Recommending relevant classes for bug reports using multi-objective search

Conference Paper · August 2016

DOI: 10.1145/2970276.2970344

CITATIONS
27

READS
258

4 authors, including:

Mohamed Wiem Mkaouer
Rochester Institute of Technology
138 PUBLICATIONS **1,332** CITATIONS

SEE PROFILE

Ali Ouni
Osaka University
136 PUBLICATIONS **2,555** CITATIONS

SEE PROFILE

Recommending Relevant Classes for Bug Reports Using Multi-Objective Search

ABSTRACT

Developers may follow a tedious process to find the cause of a bug based on code reviews and reproducing the abnormal behavior. In this paper, we propose an automated approach to finding and ranking potential classes with the respect to the probability of containing a bug based on a bug report description. Our approach finds a good balance between minimizing the number of recommended classes and maximizing the relevance of the proposed solution using a multi-objective optimization algorithm. The relevance of the recommended classes (solution) is estimated based on the use of the history of changes and bug-fixing, and the lexical similarity between the bug report description and the API documentation. We evaluated our system on 6 open source Java projects, using the version of the project before fixing the bug of many bug reports. The experimental results show that the search-based approach significantly outperforms three state-of-the-art methods in recommending relevant files for bug reports. In particular, our multi-objective approach is able to successfully locate the true buggy methods within the top 10 recommendations for over 87% of the bug reports.

CCS Concepts

• Software and its engineering → Software notations and tools → Software maintenance tools

Keywords

Search-based software engineering; bug reports; multi-objective optimization; software maintenance.

1. INTRODUCTION

A software bug is a coding error that may cause abnormal behaviors and incorrect results when executing the system [1]. After identifying an unexpected behavior of the software project, a user or developer will report it in a document, called a bug report [2]. Thus, a bug report should provide useful information to identify and fix the bug. The number of these bug reports can be large. For example, MOZILLA had received more than 420,000 bug reports [3]. These reports are important for managers and developers during their daily development and maintenance activities [4].

A developer always uses a bug report to reproduce the abnormal behavior to find the origin of the bug. However, the poor quality of bug reports can make this process tedious and time-consuming due to missing information. To find the cause of a bug, developers are not only using their domain knowledge to investigate the bug report, but interact with peer developers to collect additional information. An efficient automated approach for locating and ranking important code fragments for a specific bug report may lead to improving the productivity of developers by reducing the time to find the cause of a bug [4]. Most of the existing studies are mainly based on lexical matching scores between the statements of bug reports and the name of code elements in software systems [5]. However, there is a significant difference between the natural language used in bug reports and the programming language which limits the efficiency of existing approaches.

In this work, we start from the following observations. First, API documentation of the classes and methods can be more useful than the name of code elements or comments to estimate the similarity between code fragments and bug reports. Second, classes associated to previously fixed bug reports may be relevant also to the current report if these previously bug reports are similar to a current bug report. Third, a code fragment that was fixed recently is more likely to still contain bugs than another class that was last fixed long time ago. Fourth, a class that has been frequently fixed, tend to be fault-prone and may cause more than one abnormal behavior in the future. Finally, the recommendation of a large number of classes to inspect may make the process of finding the cause of a bug time-consuming.

To consider the above observations, we propose a comprehensive approach for bugs localization based on bug reports description. To this end, we propose, for the first time, to use a multi-objective optimization algorithm [6] to find a balance between maximizing lexical and history-based similarity, and minimizing the number of recommended classes. The problem is formulated as a search for the best combination and sequence of classes from all the classes of the system that optimize as much as possible the above two conflicting objectives.

We have executed an extensive empirical evaluation of 6 large open-source software projects with more than 22,000 bug reports in total based on an existing benchmark [7]. The results on the before-fix versions show that our system outperforms, on average, three state-of-the-art approaches not based on search techniques [7] [8] [9]. In particular, our search-based approach is able to successfully locate the true buggy methods within the top 10 recommendations for over 87% of the bug reports.

The primary contributions of this paper can be summarized as follows:

- To the best of our knowledge and based on recent surveys [10], the paper proposes the first search-based software engineering approach to address the problem of finding relevant code fragments for bug reports. The approach combines the use of lexical and history based similarity measures to locate and rank relevant code fragments for bug reports while minimizing the number of recommended classes.
- The paper reports the results of an empirical study with an implementation of our multi-objective approach. The obtained results provide evidence to support the claim that our proposal is more efficient, on average, than existing

techniques [7] [8] [9] based on a benchmark of 6 open source systems. We also compared the results of our multi-objective approach with a mono-objective formulation to make sure that our objectives are conflicting.

The remainder of this paper is as follows: Section 2 describes the search algorithm; an evaluation of the algorithm is explained and its results are discussed in Section 3; the different threats that affect our experimentations are described in Section 4; Section 5 is dedicated to related work. Section 6 describes the threats to validity related to our experiments. Finally, concluding remarks and future work are provided in Section 7.

2. RELATED WORK

In this section, we survey different studies related to the areas of bug localization and search-based software engineering.

2.1 Bug Localization

The problem of bug localization can be considered as searching the source of a bug given its description. To address this problem, the majority of existing studies is based on the use Information-Retrieval (IR) techniques through the detection of textual and semantic similarities between a newly given report and source code entities [5]. Several IR techniques have been investigated, namely the Latent Semantic Indexing (LSI) [11], Latent Dirichlet Allocation (LDA) [12] and the Vector Space Model (VSM) [13]. In addition, hybrid models extracted from these IRs techniques to tackle the problem of bug localization were proposed[7].

We summarize, in the following, the different tools and approaches proposed in the literature based on the above IR techniques. BugScout [8] is a topic-based approach using LDA to analyze the bug related information (description, comments, external links, etc.) to detect the source of a bug and duplicated bug reports. The main limitation of BugScout is the dependency of the results on the keywords entered by the user. DebugAdvisor's [14] is a bug investigation system that takes as input a bug report in terms of text queries then uses them to mine existing fixed bug repository and generate a graph of possible reports. However, DebugAdvisor accuracy depends on the accuracy of the report's description and its accuracy when describing the bug and its related code entities.

BugLocator [9] combines several similarity scores from previous bug reports for bug localization. It generates a VSM model to extract suspect source files for a given bug report. Then, BugLocator mines previously fixed bug reports along with related files involved to rank suspect code fragments. The main issue raised in this work is the proneness of the weight density to the noise in the large files. To overcome this limitation, [15] added segmentation and stack-trace analysis to improve the performance of the BugLocator approach. The limitation of this extension is that execution traces are not necessarily available in bug repositories.

BLUiR [16] has been proposed also to compare a bug report to the structure of source files. It decomposes reports into summaries and then uses the structural retrieval to calculate similarities between these tokenized elements and source code ones to rank source code files. Saha et al. [17] extended BLUiR to consider similar reports information, similarly to BugLocator as an additional similarity score. DHbPd [18] incorporated code change information for bug localization. The main idea is to consider recently changed source code elements as potential candidates for hosting a bug.

Ye et al. [7] has modeled the similarity between bug reports and source code through several characteristics that are captured through the use of 6 similarity features that describe the project's domain knowledge. The combination of these measures is fed to a ranking heuristic called learning-to-rank. The ranking model returns the top candidate source files to investigate for a given bug report. The main originality of their work is the use of projects API description and auto-generated documentation as one of the features to utilize to reduce the lexical gap between the human description and the source code.

In [19] Ye et al. extended their previous work by extending their ranking features utilized by learning-to-rank from 6 to 19. Besides the existing surface lexical similarity, API-based lexical similarity, collaborative filtering, code element's naming similarity, fixed bug's frequency, they included other source code characteristics that can be extracted from the projects such as summaries, naming conventions, inter-class dependencies etc. Although taking these features into account has given better results in terms of better files' ranking, such information may not be available in all projects and sometimes it may be outdated and that may deteriorate the localization's accuracy.

We propose, in this paper, a more comprehensive approach to address the problem of bug's localization from different perspectives as detailed in the next sections.

2.2 Search-Based Software Engineering

Search-Based Software Engineering (SBSE) uses a computational search approach to solve optimization problems in software engineering [20]. Once a software engineering task is framed as a search problem, by defining it in terms of solution representation, fitness function, and solution change operators, there is a multitude of search algorithms that can be applied to solve that problem.

Many search-based software testing techniques have been proposed for test cases generation [21], mutation testing [22], regression testing [23] and testability transformation. However, the problem of bug's localization was not addressed before using SBSE. The closest problem addressed using SBSE techniques is the bug's prioritization problem [24]. A mono- objective genetic algorithm was proposed to find the best sequence of bugs' resolution that maximizes the relevance and importance of the bugs to fix while minimizing the cost. The main limitation of this work is the use of a mono-objective technique that aggregates two conflicting objectives.

In the next section, we describe our formulation of bug localization as a multi-objective problem.

3. MULTI-OBJECTIVE FORMULATION

We first present an overview of our multi-objective approach to identify and prioritize relevant code fragments (e.g. classes) for bug reports, and then we describe the details of our multi-objective formulation.

3.1 Approach Overview

Our approach aims at exploring a large search space to find relevant classes, to inspect by developers, given a description of a bug report. The search space is determined not only by the number of possible class combinations to recommend, but also by the order in which they are proposed to the programmer. In fact, bug reports may require the inspection of more than one class to identify and fix bugs.

A heuristic-based optimization method is proposed based on two main conflicting objectives. The first objective is the correctness function that includes two sub-functions: 1.a) maximizing the Lexical similarity between recommended classes and the description of the bug report (including the API and name of code elements similarity), and 1.b) maximizing the history-based function score that includes the number of a recommended classes that have been fixed in the past, recent changes introduced by the developers to these classes and similarities with previous bug reports. The second objective is to minimize the number of classes to recommend.

It is clear that these two objectives are conflicting since maximizing the relevance of recommended classes may leads to a lower precision and thus increases the number of recommended classes. Thus, we consider, in this paper, the task of localizing bugs as a multi-objective optimization problem using the non-dominated sorting genetic algorithm (NSGA-II) [6]. The proposed algorithm will explore a large search space of a combinatorial number of combinations of classes to recommend.

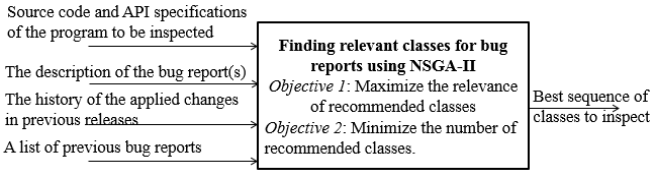

Figure 1. Approach Overview

The general structure of our approach is sketched in Figure 1. It takes as input the source code of the program to be inspected, the API specifications of the classes of the system, the description of the bug report and a list of previous bug reports and the history of the applied changes in previous releases. Our approach generates as output a near-optimal sequence of ranked classes that maximizes the relevance to the bug report and minimizes the number of recommended classes. In the following, we describe an overview of NSGA-II, the solution representation, a formal formulation of the two objectives to optimize and the change operators.

3.2 NSGA-II

In this paper, we adapted one of the widely used multi-objective algorithms called NSGA-II [6]. NSGA-II is a powerful search method stimulated by natural selection that is inspired by the theory of Darwin. Hence, the basic idea of NSGA-II is to make a population of candidate solutions evolve toward the near-optimal solution in order to solve a multi-objective optimization problem. NSGA-II is designed to find a set of optimal solutions, called non-dominated solutions, also Pareto set. A non-dominated solution is the one which provides a suitable compromise between all objectives without degrading any of them. As described in Figure 2, the first step in NSGA-II is to create randomly a population P_0 of individuals encoded using a specific representation (line 1). Then, a child population Q_0 is generated from the population of parents P_0 using genetic operators such as crossover and mutation (line 2). Both populations are merged into an initial population R_0 of size N (line 5). As a consequence, NSGA-II starts by generating an initial population based on a specific representation that will be discussed later, using the exhaustive list of classes from the system to inspect given as input as mentioned in the previous section. Thus, this population stands of a set solutions represented as sequences of classes to inspect, which are randomly selected and ordered, for a specific bug report description taken as input.

The whole population that contains N individuals (solutions) is sorted using the dominance principle into several fronts (line 6). The dominance level becomes the basis of a selection of individual solutions for the next generation. Fronts are added successively until the parent population P_{t+1} is filled with N solutions (line 8). When NSGA-II has to cut off a front F_i and select a subset of individual solutions with the same dominance level, it relies on the crowding distance to make the selection (line 9). This front F_i to be split, is sorted in descending order (line 13), and the first (N-$|P_{t+1}|$) elements of F_i are chosen (line 14). Then a new population Q_{t+1} is created using selection, crossover and mutation (line 15). This process will be repeated until reaching the last iteration according to stop criteria (line 4).

1. Create an initial population P_0
2. Generate an offspring population Q_0
3. t=0;
4. **while** *stopping criteria not reached* **do**
5. $R_t = P_t \cup Q_t$;
6. F = fast-non-dominated-sort (R_t);
7. $P_{t+1} = \emptyset$ and i=1;
8. **while** $| P_{t+1}| + |F_i| \leq N$ **do**
9. Apply crowding-distance-assignment(F_i);
10. $P_{t+1} = P_{t+1} \cup F_i$;
11. $i = i$+1;
12. **end**
13. $Sort(F_i, \prec n)$;
14. $P_{t+1} = P_{t+1} \cup F_i[1 : (N\text{-}| P_{t+1} |)]$;
15. Q_{t+1} = create-new-pop(P_{t+1});
16. $t = t$+1;

end

Figure 2. NSGA-II Pseudo-Algorithm

The following three subsections describe more precisely our adaption of NSGA-II to the model change detection problem.

3.3 Solution Approach

3.3.1 Solution representation

To represent a candidate solution (individual), we used a vector representation. Each dimension of the vector represents a class to recommend for a specific bug report. Thus, a solution is defined as a sequence of classes to recommend for inspection by the developer to locate the bug.

When created, the order of recommended classes corresponds to their positions in the vector. The classes to recommend are dependent since a bug can be located in different classes. In addition, the goal is to recommend a minimum set of classes while maximizing the correctness objective.

StackRenderer	CompositeRenderer	TrimUtil

Figure 3. Simplified Example of a Solution Representation.

Bug ID: 378535
Summary: "**Close All**" and "**Close Others**" **menu** options available when right **clicking** on tab in **PartStack** when no **part** is **closeable**.

Description: If I create a **PartStack** that contains multiple **parts** but none of the **parts** are **closeable**, when I right **click** on any of the tabs I get **menu** options for "**Close All**" and "**Close Others**". Selection of either of the **menu** options doesn't **cause** any tabs to be **closed** since none of the tabs can be **closed**.

Reported: 2012-05-04 14:03

Figure 4. An Eclipse Bug Report Example[1] (ID 378535)

Figure 3 describes a simplified solution generated to find possible relevant classes for the bug report of Figure 4 that shows an example of a bug report from the Eclipse project (ID 378535). This bug report describes a defect about incorrect menu options for parts that are not closeable. The solution consists of a sequence of three classes to inspect extracted from the Eclipse project.

3.3.2 Fitness functions

Correctness objective: This objective is defined as the average of two functions: lexical-based similarity (LS) and history-based similarity (HS). Thus, we formally define this function as:

$$f_1 = \frac{LS + HS}{2} \qquad (1)$$

The lexical-based similarity (LS) consists of an average of two functions. The first function is based on a cosine similarity [25] between the description of a bug report and the source code. We used the whole content of a source code file (the code and comments). The vocabulary was extracted from the names of variables, classes, methods, parameters, types, etc. We used the *Camel Case Splitter* to perform the Tokenization for preprocessing the identifiers [26].

During the tokenization process, we used a standard information retrieval *stop words* to eliminate irrelevant information such as punctuation, numbers, etc. In addition, the words are reduced to their stem based on a *Porter stemmer*. This operation reduces the deviation between related words such as *designing* and *designer* to the same stem *design*. Then, the cosine similarity measure is used to compare between the description of a bug report and the source code.

Equation 2 calculates the cosine similarity between two actors. Each actor is represented as an n dimensional vector, where each dimension corresponds to a vocabulary term. The cosine of the angle between two vectors is considered as an indicator of similarity. Using cosine similarity, the conceptual similarity between two actors: c_1 and c_2 is determined as follows:

$$Sim(C_1, C_2) = Cos(C1, C2) = \frac{\vec{C}_1 \bullet \vec{C}_2}{\left\|\vec{C}_1\right\| \times \left\|\vec{C}_2\right\|}$$

$$= \frac{\sum_{i=1}^{n} (w_{i,1} \times w_{i,2})}{\sqrt{\sum_{i=1}^{n} (w_{i,1})^2 + \sum_{i=1}^{n} (w_{i,2})^2}} \qquad (2)$$

where $\vec{c_1} = (w_{1,1}; \ldots w_{n,1})$ is the term vector corresponding to actor c_1 and $\vec{c_2} = (w_{1,2}; \ldots w_{n,2})$ is the term vector corresponding to c_2. The weights $w_{i,j}$ is computed using information retrieval based techniques such as the Term Frequency - Inverse Term Frequency (TF-IDF) method[27]. The first lexical similarity function is then defined as the sum of the of the cosine similarity scores between a description of a bug report and the source code of each the suggested classes divided by the total number of recommended classes.

As described in Figure 4 and Figure 5, the description of the bug report example includes several similar words with one of the recommended classes to inspect, the class *StackRenderer*. Thus, the cosine similarity function applied between the source code of that class and the description of the bug report will detect such similarities. However, the only use of this similarity function may not be enough.

In fact, the text of a bug report is, in general, expressed in a natural language however the large part of the content of a source code is described in a programming language. Thus, the similarity score between a bug report description and a source code will be higher in case of an extensive use of comments in the code or if the bug report clearly uses the names of code elements. To address this challenge, we propose to use an additional lexical similarity function.

The second lexical function is based on the use of cosine similarity between the bug report description and the API specification of each method of a recommended buggy class. Thus, it is defined as the sum of the maximum of the cosine similarity scores between a description of a bug report and each of the methods composing the suggested class divided by the total number of recommended classes.

As described in Figure 5, the class *StackRenderer* includes a variable *uiElement* having as a type *MUIElement*. Figure 6 shows the API specification of the *MUIElement* interface that includes different terms such as *parts* and *menus* that also exists in the bug report description of Figure 4. Thus, the lexical similarity between the API specification and the description of a bug report may also help to better identify relevant buggy classes.

[1] https://bugs.eclipse.org/bugs/show bug.cgi?id=378535

```
997         private boolean closePart(Widget widget, boolean check) {
998                 MUIElement uiElement = (MUIElement) widget
999                                 .getData(AbstractPartRenderer.OWNING_ME);
1000                MPart part = (MPart) ((uiElement instanceof MPart) ? uiElement
1001                                : ((MPlaceholder) uiElement).getRef());
1002                if (!check && !isClosable(part)) {
1003                        return false;
1004                }
1005
1006                IEclipseContext partContext = part.getContext();
1007                IEclipseContext parentContext = getContextForParent(part);
1008                // a part may not have a context if it hasn't been rendered
1009                IEclipseContext context = partContext == null ? parentContext
1010                                : partContext;
1011                // Allow closes to be 'canceled'
1012                EPartService partService = (EPartService) context
1013                                .get(EPartService.class.getName());
1014                if (partService.savePart(part, true)) {
1015                        partService.hidePart(part);
1016                        return true;
1017                }
1018                // the user has canceled out of the save operation, so don't close the
1019                // part
1020                return false;
1021        }
1022
```

Figure 5: A code fragment extracted from the class *StackRenderer*.

API Specification of **MUIElement**:

Description: A representation of the model object 'UI Label'. This is a mix in that will be used for UI Elements that are capable of showing label information in the GUI (e.g. **Parts**, **Menus** / Toolbars, Perspectives, ...). The following features are supported: Label, Icon URI, Tooltip ...

Figure 6: API Specification of the interface *MUIElement*.

The second component of the correctness objective is the history-based similarity. This measure is an average of three functions. The first function counts the number of times that a class was fixed to eliminate bugs based on the history of bug reports. In fact, a class that was fixed several times has a high probability of being a buggy class and includes new bugs. Formally, this function, normalized between [0,1] is defined as:

$$H_1 = \frac{\sum_{i=1}^{Size(S)} NbFixedBug\ s(report, c_i)}{Size(S) \cdot Max(NbFixedBug\ s(report, c_i))} \quad (3)$$

The second function checks if a recommended class was recently changed or fixed. In fact, a class that was modified recently has a higher probability of containing a bug. Thus, the function compares between the date of the bug report and the last date where the recommended class was modified. If a suggested class was modified on the same day of the bug report then the value of this function is 1. We define this normalized function, normalized in the range of [0, 1] as following:

$$H_2 = \frac{\sum_{i=1}^{Size(S)} \frac{1}{report.date - last(report, c_i) + 1}}{Size(S)} \quad (4)$$

The third function evaluates the consistency between the recommended classes based on previous bug reports. The classes that are recommended together for similar previous bug reports have a high probability to include a bug evolving most of them. To this end, this function calculates first the cardinality, *Cbr*, of the largest intersection set of classes between the solution *S* and the sets of classes recommended for each of previous bug reports. Then, this measure is normalized as follows:

$$H_3 = \frac{Cbr}{Size(S)} \quad (5)$$

3.3.3 *Change operators*

In a search algorithm, the variation operators play the key role of moving within the search space with the aim of driving the search towards better solutions. We used the principle of the Roulette wheel [28] to select individuals for mutation and crossover. The probability to select an individual for crossover and mutation is directly proportional to its relative fitness in the population. In each iteration, we select half of the population in iteration *i*. These selected individuals will give birth to another half of the population of new individuals in iteration *i+1* using a crossover operator. Therefore, two parent individuals are selected, and a few dimensions (recommended classes) picked on each one. The one point crossover operator allows creating two offspring P'_1 and P'_2 from the two selected parents P_1 and P_2. It is defined as follows: a random position, *k*, is selected. The first *k* classes of P_1 become the first *k* elements of P'_1. Similarly, the first *k* operations of P_2 become the first *k* operations of P'_2. Our crossover operator could create a child that contains redundant recommended classes. In order to resolve this problem, for each obtained child, we verify whether there are redundant classes or not. In case of redundancy, we replace the redundant classes by randomly chosen ones from the system without causing another redundancy.

The mutation operator can be applied to pairs of dimensions of the vector selected randomly. Given a selected solution, the mutation operator first randomly selects one or many pairs of dimensions of the vector. Then, for each selected pair, the dimensions, which correspond to classes, are deleted or replaced by new classes. We used the same repair operator, described previously, to eliminate redundancy.

4. EVALUATION

In order to evaluate our approach for recommending relevant classes to inspect for bug reports, we conducted a set of experiments based on different versions of 6 open source systems. Each experiment is repeated 30 times, and the obtained results are subsequently statistically analyzed with the aim to compare our NSGA-II proposal with a variety of existing approaches not based on heuristic search [7, 8, 9] and a mono-objective formulation. In this section, we present our research questions and then describe and discuss the obtained results.

4.1 Research Questions

In our study, we assess the performance of our approach by finding out whether it could identify the most relevant classes to inspect for bug reports. Our study aims at addressing the following research questions outlined below. We also explain how our experiments are designed to address these questions. The main question to answer is to what extent the proposed approach can propose meaningful bug localization solutions based on the description of a bug report. To this end, we defined the following research questions:

- **RQ1.** (Efficiency) To what extent can the proposed approach identify relevant classes to localize bugs based on bug reports description?

- **RQ2.** (Comparison to search techniques) How does the proposed multi-objective approach based on NSGA-II perform compared to random search and a mono-objective approach?
- **RQ3.** (Comparison to state-of-the-art) How does our approach perform compared to existing bugs localization techniques not based on heuristic search?

To answer RQ1, we validate the proposed multi-objective technique on six medium to large-size open-source systems, as detailed in the next section, to evaluate the correctness of the recommended classes to inspect for a bug report. To this end, we used the following evaluation metrics:

- **Precision@k** denotes the number of correct recommended files in the top *k* of recommended files by the solution divided by the minimum number of files to inspect, in the ranked recommendations list, to localize the bug.
- **Recall@k** denotes the number of correct recommended files in the top *k* of recommended files by the solution divided by the total number of expected files to be recommended that contain the bug.
- **Accuracy@k** measures the percentage of bug reports for which at least one correct recommendation was provided in the top *k* ranked classes.

To answer RQ2, we compared, using the above metrics, the performance of NSGA-II with random search and a mono-objective genetic algorithm aggregating all the objectives into one objective with equal weight. If Random Search outperforms a guided search method thus, we can conclude that our problem formulation is not adequate. It is important also to determine if our objectives are conflicting and outperforms a mono-objective technique. The comparison between a multi-objective technique with a mono-objective one is not straightforward. The first one returns a set of non-dominated solutions while the second one returns a single optimal solution. To this end, for we choose the nearest solution to the Knee point [29] (i.e., the vector composed of the best objective values among the population members) as a candidate solution to be compared with the single solution returned by the mono-objective algorithm. To answer RQ3, we compared our multi-objective approach to different existing techniques not based on heuristic search: 1. *BugScout* [8] identifies relevant classes based on the use of Latent Dirichlet Allocation measure [12]; 2. *BugLocator* [9] ranks classes using both textual and structural similarity.3. *Learning-to-rank* (LR) [7] technique ranks classes using a machine learning technique to learn from the history of previous bug reports. In addition, we compared our work with two additional baselines. The first one is based on the only use of the lexical measure (LS) to rank classes and the second one is based on the only use of the history measure (HS). These two baselines may justify or not the need of considering complementary information from both the lexical and history similarities in our multi-objective formulation.

In the next section, we describe the different projects and the 10-fold cross-validation used in our experiments.

4.2 Software Projects and Experimental Setting

As described in Table 1, we used a benchmark datasets for six open-source systems [7].

- **Eclipse UI** is the user interface of the Eclipse development framework.
- **Tomcat** implements several Java EE specifications.
- **AspectJ** is an aspect-oriented programming (AOP) extension created for the Java programming language.
- **Birt** provides reporting and business intelligence capabilities.
- **SWT** is a graphical widget toolkit.
- **JDT** provides a set of tool plug-ins for Eclipse.

Table 1 shows the different statistics of the analyzed systems including the time range of the bug reports, the number of bug reports, the size, the number of APIs, and the number of fixed classes per bug report.

The total number of collected bug reports and associated classes is more than 22,000 bug reports for the six open source systems. All these projects are using BugZilla tracking system and GIT as a version control system. To avoid using a fixed code revision, we associated a before-fixed version of the system to each bug report. Therefore, for each bug report, the version of the software package just before the fix was committed was used in our validation.

Based on the collected data, we created two sets: one for the training data and the other for the test data. The bug reports for each system were sorted chronologically based on the time dimension. The sorted bug reports are then split into 10 folds with equal sizes, where $fold_1$ contains the most recent bug reports and the last fold $fold_{10}$ contains the oldest ones. In addition, the oldest fold is split into 70% training (history of bug reports) and 30% validation. The approach is trained on $fold_{i+1}$ and tested on $fold_i$, for all i from 1 to 9. The best recommended solution is then compared with expected solution of classes that contain the bug.

4.3 Parameters Tuning and statistical tests

Since metaheuristic algorithms are stochastic optimizers, they can provide different results for the same problem instance from one run to another. For this reason, our experimental study is performed based on 30 independent simulation runs for each problem instance and the obtained results are statistically analyzed by using the Friedman test with a 95% confidence level (α = 5%). The Friedman test is a non-parametric statistical test useful for multiple pairwise comparisons. The latter verifies the null hypothesis H_0 that the obtained results of the different algorithms are samples from continuous distributions with equal medians, as against the alternative that they are not, H_1. The p-value of the Friedman test corresponds to the probability of rejecting the null hypothesis H_0 while it is true (type I error). A p-value that is less than or equal to α (≤ 0.05) means that we accept H_1 and we reject H_0. However, a p-value that is strictly greater than α (> 0.05) means the opposite. In this way, we could decide whether the superior performance of NSGA-II to one of each of the other algorithms (or the opposite) is statistically significant or just a random result.

Table 1. Studied Projects

Project	# Bug reports	Time	# API	# files in the project (average per version)	# fixed files/classes per bug report (median)
Eclipse UI	6495	10/2001-01/2014	1314	3454	2
Birt	4178	06/2005-12/2013	957	6841	1
JDT	6274	10/2001-01/2014	1329	8184	2
AspectJ	593	03/2002-01/2014	54	4439	2
Tomcat	1056	07/2002-01/2014	389	1552	1
SWT	4151	02/2002-01/2014	161	2056	3

The Friedman test allows verifying whether the results are statistically different or not. However, it does not give any idea about the difference in magnitude. To this end, we used the Vargha and Delaney's A statistics which is a non-parametric effect size measure. In our context, given the different performance metrics (such as Precision and Recall), the A statistics measures the probability that running an algorithm B1 (NSGA-II) yields better performance than running another algorithm B2 (such as GA). If the two algorithms are equivalent, then A = 0.5.

An often-omitted aspect in metaheuristic search is the tuning of algorithm parameters. In fact, parameter setting influences significantly the performance of a search algorithm on a particular problem. For this reason, for each search algorithm and each system, we performed a set of experiments using several population sizes: 10, 20, 30, 40 and 50. The stopping criterion was set to 100,000 fitness evaluations for all search algorithms in order to ensure fairness of comparison. We used a high number of evaluations as a stopping criterion since our approach requires involves multiple objectives. Each algorithm was executed 30 times with each configuration and then the comparison between the configurations was performed based on different metrics described previously using the Friedman test. The other parameters values were fixed by trial and error and are as follows: (1) crossover probability = 0.4; mutation probability = 0.2 where the probability of gene modification is 0.1.

4.4 Results

4.4.1 Results for RQ1

The results of Table 2 and Figures 7 to 9 confirm the efficiency of our multi-objective approach to identify the most relevant classes for bug reports that include the bugs on the 6 open source systems. Table 2 shows the average *precision@k* results of our multi-objective technique on the different six systems, with *k* ranging from 5 to 20. For example, most of the recommended classes to inspect in the top 5 (*k=5*) are relevant with a precision of 89%. The lowest precision is around 70% for *k=20* which is still could be considered acceptable since most of the bug reports do not have many classes to inspect. In terms of recall, Table 2 confirms that the majority of the expected classes to recommend are located in the top 20 (*k=20*) with an average recall score of 94%. An average of more than 72% of classes recommended in the *top5* cover the expected buggy classes.

The average *accuracy@k* results on the different six systems are described in Table 2 showing that an average of 68%, 86%, 94% and 97% are achieved for *k* = 5, 10, 15 and 20 respectively. These results confirm that if we recommend only 10 classes to programmers, we can make correct recommendations for 86% of the thousands of collected bug reports for every system.

Table 2. Median Precision@k, Recall@k and Accuracy@k on 30 independent runs. The results were statistically significant on 51 independent runs using the Friedman test with a 95% confidence level (α < 5%).

k	Precision@k							
	NSGA-II	Bug Scout	Bug Locator	LR	LS	HS	RS	GA
5	**89**	76	78	81	69	71	34	71
10	**82**	71	74	76	61	64	29	61
15	**74**	63	69	72	57	58	33	55
20	**68**	48	51	58	48	51	24	53

k	Recall@k							
	NSGA-II	Bug Scout	Bug Locator	LR	LS	HS	RS	GA
5	**72**	59	62	64	54	56	27	54
10	**81**	64	67	72	60	62	31	62
15	**87**	69	72	79	65	67	26	69
20	**94**	74	80	83	70	72	24	76

k	Accuracy@k							
	NSGA-II	Bug Scout	Bug Locator	LR	LS	HS	RS	GA
5	**68**	41	44	49	37	34	29	38
10	**86**	62	69	71	56	59	24	59
15	**94**	74	78	82	68	72	31	79
20	**97**	79	82	86	74	77	33	77

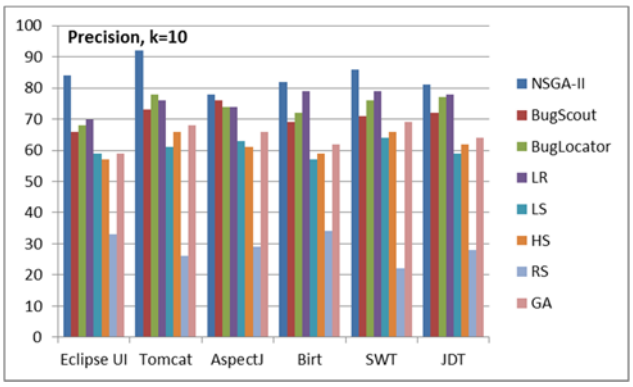

Figure 7: Average Precision@k of NSGA-II, BugScout, BugLocator, LR, LS, HS, RS and GA on the different systems for 30 independent runs.

Figures 7 to 9 summarize the results of the precision@10, recall@10 and accuracy@10 for every of the studied systems. The obtained results clearly show that most of the buggy classes were recommended correctly by our multi-objective approach in the top 10 with a minimum precision of 78% for AspectJ, a

minimum recall of 79% for Eclipse and a minimum accuracy of 82% for Eclipse as well. Thus, we noticed that our technique does not have a bias towards the evaluated system. As described in Figures 7-9, in all systems, we had almost similar average scores of precision, recall and accuracy. All these results based on the different measures were statistically significant on 30 independent runs using the Friedman test with a 95% confidence level ($\alpha < 5\%$).

To answer RQ1, the obtained results on the six open source systems using the different evaluation metrics of precision, recall and accuracy clearly validate the hypotheses that our multi-objective approach can recommend efficiently relevant buggy classes to inspect for each bug report.

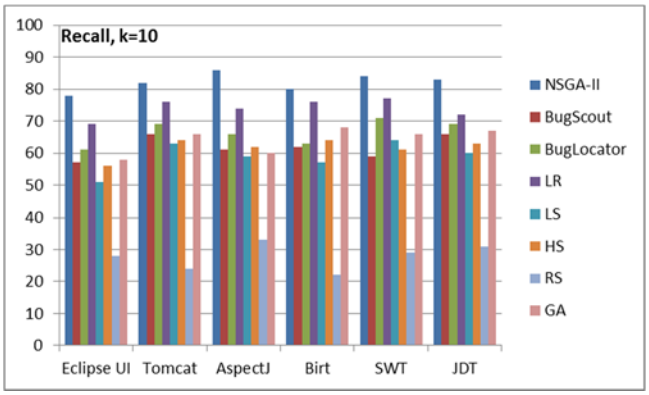

Figure 8: Average Recall@k of NSGA-II, BugScout, BugLocator, LR, LS, HS, RS and GA on the different systems for 30 independent runs.

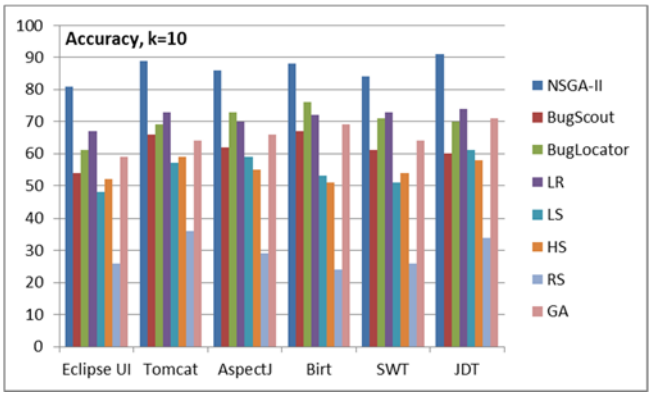

Figure 9: Average Accuracy@k of NSGA-II, BugScout, BugLocator, LR, LS, HS, RS and GA on the different systems for 30 independent runs.

4.4.2 *Results for RQ2*

Concerning RQ2, Table 2 and Figures 6-11 confirm that NSGA-II is better than random search and the three mono-objective formulations (LS, HS and GA) based on the three metrics of precision, recall and accuracy on all the 6 systems. Three mono-objective formulations were implemented:

1. with an equal aggregation of both objectives (GA);
2. a mono-objective algorithm with the only objective of lexical similarity (LS); and
3. a mono-objective algorithm with the only objective of history similarity (HS).

The average accuracy, precision and recall values of random search (RS) on the six systems are lower than 35% as described in Table 2. This can be explained by the huge search space to explore to identify the best order of classes to inspect for bugs localization. The performance of the three mono-objective algorithms was much better than random search but lower than our multi-objective formulation. The aggregation of both objectives into one objective generates better results on all the six systems than the two other algorithms considering each objective separately. Thus, an interesting observation is the clear complementary between the history-based similarity function and the lexical-based measure. In fact, we found that the buggy classes that are not detected by one of the two algorithms were identified by the other algorithm. The average precision, recall and accuracy of each of the two algorithms (LH and HS) was between 67% and 72% but the aggregation of both objectives into one in our multi-objective formulation improve a lot the obtained results. In addition, since NSGA- II outperforms the mono-objective GA then it is clear that the two objectives of correctness/relevance and the number of recommended classes are conflicting.

All these results were statistically significant on 30 independent runs using the Friedman test with a 95% confidence level ($\alpha < 5\%$). We have also found the following results of the Vargha Delaney A_{12} statistic : a) On large and medium scale systems (Birt, JDT, Eclipse UI and AspectJ) NSGA-II is better than all the other algorithms based on all the performance metrics with an A effect size higher than 0.93; b) On small scale systems (Tomcat, SWT), NSGA-II is better than all the other algorithms with a an A effect size higher than 0.96.

We conclude that there is empirical evidence that our multi-objective formulation surpasses the performance of random search and mono-objective approaches thus our formulation is adequate (this answers RQ2).

4.4.3 *Results for RQ3*

Since it is not sufficient to compare our approach with only search-based algorithms, we compared the performance of NSGA-II with three different bugs localization techniques not based on heuristic search [7] [8] [9]. Table 2 and Figures 7 to 9 present the *precision@k, recall@k* and *accuracy@k* results for the 3 implemented methods, with *k* ranging from 5 to 20. NSGA-II achieves better results, on average, than the other three methods on all six projects. For example, our approach achieved, on average, *Precision@k* of 90%, 84%, 73% and 69% are achieved for k= 5, 10, 15 and 20 respectively as described in Table 2. In comparison, BugLocator achieved an average Precision@k of 68%. BugScout and LR achieved an average Precision@k of 66% and 72%, respectively. Similar observations are also valid for the recall@k and accuracy@k.

Based on the results of Figures 7-9 Birt and Tomcat are two projects where *LR* performs close to the NSGA-II approach. For many bug reports in Birt, most of the buggy classes are those that have been frequently fixed in previous bug reports which explain the relatively high performance obtained by *LR* and NSGA-II. Since the bug fixing information is exploited by both the NSGA-II approach and LR, it is expected that they obtain the best performance results.

To answer RQ3, the obtained results on the six open source system using the different evaluation metrics of precision, recall and accuracy clearly validate the hypotheses that our multi-

objective approach outperforms several bugs localization techniques not based on heuristic search.

5. DISCUSSIONS

Impact of Data Size. To evaluate the impact of increasing the size of the data used (history of previous bug reports and changes), we executed a scenario on the JDT project in which we increased the size of the dataset incrementally fold by fold until we include all the 9 folds in the dataset. It is clear from Figure 12 that for all the three metrics of *Precision@k, Recall@k* and *Accuracy@k* that increasing the size of the previous bug reports do not improve all the three metrics. This can be explained by the fact that recent bug reports and history of changes are the most important part of the data. The obtained results confirm also that our multi-objective approach did not require a large set of data to generate good results in terms of finding possible buggy classes for bug reports.

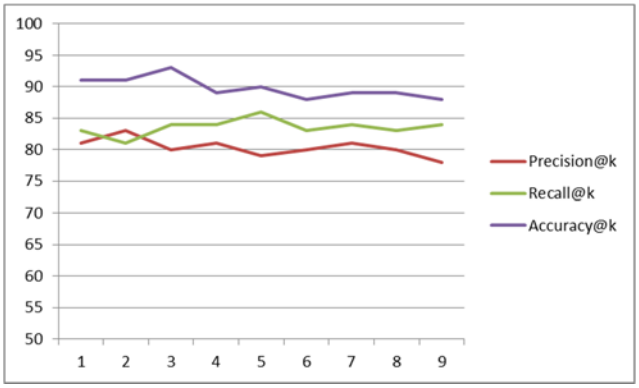

Figure 10: Impact of the data training size (folds) on the three metrics based on the JDT project.

Execution time. We executed our multi-objective algorithm on a desktop computer with CPU Intel(R) Core(TM) i7 3.2 GHz and 20G RAM. Figure 13 presents the execution time performance of our approach. The average execution time on the different systems was around 18 minutes. The highest execution time was observed on the JDT system with 23 minutes and the lowest one was around 11 minutes for SWT. We believe that the execution is reasonable since bugs localization is not a real-time problem. We also found that the execution time is related to the number of files to parse and the history of bug reports.

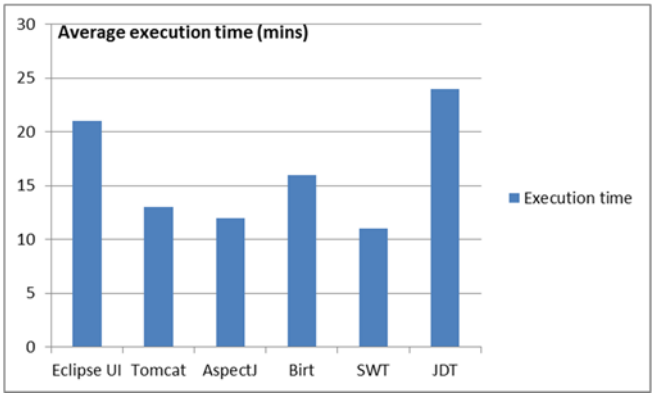

Figure 11: Average execution time (in minutes) of NSGA-II, on the different systems for 30 independent runs on the different systems

6. THREATS TO VALIDITY

We explore, in this section, the factors that can bias our empirical study. These factors can be classified in three categories: construct internal and external validity. Construct validity concerns the relation between the theory and the observation. Internal validity concerns possible bias with the results obtained by our proposal. Finally, external validity is related to the generalization of observed results outside the sample instances used in the experiment.

In our experiments, construct validity threats are related to the absence of similar work that uses search-based techniques for bug's localization. For that reason, we compared our proposal with different mono-objective formulations to check the need for a multi-objective approach. A construct threat can also be related to the corpus of manually localized bugs for every bug report. A limitation related to our experiments is the difficulty to set the thresholds for some of the parameters of Bug Locator. In fact, we used the default thresholds used by the authors that can have an impact on the quality of the generated results.

We take into consideration the internal threats to validity in the use of stochastic algorithms since our experimental study is performed based on 30 independent simulation runs for each problem instance, and the obtained results are statistically analyzed by using the statistical test with a 95% confidence level (α = 5%). The parameter tuning of the different optimization algorithms used in our experiments creates another internal threat that we need to evaluate in our future work by additional experiments to evaluate the impact of the parameters on the quality of the results.

External validity refers to the generalization of our findings. In this study, we performed our experiments on six different widely-used open-source systems belonging to different domains and with different sizes. However, we cannot assert that our results can be generalized to other applications, other programming languages, and to other practitioners.

7. CONCLUSION AND FUTURE WORK

We propose, in this paper, an automated approach to localize and rank potential relevant classes for bug reports. Our approach finds a trade-off between minimizing the number of recommended classes and maximizing the correctness of the proposed solution using a multi-objective optimization algorithm. The correctness of the recommended classes is estimated based on the use of the history of changes and bug-fixing, and the lexical similarity between the bug report description and the API documentation. We have executed extensive empirical evaluations on 6 large open-source software projects with more than 22,000 bug reports in total based on an existing benchmark. The results on the before-fix versions show that our system outperforms, on average, three state-of-the-art approaches not based on search techniques [7] [8] [9]. In particular, our search-based approach is able to successfully locate the true buggy methods within the top 10 recommendations for over 87% of the bug reports.

As part of our future work, we plan to evaluate our multi-objective approach on further projects in other different programming languages. In addition, we will extend our work to

address the problem of the software bugs management and prioritization using multi-objective search techniques.

8. REFERENCES

[1] Bruegge, B., and Dutoit, A.H.: 'Object-Oriented Software Engineering Using UML, Patterns and Java-(Required)' (Prentice Hall, 2004. 2004)

[2] Bettenburg, N., Just, S., Schröter, A., Weiss, C., Premraj, R., and Zimmermann, T.: 'What makes a good bug report?', in Editor (Ed.)

[3] Bettenburg, N., Premraj, R., Zimmermann, T., and Kim, S.: 'Duplicate bug reports considered harmful… really?', in Editor (Ed.) (IEEE, 2008, edn.), pp. 337-345

[4] Fischer, M., Pinzger, M., and Gall, H.: 'Analyzing and relating bug report data for feature tracking' (IEEE, 2003. 2003)

[5] Sun, C., Lo, D., Wang, X., Jiang, J., and Khoo, S.-C.: 'A discriminative model approach for accurate duplicate bug report retrieval', in Editor (Ed.) (ACM, 2010, edn.), pp. 45-54

[6] Deb, K., Pratap, A., Agarwal, S., and Meyarivan, T.: 'A fast and elitist multiobjective genetic algorithm: NSGA-II', IEEE Transactions on Evolutionary Computation, 2002, 6, (2), pp. 182-197

[7] Ye, X., Bunescu, R., and Liu, C.: 'Learning to rank relevant files for bug reports using domain knowledge', in Editor (Ed.) (ACM, 2014, edn.), pp. 689-699

[8] Nguyen, A.T., Nguyen, T.T., Al-Kofahi, J., Nguyen, H.V., and Nguyen, T.N.: 'A topic-based approach for narrowing the search space of buggy files from a bug report', in Editor (Ed.) (IEEE, 2011, edn.), pp. 263-272

[9] Zhou, J., Zhang, H., and Lo, D.: 'Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports', in Editor (Ed.) (IEEE, 2012, edn.), pp. 14-24

[10] Harman, M., Mansouri, S.A., and Zhang, Y.: 'Search-based software engineering: Trends, techniques and applications', ACM Computing Surveys (CSUR), 2012, 45, (1), pp. 11

[11] Dumais, S.T.: 'Latent semantic analysis', Annual review of information science and technology, 2004, 38, (1), pp. 188-230

[12] Blei, D.M., Ng, A.Y., and Jordan, M.I.: 'Latent dirichlet allocation', the Journal of machine Learning research, 2003, 3, pp. 993-1022

[13] Salton, G., Wong, A., and Yang, C.-S.: 'A vector space model for automatic indexing', Communications of the ACM, 1975, 18, (11), pp. 613-620

[14] Ashok, B., Joy, J., Liang, H., Rajamani, S.K., Srinivasa, G., and Vangala, V.: 'DebugAdvisor: a recommender system for debugging', in Editor (Ed.) (ACM, 2009, edn.), pp. 373-382

[15] Wong, C.-P., Xiong, Y., Zhang, H., Hao, D., Zhang, L., and Mei, H.: 'Boosting bug-report-oriented fault localization with segmentation and stack-trace analysis', in Editor (Ed.) (IEEE, 2014, edn.), pp. 181-190

[16] Saha, R.K., Lease, M., Khurshid, S., and Perry, D.E.: 'Improving bug localization using structured information retrieval', in Editor (Ed.) (IEEE, 2013, edn.), pp. 345-355

[17] Saha, R.K., Lawall, J., Khurshid, S., and Perry, D.E.: 'On the effectiveness of information retrieval based bug localization for c programs', in Editor (Ed.) (IEEE, 2014, edn.), pp. 161-170

[18] Rao, S., and Kak, A.: 'Retrieval from software libraries for bug localization: a comparative study of generic and composite text models', in Editor (Ed.) (ACM, 2011, edn.), pp. 43-52

[19] Ye, X., Bunescu, R., and Liu, C.: 'Mapping Bug Reports to Relevant Files: A Ranking Model, a Fine-grained Benchmark, and Feature Evaluation', IEEE Transactions on Software Engineering, 2016, 42, (2), pp. 379-402

[20] Harman, M., and Jones, B.F.: 'Search-based software engineering', Information and software Technology, 2001, 43, (14), pp. 833-839

[21] Núñez, A., Merayo, M.G., Hierons, R.M., and Núñez, M.: 'Using genetic algorithms to generate test sequences for complex timed systems', Soft Computing, 2013, 17, (2), pp. 301-315

[22] Henard, C., Papadakis, M., and Le Traon, Y.: 'Mutation-based generation of software product line test configurations': 'Search-Based Software Engineering' (Springer, 2014), pp. 92-106

[23] Shelburg, J., Kessentini, M., and Tauritz, D.R.: 'Regression testing for model transformations: A multi-objective approach': 'Search Based Software Engineering' (Springer, 2013), pp. 209-223

[24] Dreyton, D., Araújo, A.A., Dantas, A., Freitas, Á., and Souza, J.: 'Search-Based Bug Report Prioritization for Kate Editor Bugs Repository': 'Search-Based Software Engineering' (Springer, 2015), pp. 295-300

[25] Tan, P.-N., Steinbach, M., and Kumar, V.: 'Introduction to data mining' (Pearson Addison Wesley Boston, 2006. 2006)

[26] Enslen, E., Hill, E., Pollock, L., and Vijay-Shanker, K.: 'Mining source code to automatically split identifiers for software analysis', in Editor (Ed.) (IEEE, 2009, edn.), pp. 71-80

[27] Salton, G., and McGill, M.J.: 'Introduction to modern information retrieval', 1986

[28] Goldberg, D.E., and Deb, K.: 'A comparative analysis of selection schemes used in genetic algorithms', Foundations of genetic algorithms, 1991, 1, pp. 69-93

[29] Branke, J., Deb, K., Dierolf, H., and Osswald, M.: 'Finding knees in multi-objective optimization', in Editor (Ed.) (Springer, 2004, edn.), pp. 722-731

[30] Arcuri, A., and Fraser, G.: 'Parameter tuning or default values? An empirical investigation in search-based software engineering', Empirical Software Engineering, 2013, 18, (3), pp. 594-623

[31] Bialas, W., Karwan, M., and Shaw, J.: 'A parametric complementary pivot approach for two-level linear programming', State University of New York at Buffalo, 1980, 57