275.7 hours of development effort. This translates to approximately 34 full workdays that the engineering team can devote to higher-value tasks.

A more detailed analysis by project allows us to evaluate the tool's "return on investment"; that is, to compare the computational cost (runtime) against the savings in human effort (reduced debt), identifying where its application is most cost-effective:

(i) High-return scenarios: in projects like FastJson, the Hybrid Method consumes 17.16 machine hours but reduces human labor by a staggering 73.48 hours. Similarly, in smaller projects like IoTBroker or ByteCode, the computational runtime is minimal compared to the development hours saved. In these cases, adopting the tool is undeniably advantageous, as the cost of machine time (which is unattended) replaces days of manual labor.

Table 2: Execution time and technical debt reduction. Runtime is the total time to fill the cache and run CPLEX. Initial and final technical debt are reported in work hours.

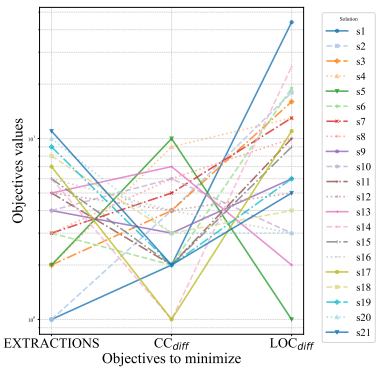
Project	#Issues	Runtime (h)		In. debt (h)	Final debt (h)	
		AUGMECON	HM		AUGMECON	HM
ASDA's project	14	1.20	1.34	4.17	1.73	1.30
ByteCode	57	4.31	6.71	25.13	11.12	9.32
CyberCaptor	35	6.77	7.16	15.48	7.00	5.50
FastJson	230	17.07	17.16	181.90	111.67	108.42
Fiware-Commons	4	0.01	0.01	0.60	0.25	0.25
IoTBroker	10	0.03	0.02	2.37	0.00	0.00
Jedis	3	1.28	1.28	1.05	0.00	0.00
jMetal	62	13.81	17.27	26.93	13.52	11.12
Knowage-core	557	110.08	123.64	261.28	142.58	128.73
MOEA-framework	82	15.45	18.00	24.80	6.93	4.95
QueryExecution	6	1.09	1.11	1.57	0.00	0.00
Total	1,060	171.11	194.02	545.28	294.80	269.58

(ii) Break-even scenarios: in extremely complex projects with hundreds of issues, such as Knowage-core, execution time increases significantly (123.64 hours for the Hybrid Method algorithm). However, it reduces technical debt by 132.55 hours. Although the ratio of machine hours to human hours saved is nearly 1:1, the implications for maintenance remain overwhelmingly favorable: leaving a server running an automated refactoring process over a weekend remains a far more efficient and cost-effective strategy than tying up a software engineer for more than three weeks to perform the same work manually and error-prone.

These results demonstrate that the proposed multi-objective approach is not merely a theoretical exercise, but a practical tool that substantially reduces the workload associated with software maintenance and lowers operational costs.

The method `obtenerTablasDinamicas` from ASDA's industrial project is shown in Figure 11. This method is the one with the most solutions obtained among those depicted from ASDA. Results are presented in parallel coordinate plots (to show the Pareto fronts), numerical tables, and three-dimensional PF plots. A parallel coordinates plot is an intuitive way to represent results for multivariate data, i.e., data with multiple variables or dimensions. It is a practical way to represent PF solutions, as it displays the results for each objective function and connects them with straight lines, allowing all solutions to be observed simultaneously.

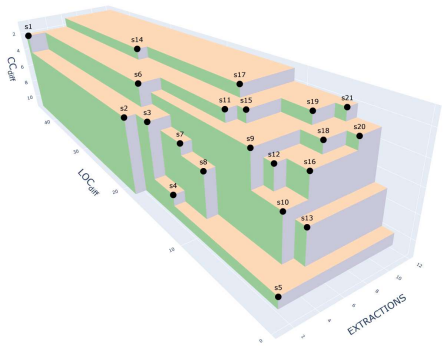
Subfigure 11a shows the parallel coordinates plot for each PF; Table 11b shows the list of Pareto-optimal solutions; Subfigure 11c shows a 3D visualization of the PF, and Subfigure 11d shows two projections of the PF to make it easier to compare 2D



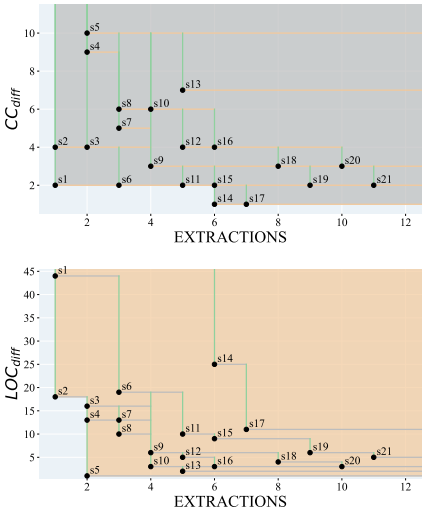
(a) Parallel coordinates.

Objective / Solution	*Extractions*	CC_{diff}	LOC_{diff}
s1	1	2	44
s2	1	4	18
s3	2	4	16
s4	2	9	13
s5	2	10	1
s6	3	2	19
s7	3	5	13
s8	3	6	10
s9	4	3	6
s10	4	6	3
s11	5	2	10
s12	5	4	5
s13	5	7	2
s14	6	1	25
s15	6	2	9
s16	6	4	3
s17	7	1	11
s18	8	3	4
s19	9	2	6
s20	10	3	3
s21	11	2	5

(b) Points of PF.



(c) 3D PF visualization.



(d) 3D PF projections.

Fig. 11: 3D PF obtained by the Hybrid Method algorithm for the `obtenerTablasDinamicas` method from ASDA's project.

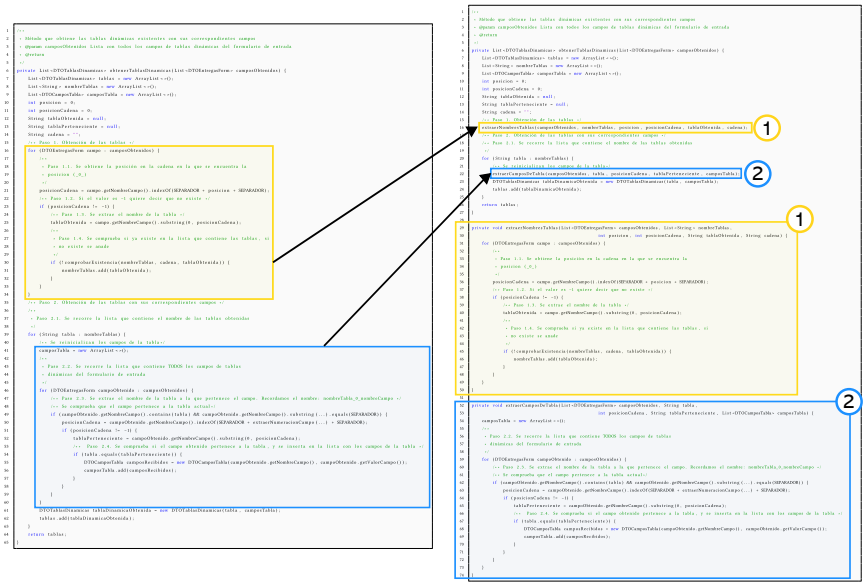


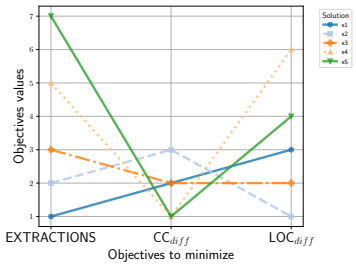
Fig. 12: Extract method refactoring applied to solution **s5** in `obtenerTablasDinamicas` method from ASDA's industrial project.

solutions. All plots can be found in Zenodo[15]. If the solution that best balances LOC in Figure 11 is chosen, i.e., the second solution (**s5**), the corresponding extractions are the ones in Figure 12. As shown in Figure 12, there are two extractions, and they are not nested within one another. In total, there are three resulting methods (the original method and the two extractions). The CC of each method, respectively, is 1, 6, and 11, so the difference between the maximum and minimum CC of all methods after the extractions have been performed is $11 - 1 = 10$. Similarly, the LOC of each resulting method is 20, 19, and 20, so the difference between the maximum and minimum LOC of all methods after the refactoring is $20 - 19 = 1$.

The solutions obtained for the running example in Figure 1 are presented in Figure 13. This second Pareto front has fewer solutions, since there are fewer possible combinations of code extraction.

If the solution that best balances LOC in Figure 13 is chosen, i.e., the second solution (**s2**), the corresponding extractions are the ones in Figure 14. As shown in Figure 14, there are two extractions, and they are nested within one another. In total, there are three resulting methods. The CC of each method, respectively, is 2, 5, and 4, so the difference between the maximum and minimum CC of all methods after the extractions have been performed is $5 - 2 = 3$. Similarly, the LOC of each resulting method is 7, 8, and 8, so the difference between the maximum and minimum LOC of all methods after the refactoring is $8 - 7 = 1$.

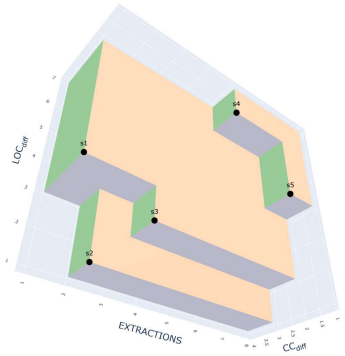
[15] https://doi.org/10.5281/zenodo.15682834



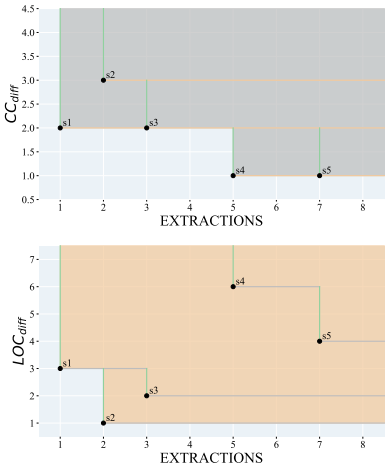
(a) Parallel coordinates.

Solution \ Objective	*Extractions*	CC_{diff}	LOC_{diff}
s1	1	2	3
s2	2	3	1
s3	3	2	2
s4	5	1	6
s5	7	1	4

(b) Points of PF.



(c) 3D PF visualization.



(d) 3D PF projections.

Fig. 13: `routeAPacketTo` method result from `Host` class in `cybercaptor-server` project. This is the solution for the method used for the running example in Figure 3.

7.2 Comparing mono-objective and multi-objective approaches

Figure 15 compares the performance of the previous single-objective ILP model [3] with that of the current MO-ILP approach for the 1,060 issues under study. The first MO-ILP boxplot shows the average value for each objective, calculated from the total set of solutions for each instance. The second MO-ILP boxplot is the best value that a specific objective can achieve when the model is solved by giving that objective absolute priority over the others.

If we focus on the original objective of minimizing extractions, the lexicographic optima of the MO-ILP, prioritizing the number of extractions, exactly match the performance of the mono-objective ILP model for the number of extractions. In contrast, the average of the MO-ILP solutions naturally exhibits slightly higher values and greater dispersion on this metric, reflecting the trade-off required when balancing all criteria simultaneously. The true enhancement of the new approach is evident in the two additional objectives: CC and LOC differences (balance). Since the original

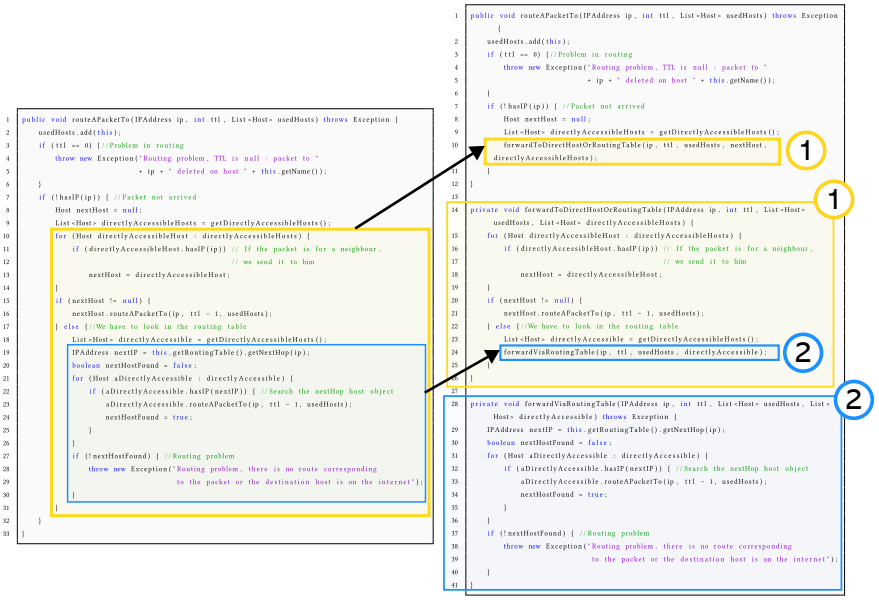


Fig. 14: Extract method refactoring applied to solution **s2** in method `routeAPacketTo` of the running example in Figure 3.

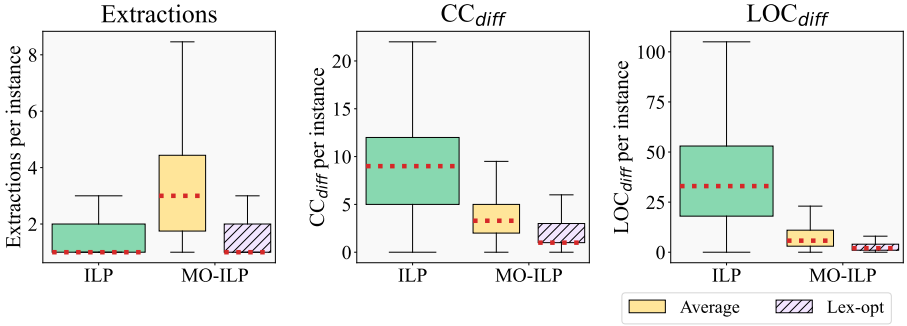


Fig. 15: Comparison of the mono-objective and multi-objective metrics results for each objective.

single-objective model completely ignored differences between the highest and lowest CC and LOC, it generated solutions with much higher values and greater variability. In contrast, the MO-ILP approach drastically compresses these ranges, reducing both the average values and the dispersion. The lexicographic optima of these new metrics reach highly concentrated minimum values, confirming that the multi-objective model achieves substantial improvements in refactoring quality in exchange for a very acceptable increase in the average number of extractions.

8 Related work

The problem of reducing CC has been approached in recent years in various ways. Saborido et al. [2] analyzed the difficulties faced by developers when deciding on the best method to extract in order to reduce the CC of Java code and proposed enumerative approaches for CC reduction. Recently, Saborido et al. modeled the software CC reduction task as an ILP optimization problem [3]. The purpose was to minimize the number of sentences extracted from a method, so that there is not a high quantity of extractions obtained. For each method with CC greater than a given threshold, their approach searches for sequences of applicable extract method refactoring operations. Finally, it outputs the changes to perform on each method. They reduced the CC to or below the default threshold in the majority of the issues under study, thereby improving search performance compared to their previous work, which used heuristic approaches.

The previous work, however, does not account for the balance of CC across the resulting methods or the length (LOC) of the extracted code. That is why the approach presented in this paper is novel: it balances CC and LOC, thereby allowing developers greater operational leeway to further edit the code after applying the tool presented in this work. This contribution is particularly relevant for collaborative, long-term projects, where code readability and maintainability are critical. If no single method has a high CC and none is significantly longer than the others, then the refactored code becomes more uniform and manageable. This approach facilitates future modifications and extensions. Furthermore, this balance provides developers with the flexibility to continue editing the code without compromising quality or easily exceeding CC thresholds again. To the best of our knowledge, no existing works or tools in the industry explicitly address this balance using a multi-objective ILP formulation, which positions this contribution as a significant advancement in the field of automated refactoring.

9 Conclusion and Future Work

We propose in this paper a new multi-objective ILP formulation to reduce software CC, minimizing the number of code extractions while balancing the LOC and CC of the original refactored method and its extracted methods. A novel software tool has also been designed and developed to ease the reduction of software's CC. This tool integrates several multi-objective ILP problem-solving algorithms, including an ε-constraint algorithm and a Hybrid Method algorithm, which traverse the entire objective space and can compute the complete Pareto front. The proposed multi-objective ILP model and the software tool have been validated using a benchmark comprising 1046 methods from ten open-source projects and 14 from an industrial Java software project. Efficient solutions have been obtained for each of them. Globally, our new approach achieves a total savings of 275.7 hours of development effort. This translates to approximately 34 full workdays that the developers can devote to higher-value tasks. Regarding the solver used, CPLEX offers high computational efficiency and advanced algorithms that enable the solution of large, complex problems quickly and accurately. This advantage enables the developed tool to be included in continuous

software integration processes. Our current work also has limitations. In particular, the reduction of the CC is achieved through the Extract Method refactoring operation. However, there are circumstances that prevent this refactoring from being applied. Some examples are the presence of "break" and "continue" statements in loops, or "return" in any position.

As future work, three main lines of development are proposed to extend and enrich this work. First, since assigning a representative name to the extractions is crucial, we can integrate LLMs into the method naming process [9]. The more extractions are performed, the more challenging it becomes to assign method names. The key is how to prompt the model to indicate all the requirements needed to infer method names adequately. Second, it is planned to design and conduct a series of surveys targeting expert and semi-expert users to assess the perceived quality and usefulness of the results generated by the proposed multi-objective ILP approach. The responses will be systematically compared with those from previous evaluations based on the single-objective ILP approach developed by Saborido et al. [3]. The purpose of this comparative study is not only to validate the improvements introduced by the multi-objective ILP formulation but also to gather qualitative insights into user preferences and expectations to guide future iterations of the system. Third, we plan to extend the experimental study, considering other programming languages and analyzing the scalability of the multi-objective ILP model, as adding numerous constraints can increase execution time to the point that the model becomes unsolvable in practice. In this case, the possibility of solving the model presented in this paper using alternative optimization techniques, such as MOEA/D, will be explored.

Supplementary information.

- "ILP CC reducer tool" on GitHub: https://github.com/anhe2p9/MOILP-ASCCR.git
- Results in Zenodo: https://doi.org/10.5281/zenodo.15682834

Acknowledgements. This research has been supported by MICIU/AEI/10.13039/501100011033 under projects PLEC2023-010266 (SOFIA), PID2022-142964OA-I00 (EGSVAI) and RED2022-134647-T (AI4Software). It has also been partially funded by European Regional Development Funds (ERDF) and the University of Malaga.

Appendix A Figures used for CC reduction problem definitions

This appendix contains the figures illustrating the components described in Section 3.4 for the example shown in Figure 3.

```
1  public void routeAPacketTo(IPAddress ip, int ttl, List<Host> usedHosts) throws Exception {
2      usedHosts.add(this);
3  1   if (ttl == 0) {//Problem in routing
4          throw new Exception("Routing problem...");
5      }
6  1   if (!hasIP(ip)) { //Packet not arrived
7          Host nextHost = null;
8          List<Host> directlyAccessibleHosts = getDirectlyAccessibleHosts();
9      1+1 for (Host directlyAccessibleHost: directlyAccessibleHosts) {
10         1+2 if (...) // If the packet is for a neighbour, we send it to him
11                 nextHost = directlyAccessibleHost;
12         }
13     1+1 if (nextHost != null) {
14             nextHost.routeAPacketTo(ip, ttl − 1, usedHosts);
15     1   } else {//We have to look in the routing table
16             List<Host> directlyAccessible = getDirectlyAccessibleHosts();
17             IPAddress nextIP = this.getRoutingTable().getNextHop(ip);
18             boolean nextHostFound = false;
19         1+2 for (Host aDirectlyAccessible: directlyAccessible) {
20             1+3 if (...) { //Search the nextHop host object
21                     aDirectlyAccessible.routeAPacketTo(ip, ttl − 1, usedHosts);
22                     nextHostFound = true;
23                 }
24             }
25         1+2 if (!nextHostFound) { //Routing problem
26                 throw new Exception("Routing problem...");
27             }
28         }
29     }
30 }
```

Fig. A1: Example of the count of Cognitive Complexity (CC), from the example in Figure 3. It is demonstrated that the inherent and nesting components are the sum of two components when a nesting component is present.



```
1  // Cognitive complexity 20
2  public void routeAPacketTo(IPAddress ip, int ttl, List<Host> usedHosts) throws Exception {
3      usedHosts.add(this);
4      // [λ=0, ι=1, ν=0, μ=0, CCR=1, NMCC=1]
5      if (ttl == 0) {//Problem in routing
6  (1)     throw new Exception("Routing problem, TTL is null : packet to "
7                              + ip + " deleted on host " + this.getName());
8      }
9      // [λ=0, ι=8, ν=11, μ=6, CCR=19, NMCC=19]
10     if (!hasIP(ip)) { //Packet not arrived
11         Host nextHost = null;
12 (8)   List<Host> directlyAccessibleHosts = getDirectlyAccessibleHosts();
13         // [λ=1, ι=2, ν=1, μ=2, CCR=5, NMCC=3]
14         for (Host directlyAccessibleHost: directlyAccessibleHosts) {
15             // [λ=2, ι=1, ν=0, μ=1, CCR=3, NMCC=1]
16 (2)        if (directlyAccessibleHost.hasIP(ip)) // If the packet is for a neighbour,
17                                                  // We send it to him
18 (1)          nextHost = directlyAccessibleHost;
19         }
20         // [λ=1, ι=5, ν=4, μ=4, CCR=13, NMCC=9]
21         if (nextHost != null) {
22 (5)        nextHost.routeAPacketTo(ip, ttl − 1, usedHosts);
23         // [λ=1, ι=4, ν=4, μ=3, CCR=12, NMCC=9]
24         } else {//We have to look in the routing table
25             List<Host> directlyAccessible = getDirectlyAccessibleHosts();
26             IPAddress nextIP = this.getRoutingTable().getNextHop(ip);
27 (4)        boolean nextHostFound = false;
28             // [λ=2, ι=2, ν=1, μ=2, CCR=7, NMCC=3]
29             for (Host aDirectlyAccessible: directlyAccessible) {
30                 // [λ=3, ι=1, ν=0, μ=1, CCR=4, NMCC=1]
31 (2)            if (aDirectlyAccessible.hasIP(nextIP)) { //Search the nextHop host object
32 (1)                aDirectlyAccessible.routeAPacketTo(ip, ttl − 1, usedHosts);
33                     nextHostFound = true;
34                 }
35             }
36             // [λ=2, ι=1, ν=0, μ=1, CCR=3, NMCC=1]
37             if (!nextHostFound) { //Routing problem
38 (1)            throw new Exception("Routing problem, there is no route corresponding
39                                     to the packet or the destination host is on the internet");
40             }
41         }
42     }
43 }
```

Fig. A2: Count of accumulated inherent component, ι, from the example in Figure 3.

```
1  // Cognitive complexity 20
2  public void routeAPacketTo(IPAddress ip, int ttl, List<Host> usedHosts) throws Exception {
3      usedHosts.add(this);
4      // [λ=0, ι=1, ν=0, μ=0, CCR=1, NMCC=1]
5      if (ttl == 0) {//Problem in routing
6          throw new Exception("Routing problem, TTL is null : packet to "
7                              + ip + " deleted on host " + this.getName());
8      }
9      // [λ=0, ι=8, ν=11, μ=6, CCR=19, NMCC=19]
10     if (!hasIP(ip)) { //Packet not arrived
11         Host nextHost = null;
12         List<Host> directlyAccessibleHosts = getDirectlyAccessibleHosts();
13     1 // [λ=1, ι=2, ν=1, μ=2, CCR=5, NMCC=3]
14     ←for (Host directlyAccessibleHost: directlyAccessibleHosts) {
15     1 1 // [λ=2, ι=1, ν=0, μ=1, CCR=3, NMCC=1]
16     ←←if (directlyAccessibleHost.hasIP(ip)) // If the packet is for a neighbour,
17                                             // We send it to him
18             nextHost = directlyAccessibleHost;
19         }
20     1 // [λ=1, ι=5, ν=4, μ=4, CCR=13, NMCC=9]
21     ←if (nextHost != null) {
22         nextHost.routeAPacketTo(ip, ttl − 1, usedHosts);
23     0 // [λ=1, ι=4, ν=4, μ=3, CCR=12, NMCC=9]
24     ←} else {//We have to look in the routing table
25         List<Host> directlyAccessible = getDirectlyAccessibleHosts();
26         IPAddress nextIP = this.getRoutingTable().getNextHop(ip);
27         boolean nextHostFound = false;
28     1 1 // [λ=2, ι=2, ν=1, μ=2, CCR=7, NMCC=3]
29     ←←for (Host aDirectlyAccessible  directlyAccessible) {
30     1 1 1 // [λ=3, ι=1, ν=0, μ=1, CCR=4, NMCC=1]
31     ←←←if (aDirectlyAccessible.hasIP(nextIP)) { //Search the nextHop host object
32             aDirectlyAccessible.routeAPacketTo(ip, ttl − 1, usedHosts);
33             nextHostFound = true;
34         }
35         }
36     1 1 // [λ=2, ι=1, ν=0, μ=1, CCR=3, NMCC=1]
37     ←←if (!nextHostFound) { //Routing problem
38         throw new Exception("Routing problem, there is no route corresponding
39                             to the packet or the destination host is on the internet");
40         }
41     }
42     }
43 }
```

Fig. A3: Count of accumulated nested component, ν, from the example in Figure 3. Note that the `else` sentence does not increment the nested component, just the inherent one.



Fig. A4: Count of the contribution of accumulated nesting component, μ, from the example in Figure 3.

References

[1] McCabe, T.J.: A complexity measure. IEEE Transactions on Software Engineering **SE-2**(4), 308–320 (1976) https://doi.org/10.1109/TSE.1976.233837

[2] Saborido, R., Ferrer, J., Chicano, F., Alba, E.: Automatizing software cognitive complexity reduction. IEEE Access **10**, 11642–11656 (2022) https://doi.org/10.1109/ACCESS.2022.3144743

[3] Saborido, R., Ferrer, J., Chicano, F.: Automatizing Software Cognitive Complexity Reduction through Integer Linear Programming (2024). https://arxiv.org/abs/2402.05559

[4] Basili, V.R., Selby, R.W., Phillips, T.: Metric analysis and data validation across fortran projects. IEEE Transactions on Software Engineering **SE-9**(6), 652–663 (1983) https://doi.org/10.1109/TSE.1983.235430

[5] Weyuker, E.J.: Evaluating software complexity measures. IEEE Transactions on Software Engineering **14**(9), 1357–1365 (1988) https://doi.org/10.1109/32.6178

[6] Kaner, C., Bond, W.P.: Software Engineering Metrics: What Do They Measure and How Do We Know? (2004). https://api.semanticscholar.org/CorpusID:6293556

[7] Misra, S., Koyuncu, M., Crasso, M., Mateos, C., Zunino, A.: A suite of cognitive complexity metrics. In: Murgante, B., Gervasi, O., Misra, S., Nedjah, N., Rocha, A.M.A.C., Taniar, D., Apduhan, B.O. (eds.) Computational Science and Its Applications – ICCSA 2012, pp. 234–247. Springer, Berlin, Heidelberg (2012)

[8] Campbell, G.A.: Cognitive Complexity: a new way of measuring understandability (2021). https://www.sonarsource.com/docs/CognitiveComplexity.pdf

[9] Recio Abad, J.C., Saborido, R., Chicano, F.: Naming Methods with Large Language Models after Refactoring Operations. In: Inteligencia Artificial Para Ingeniería del Software. Sistedes, A Coruña, Spain (2024). https://hdl.handle.net/11705/JISBD/2024/29

[10] Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, Third Edition, 3rd edn. The MIT Press, Cambridge, Masachusetts (2009)

[11] Saborido, R., Ruiz, A.B., González-Gallardo, S., Luque, M., Borrego, A.: Performance assessment of population-based multiobjective optimization algorithms using composite indicators. IEEE Transactions on Evolutionary Computation **1**(1), 1–1 (2025) https://doi.org/10.1109/TEVC.2025.3544412

[12] Mavrotas, G.: Effective implementation of the ε-constraint method in multi-objective mathematical programming problems. Applied Mathematics and Computation **213**(2), 455–465 (2009) https://doi.org/10.1016/j.amc.2009.03.037

[13] Miguel Ángel Domínguez Ríos: Anytime methods for multi-objective combinatorial optimization problems with applications to software engineering. Phd thesis, University of Málaga, Málaga, Spain (January 2022). Supervisors: José Francisco Chicano García and Enrique Alba Torres. https://hdl.handle.net/10630/24333

[14] Guddat, J., Vasquesz, F.G., Tammer, K., Wendler, K.: Multiobjective and Stochastic Optimization Based on Parametric Optimization. Mathematical Research. De Gruyter, Berlin, Germany (1985). https://books.google.es/books?id=G_Y_EQAAQBAJ

[15] Bradstreet, L.: The hypervolume indicator for multi-objective optimisation: calculation and use. PhD thesis, The University of Western Australia (2011). https://research-repository.uwa.edu.au/en/publications/the-hypervolume-indicator-for-multi-objective-optimisation-calcul