Noname manuscript No.
(will be inserted by the editor)

Detecting Dependencies Between Bug Reports to Improve Bugs Triage

Rafi Almhana · Marouane Kessentini

the date of receipt and acceptance should be inserted later

Abstract Software development teams need to deal with several open reports of critical bugs to be addressed urgently and simultaneously. The management of these bugs is a complex problem due to the limited resources and the deadlines-pressure. Most of the existing studies treated bug reports in isolation when assigning them to developers. Thus, developers may spend considerable cognitive efforts moving between completely unrelated bug reports thus not sharing any common files to be inspected. In this paper, we propose an automated bugs triage approach based on the dependencies between the open bug reports. Our approach starts by localizing the files to be inspected for each of the pending bug reports. We defined the dependency between two bug reports as the number of common files to be inspected to localize the bugs. Then, we adopted multi-objective search to rank the bug reports for programmers based on both their priorities and the dependency between them. We evaluated our approach on a set of open source programs and compared it to the traditional approach of considering bug reports in isolation based mainly on their priority. The results show a significant time reduction of over 30% in localizing the bugs simultaneously comparing to the traditional bugs prioritization technique based on only priorities.

Keywords Bugs Management · Bug Triage · Bug Localization · Bug Prioritization · Search-Based Software Engineering · Software Quality Assurance

Rafi Almhana
University of Michigan
E-mail: ralmhana@umich.edu

Marouane Kessentini
University of Michigan
E-mail: marouane@umich.edu

1 Introduction

Software maintenance involves, typically, localizing and fixing a large number of defects that arise during development and evolution of systems [1]. Localizing these software defects is expensive and time-consuming process which typically requires highly skilled and knowledgeable developers of the system. The localization process includes a manual search through the source code of the project in order to localize a single bug at a time [2]. 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 including bug localization [4]. Due to the large number of reported bugs in successful projects, it is critical to efficiently manage them to improve developers productivity and quickly localize and fix these bugs [5].

Each bug report has a set of attributes such as the bug's summary, description, reported date, reporter's information, bug's severity, and bug's priority. According to the Bugzilla's definition about severity and priority, severity indicates how severe the problem is, and it ranges from blocker ('application unusable') to trivial ('minor cosmetic issue'). Priority options ranges from P1 (the highest) to P5 (the lowest) whereas severity could be any of the following options: blocker, critical, enhancement, major, minor, normal, and trivial.

In general, bug triage process consists of two phases. The first phase involves mainly the project managers and project owners, the goal of this step is to understand the business needs and/or the urgency of some of the bugs, the outcome would be to assign severity values on the bugs. The second phase involves the project managers and the developers (scrum planning meeting) in which managers and developers review the backlog of bugs, understand the technical tasks, improve the bug's description, and eventually prioritize them and assign them to developers. This bug triage process plays an important role in software maintenance since the timely localization and correction of bugs are critical for the reputation of the organization and customers' satisfaction.

Once a bug report is assigned to a team, one of the developers uses it 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. An efficient automated approach for locating and ranking important code fragments for a specific bug report may lead to improve the productivity of developers by reducing the time to find the cause of a bug [4].

Although several techniques have been proposed to localize bugs [6, 7] and predict the severity of bugs [8, 9, 1], the existing studies related to the management of bugs report are mainly based on the priority scores to rank and assign bug reports without looking to the possible dependencies between them [10–12]. Thus, developers may get assigned bug reports related to completely different files to be inspected which may increase the cognitive effort of the developers navigating between these independent bug reports. For instance, a developer may spend time understanding files A and B for Bug report B1 then he needs to check again these same files for bug report after working on three other independent bugs reports. We start, in this paper, from the hypothesis that a better way to manage bugs reports is to group together those with a similar level of priorities and also sharing a common number of files to be inspected and fixed. In fact, several empirical studies show that the majority of bugs may not appear in isolation and they are related to each other [10–12]. These dependent bug reports have several common files to inspect to localize the bugs.

To the best of our knowledge, we propose one of the first studies that consider the dependencies between bug reports in order to rank and group them while still considering their priorities. The proposed approach is mainly to validate the hypothesis that ranking and grouping bug reports based on the dependencies between them (classes to be inspected) besides the bugs priority can improve the productivity of developers and help them to localize bugs faster and more efficiently than considering them in isolation based only on the priority scores of the bug.

Our approach aims to find a trade-off between ranking the bug reports based on (1) their dependency and (2) their priority. The dependencies are extracted based on the list of files to be inspected from the bug report description using our previous bugs localization work [7] using a combination of lexical and history based measures. We selected that technique due to its high accuracy in localizing relevant files with over 80% in precision and recall. After extracting the list of files to inspect for each bug report, we adopted a multi-objective search, based on NSGA-II [13], to find a trade-off between bugs priority and dependencies to rank the bug reports when assigned to developers. Thus, the manager or developer can select the best schedule of the bugs based on his/her preferences from the list of non-dominated ranking solutions generated by NSGA-II. For instance, a solution with high priority score and low dependency can be selected when the goal is to mainly focus on localizing the most severe bugs independently from the required effort.

To the best of our knowledge, this paper represents the first study to formulate the bug prioritization problem as multi-objective search and consider the dependency between bugs in terms of classes related to the resolution. Thus, our goal is to evaluate the formulation of the problem as a multi-objective search to deal with the conflicting objectives. Thus, we compared with a mono-objective formulation to confirm that the objectives are actually conflicting and the out-performance of the proposed search algorithm. Based on our previous Search Based Software Engineering (SBSE) work and existing studies, most search algorithms will perform similarly when the formulation is the same (fitness functions, solution representation, etc.) thus we selected NSGA-II algorithm since it is widely used in similar software engineering problems such as the next release problem [14].

An experiment has been conducted to compare our approach with the only use of bugs priority to rank bug reports [15–20]. We conducted a pre-study and post-study survey to evaluate the performance of our tool with participants based on 6 open source projects. Our multi-objective approach uses multiple conflicting objectives. In our case, we have two fitness functions F1 & F2 to represent our objectives. On the other hand, a mono-objective approach uses only one objective / fitness function aggregating all the objectives. Thus, we have also compared our approach to mono-objective search. The results show significant time reduction of over 30% in correctly localizing the bugs simultaneously comparing to the traditional bugs prioritization technique based on priority.

The remainder of this paper is as follows: Section 2 is dedicated to describing the problem and our motivation to find a solution for it. Section 3 describes the proposed approach to localize bugs and then prioritize them. The evaluation of our approach and its results on several research questions with the answers and the discussions on those research questions are explained in Section 4. Section 5 describes the threats to validity related to our experiments. Section 6 is dedicated to related studies. Finally, concluding remarks and future work is provided in Section 7.

2 Related Work and Motivating Example

2.1 Related Work

A survey on bug prioritization was proposed in [8]. The authors collected 84 papers about bug prioritization or related topics from 2000 to 2015, they eliminated 32 papers after 2 steps review process. The majority of those papers used information retrieval technique such as Naive Bayes, Support Vector Machine (SVM) and Neural Networks for bugs prioritization. The survey focused mainly on predicting bugs priority and to estimate the severity of the bugs.

Table 1 summarizes the main studies related to bugs management and prioritization.

Kanwal et al. [23] proposed a classification based approach to develop a tool which uses the Naive Bayes and Support Vector Machine (SVM) classifiers. This tool mines the bug data from a bug repository so that it builds a piece of knowledge about the software to be inspected and its bugs repository and eventually rank or classify bugs.

The authors in [18] proposed an approach to predict the priority of bug report using different machine learning algorithms like Naive Bayes, Decision Trees, and Random Forest.

Xuan et al. [17] proposed a new way to prioritize bugs based on 3 different stages from mining the social interactions between developers.

Search-Based Software Engineering (SBSE) uses a computational search approach to solve optimization problems in software engineering [30]. 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 [31], mutation testing [32], regression testing

Table 1 Overview of bug prioritization related work

Study	**Input**	**Output**	**Technique**	**Published**
Yu et al. [15]	Bug reports	Predict bug priority	Neural Networks	2010
Jaweria Kanwal [20]	Bug reports	Recommend bug priority	SVM	2010
Lamkanfi et al. [19]	Bug reports	Predict the severity of bug	Naive Bayes	2010
Chaturvedi and Singh [9]	Bug reports	Determine bug severity	Naive Bayes	2012
Abdelmoez et al. [21]	Bug reports	Predict bug fix-time	Naive Bayes	2012
Dommati et al. [22]	Bug reports	Classify bug reports	Naive Bayes	2012
Kanwal and Maqbool [23]	Bug reports	Prioritize bug reports	SVM	2012
Sharma et al. [24]	Bug reports	Predict bug priority	SVM	2012
Thung et al. [25]	Bug reports	Predict bug priority	SVM	2012
Tian et al. [26]	Bug reports	Predict the severity of bug	Nearest Neighbors	2012
Xuan et al. [17]	Developer prioritization	Predict the severity of bug	NB, SVM	2012
Alenezi and Banitaan [18]	Bug reports	Predict bug priority	Decision Tree, Random Forests	2013
Zanetti et al. [27]	Bug reports	Classify bug reports	SVM	2013
Behl et al. [28]	Bug reports	Predict the severity of bug	TF-IDF	2014
Garcia and Shihab [29]	Bug reports	predicting blocking bugs	Decision trees	2014
Goyal et al. [16]	Bug reports	Predict bug priority	Bayes Net, Random Forest,	2015

[33] and testability transformation. However, the problem of bugs localization was not addressed before using SBSE. The closest problem addressed using SBSE techniques is the bugs prioritization problem [34]. 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. To overcome the limitation of aggregating two attributes that may experience conflicts, they extended their work [35] to better find the trade-off between bugs with low relevance and the bugs that may have high severity scores.

The problem of bug localization can be considered as searching the source for a bug given its description. To address this problem, the majority of existing studies is based on the use of Information-Retrieval (IR) techniques through the detection of textual and semantic similarities between a newly given report and source code entities [36]. Several IR techniques have been investigated, namely the Latent Semantic Indexing (LSI) [37], Latent Dirichlet Allocation (LDA) [38] and the Vector Space Model (VSM) [39]. Also, hybrid models extracted from these IRs techniques to tackle the problem of bug localization were proposed [40].

2.2 Motivating Example

The bug triage process involves intensive time and resources in order to manage and analyze all reported bugs on a daily basis. Typically, project managers need to understand the reported bug, tweak the bug description and check for duplication, then assign priority or severity of a bug and finally assign it to a developer.

As of May 2019, the Mozilla bug database contains over 172,000 bug reports for Firefox project; the Eclipse bug database reports over 210,000 bug reports for Eclipse project. On average, Mozilla received 212 and Eclipse 224 new bug reports on each week. Thus, clearly, the manual management of defects for large software projects is not practical to prioritize and rank a large load of reported bugs. Furthermore, it is important to efficiently assign these bugs to reduce potential delays in localizing and fixing them.

Most of the existing work on the bugs prioritizing mainly focus on the assigned priority or severity to a bug either manually or automatically using static/dynamic analysis and the history of changes/bugs [15–20]. They treated bug reports in isolation despite that recent empirical studies show that a large number of simultaneous bugs were located on the same files [10–12]. To the best of our knowledge, none of those techniques considered finding the dependencies among several bugs when ranking and grouping them to assign to developers. Recommending a list of bugs that share some common potential files to be inspected would be helpful to minimize the cognitive effort spent by a developer to jump from package to package or from file to file that are not related. Recent studies show that reducing such cognitive effort is a key to improve the productivity of developers working on multiple tasks [10–12].

Table 2 shows a list of 4 bug reports from the Eclipse Birt project that were reported on Bugzilla within two days. By looking at the bugs description and their resolution on Github, we found that all of them are related to the core component/module of the software and require inspecting almost the same files and/or directory to localize and fix them. Typically, developers prefer to work on defects that are dependent on each other so that they can focus on one set of files rather getting disrupted with multiple not related bugs. Our hypothesis that the bug triage process will significantly save time and resources

Table 2 List of 4 bugs in Eclipse Birt project

Bug ID	Bug Summary	Bug Reported	Inspected Files
Bug 456730	Missing default value in initializing scriptContext	2015-01-05	core/org.eclipse.birt.core/src/org/eclipse/birt/core/.. /ScriptContext.java
Bug 456725	Optimize the performance of ULocale.forLocale	2015-01-05	core/org.eclipse.birt.core/src/org/eclipse/birt/core/.. /LocaleUtil.java
Bug 456723	org.eclipse.birt.core.util.IOUtil doesn't check EOF	2015-01-05	core/org.eclipse.birt.core/src/org/eclipse/birt/core/.. /IOUtil.java
Bug 456847	BirtDateTime function in chart's onRender function causes render failure	2015-01-06	core/org.eclipse.birt.core/src/org/eclipse/birt/core/.. /CategoryWrapper.java

if we consider the dependencies between bugs as an additional criterion to the bugs severity.

3 Approach

3.1 Approach Overview

Our approach aims at exploring a large number of possible combination to find the best ranking of bug reports based on the dependency between them and their priority. The search space is determined not only by the number of possible dependencies between bug reports but also by the order in which they are proposed to the developer.

In fact, bug reports may require the inspection of more than one class to identify and fix bugs [10]. Our previous work for bugs localization [7] is executed to identify relevant files/classes to inspect for all the pending bug reports. The identified common files between the bug reports will represent the dependencies of all reported bugs we want to prioritize. Then, our bug prioritization component takes as input these dependencies along with the bug priority that has been assigned to each bug report. Our multi-objective search algorithm generates the best possible scheduling solutions to inspect the bugs to find a balance between priorities and dependencies of bugs. We represented the solution as a graph to guide developers to which bug needs to be resolved first, taking into consideration the two objectives of maximizing the number of files to inspect (maximize the intersection between consecutive bug reports in terms of files to inspect) and the bugs priority/severity that has been assigned manually by the project's stakeholders (e.g. developers or project managers). Since the bugs localization is performed at the files level based on our previous work [7]; thus, the clustering of our recommended solution is actually performed at the files level and not at the package or directory level.

The general structure of our approach is sketched in Figure 1. It takes two inputs, the bug priority assigned by the user and recommended classes

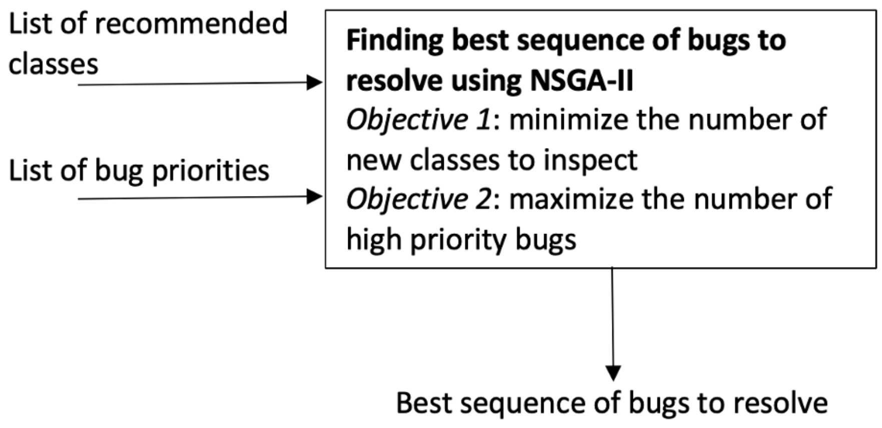
List of recommended classes
List of bug priorities
Finding best sequence of bugs to resolve using NSGA-II
Objective 1: minimize the number of new classes to inspect
Objective 2: maximize the number of high priority bugs
Best sequence of bugs to resolve

Fig. 1 Approach Overview

generated by the bugs localization tool (dependencies). The output is a set of non-dominated solutions of ranked bugs to inspect by the developer. Our heuristic-based optimization steps are formulated based on two main conflicting objectives. The first objective is to minimize the number of new classes to inspect between each pair of consecutively reported bugs. The second objective is to maximize the number of high priority bugs to be ranked first in the sequence of reported bugs. Thus, we consider, in this paper, the task of prioritizing bugs as a multi-objective optimization problem using the non-dominated sorting genetic algorithm (NSGAII) [13].

3.2 NSGA-II

In this paper, we adapted one of the widely used multi-objective algorithms called NSGA-II [13]. NSGA-II is a powerful global search method stimulated by natural selection that is inspired by the theory of Darwin. We selected this multi-objective search algorithm since it was used for similar problems in software engineering [7,14,41–46].

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 Algorithm 1, 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 bugs from the bug reports to resolve given as input. Thus, this population stands for a set of solutions represented as sequences of defects to resolve, which are randomly selected and ordered [7].

Algorithm 1 High level pseudo code for NSGA-II

```
1: Create an initial population $P_0$
2: Create 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 $\mid P_{t+1} \mid + \mid F_i \mid \leqslant N$ do
9:         Apply crowding-distance-assignment($F_i$)
10:        $P_{t+1} = P_{t+1} \cup F_i$
11:        $i = i + 1$
12:    end while
13:    $Sort(F_i, \prec n)$
14:    $P_{t+1} = P_{t+1} \cup F_i[N - \mid P_{t+1} \mid]$
15:    $Q_{t+1}$ = create-new-pop($P_{t+1}$)
16:    t = t+1
17: end while
```

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) [7]. The following subsections describe more precisely our adaption of NSGA-II to the bugs triage problem.

3.3 Solution Approach

3.3.1 Solution representation

Figure 2 shows a simplified representation of a solution (recommended schedule of bugs to resolve) generated by our web-based tool for bugs selected randomly from the bug repository (Bugzilla website) of Eclipse Birt project. This solution represents a possible sequence to resolve the reported bugs in Table 2 for Eclipse Birt project. The recommended classes of those defects share the

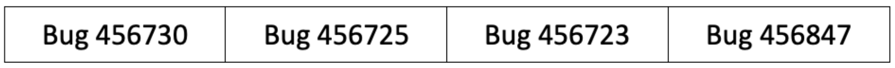

Fig. 2 A simplified example of solution representation

same package or directory (core/ org.eclipse.birt.core/ src/ org/ eclipse/ birt/ core) that needs to be inspected by programmer. Thus, we group those defects together and recommend this cluster of bugs to one developer to resolve as a sequence. This simplified representation may not be sufficient to show the dependencies between the bug reports thus we adopted a graph-based representation that can be visualized by the project's stakeholders (e.g. developers or project managers).

Figure 3 shows the sequence of bugs recommended for 10 pending bugs selected randomly from the bug repository (Bugzilla website) of Eclipse Birt project. The different bugs scheduling solutions that can be explored by the developers or managers are represented in Figure 4 balancing the two objectives of severity and dependencies. The purpose of Figure 3 is to show all possible routes presented in Table 3 of the paper, the nodes on the graph represent the bugs and the directed edges represent the order of the bugs in which it minimizes the trade-off between our objectives. An example of one possible route starts from first node (bug 28974) directed to next node (bug 29186) then bug 29665 $\rightarrow$ bug 28919 $\rightarrow$ bug 29684 $\rightarrow$ bug 29662 $\rightarrow$ bug 29693 $\rightarrow$ bug 20689 $\rightarrow$ bug 29691 $\rightarrow$ bug 29669 which is the last node in this route (recommended solution)

Figure 4 presents the Pareto Front line of the recommended solutions presented in Table 3 as 4 different recommendations, each one has a different route to examine the bugs with its value for objective 1 and objective 2 which are illustrated in this figure as 4 points correspond to the values of fitness function F1 & F2.

3.3.2 Fitness functions

There are two fitness functions used in our multi-objective search based algorithm. The first fitness function measure encourages keeping high priority bugs first in a sequence and low priority bugs last in a sequence. The first fitness function is to maintain low cognitive effort between each pair of consecutively reported bugs. Our goal is to minimize as much as possible the number of new classes to inspect when the developer moves from one bug to the next consecutive bug in the sequence. Equation 1 preserves the level of dependencies between each pair of consecutive bugs, a higher value represents high similarity in dependencies (recommended classes) among bug reports. The objective of the formula is to maximize the intersection (in number of inspected files) between two consecutive bug reports. $NumFiles_{i,i+1}$ represents the total number of distinct files to inspect for bug(i) and bug(i+1). Bug(i) represents the set of classes that are related to bug (i) and similarly Bug(i+1)

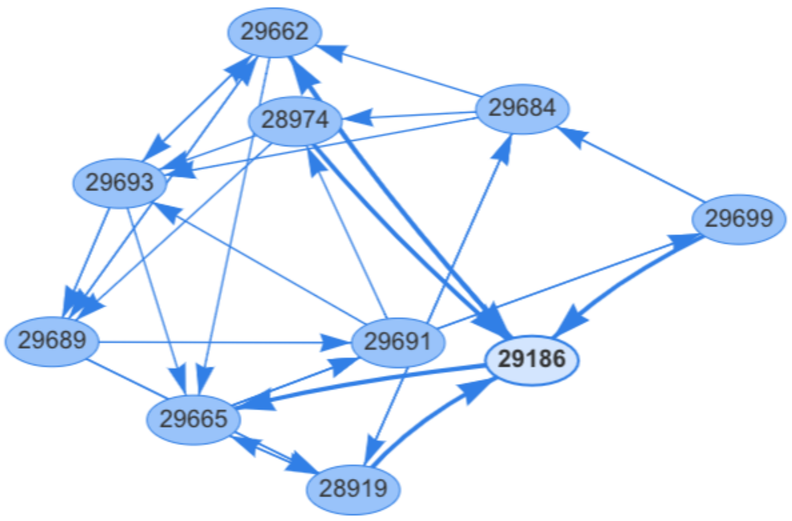

Fig. 3 Four different routes that shows the order of each recommended solutions generated by our web-based software for particular set of pending bugs in Eclipse Birt Project

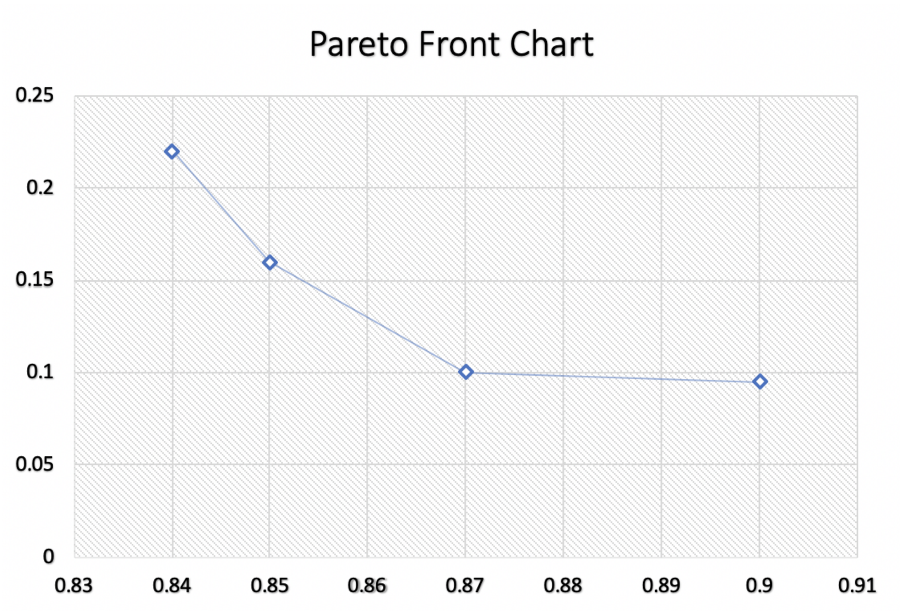

Fig. 4 The Pareto Front of recommended solutions generated by our web-based software for pending bugs in Eclipse Birt Project to balance both severity (X-Axis) and dependencies (Y-Axis).

represents the set of classes that are related to bug (i+1), where (n) represents the number of bugs.

$$f1 = \sum_{i=1}^{i=n} \frac{Bug_i \bigcap Bug_{i+1}}{NumFiles_{i,i+1}} \tag{1}$$

The objective of the second fitness function is to minimize the differences between the priority of bug reports and the order of recommendations to solve the reported bug reports. Equation 2 calculates the difference in priority for a bug between the bug report and the recommended solution. We build a vector of reported bugs and sort them based on the priority value reported in the bug report. Then, we compare the position of a reported bug B_i in recommended solution with the position of the same bug B_i in the original order of reported bugs which is based on priority value reported on bug reports. Equation 2 calculates the difference between the priority value in bug report and the priority value in the recommended solution which is the position of the bug in the solution vector. Equation 2 calculates the sum of differences in priority between bug report and the recommended solution for each of the bugs where (n) represents the number of bugs.

$$f2 = \sum_{i=1}^{i=n} |IndexOfBug_{i,solution} - IndexOfBug_{i,report}| \tag{2}$$

The above two objectives are conflicting since minimizing the number of new classes to inspect between each pair of consecutively reported bugs may lead to resolving some low priority bugs however the scheduling solution may improve the overall productivity.

Table 3 shows the Pareto Front sketched by our web-based tool for 10 bugs selected randomly from bug repository (Bugzilla website) of Eclipse Birt project. This is an example of Pareto Front results (the recommended solutions) generated by our web-based software for particular set of bugs in Eclipse Birt Project.

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 randomly 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, new two-parent individuals are selected for next iteration/generation.

Table 3 Pareto Front Results

No.	Solution	Objective 1	Objective 2
1	28974, 29186, 29665, 28919, 29684, 29662, 29693, 29689, 29691, 29699	0.84	0.22
2	28974, 29693, 29689, 29662, 29665, 29691, 29699, 29684, 28919, 29186	0.85	0.16
3	29699, 29684, 28974, 29689, 28919, 29665, 29691, 29693, 29662, 29186	0.87	0.10
4	29699, 29186, 29662, 29689, 28919, 29684, 29693, 29665, 29691, 28974	0.90	0.095

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 bugs of P_1 become the first k elements of P_1. Similarly, the first k bugs of P_2 become the first k elements of P_2. Our crossover operator could create a child that contains redundant recommended bugs. In order to resolve this problem, for each obtained child, we verify whether there are redundant bugs or not. In the case of redundancy, we do not apply crossover operation on this particular bug.

An example of crossover operation, consider there are (2) vectors of recommended solutions as follows:
Solution 1 $\rightarrow$ (bug A, bug B, bug C, bug D, bug E)
Solution 2 $\rightarrow$ (bug F, bug G, bug H, bug I, bug J)

After applying crossover operator on both solutions, the outcome will be as follows:
Solution 1 $\rightarrow$ (bug A, bug B, bug H, bug I, bug J)
Solution 2 $\rightarrow$ (bug F, bug G, bug C, bug D, bug E)

4 Evaluation

In order to evaluate our approach for prioritizing multiple defects for developers, we conducted a human validation to evaluate the benefits of our work. The experiments included a pre-study survey to gather some personal information and technical background of the participants then a post-study survey to gather developer's feedback about our tool with some insights about future improvements to the tool. The obtained results are subsequently statistically analyzed with the aim to compare our multi-objective approach with three other approaches. The first approach is a traditional bug priority based approach and the second one is based on the dependencies between bug reports without considering the score of the priority reported in the bug report. The third approach is based on a first come first served resolution based approach. In this section, we present our research questions followed by experimental settings and parameters. Then, we discuss our results for each of the research questions. The data related to our experiments can be found in the following link [47]

4.1 Research Questions

In our study, we wanted to assess the performance of our approach by finding out whether it could identify the most appropriate sequence of bugs to resolve by developers. In order to examine our web-based software prioritization tool, we explored two primary research questions outlined below. The goal of this experiment is to check whether our proposed approach can propose a meaningful sequence of defects in which developers can localize and fix related bugs quickly and therefore companies can save some efforts in terms of time, resource and cost to make their systems more responsive to most recent bug reports. To this end, we defined the following research questions:

– RQ1: (Effectiveness) To what extend can the proposed approach recommend an appropriate sequence of bugs to resolve by developer?
– RQ2: (Comparison to other techniques) How does our approach perform compared to typical bugs management techniques?

The goal of RQ1 is to measure the effectiveness of our approach by calculating three different metrics mentioned in this paper whereas the RQ2 aims to compare our approach with other approaches to measure the effectiveness compared to three other approaches (FCFS, 2 mono-objectives approaches).

To answer RQ1, we evaluate the effectiveness of the recommended order of bugs to resolve by programmers. The effectiveness is evaluated by measuring the following metrics:

– **Number of Bugs** denotes the number of bugs that one individual developer can resolve within a time frame. The goal for this measure is to maximize the number of bugs that developer can finalize in order to have better productivity.
– **Resolution Time** denotes the time spent by developer to understand, identify, and resolve a single particular bug. Our goal is to minimize this measure in order to save resource cost.
– **Disruption Cost** measures the cost of transition time that developer may spend between each pair of bugs. Our approach aims to minimize this cost by recommending most related sequence of bugs.

To answer RQ2, we compared, using the above metrics, the performance of our multi-objective approach with first come first serve approach. Furthermore, we implemented two mono-objective formulations. The first one is a mono-objective algorithm with the only objective of bug priority score and a second one is a mono-objective algorithm with the only objective of bug dependency. Disruption cost means the time in which a developer spends to make the transition between one bug to another unrelated bug. This transition involves the time to change the developer's focus to understand the information given to the developer in the new bug and the time to examine the files related to the new bugs. This disruption cost is important because it can show the cognitive effort required by developers to move from one bug to the other when they are not related. Equation 3 formulates the distribution cost where n is the number of bugs to resolve. To best of our knowledge, there is no similar prior work to compare with that uses currently similar objectives of our approach.

$$DisruptionCost = \sum_{i=1}^{i=n}(|EndTimeBug_i - StartTimeBug_{i+1}|) \quad (3)$$

One way to show if the two objectives are conflicting is to compare the performance of the multi-objective search with a mono-objective formulation (aggregation of all the objectives). The comparison between a multi-objective technique with a mono-objective one is not straightforward. The multi-objective technique returns a set of non-dominated solutions while the mono-objective technique one returns a single optimal solution. To this end, we choose the nearest solution to the Knee point [13] (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.

The knee point represents the maximum trade-off between the objectives thus it is reasonable to compare it with a mono-objective solution with equal weights of the different objectives aggregated in one fitness function. The fact that we are comparing a mono-objective formulation with equal weights to a knee point (representing the maximum possible trade-off) ensures a fair comparison. We used the knee point method as recommended by the current literature [48–50]

Both surveys (pre-study and post-study questionnaire) were conducted on twenty-nine developers who have a variety of skills and expertise. Table 4 shows a list of six open source systems that developers use in the experiment. The survey tells us whether our approach was successful to save cost and time in resolving bugs.

4.2 Software Projects and Experimental Setting

Multiple bugs are assigned randomly to multiple developers while making sure that 1) they all received a similar number of bugs to fix per system; and 2) they did not evaluate the same system with multiple tools. Developers are asked to resolve bugs that are already fixed in production without telling them they are already fixed in the next releases and they work on the versions before the bugs get fixed. The developers worked on multiple systems using the different approaches since we wanted to address the training threat if they just focus on one system. Developers are asked to evaluate different tools (not evaluated before) when they are asked to evaluate different systems.

Table 4 Studied Projects

Project	# Bugs	# Resolved Bugs	# Developers	Average Resolution Time	Time Frame
Eclipse UI	84,136	57,251	778	89 days	Oct-2001 to May-2019
Eclipse Birt	23,218	19,452	154	40 days	Jan-2005 to May-2019
Eclipse JDT	58,822	34,050	272	52 days	Oct-2001 to May-2019
Eclipse AspectJ	3021	2270	22	38 days	Sep-2002 to May-2019
Eclipse Jetty	3813	1184	14	43 days	Mar-2009 to May-2019
Eclipse SWT	24,049	19,559	184	61 days	Oct-2001 to May-2019

Table 5 List of developers participated in the experiment and their distribution among several projects along with the number of years of experience

Project	# of Developers	Avg. # of Experience
AspectJ	15	9.5 years
Birt	18	7 years
SWT	15	10 years
Jetty	14	10.5 years
Eclipse UI	24	6 years
JDT	21	6.5 years

We asked our participants to report on the bug reports they worked on, the start time and end time of each bug report. By analyzing this data, we are able to know the number of bugs they worked on, the number of resolved bugs, the resolution time of each bug report, and the disruption cost by looking at the end time of one bug and the start time of another bug.

As described in Table 4, we used six open-source systems:

- **Eclipse UI** is the user interface of the Eclipse development framework.
- **Eclipse Jetty** is a Java HTTP server and Java Servlet container.
- **Eclipse AspectJ** is an aspect-oriented programming (AOP) extension created for the Java programming language.
- **Eclipse Birt** provides reporting and business intelligence capabilities.
- **Eclipse SWT** is a graphical widget toolkit.

– **Eclipse JDT** provides a set of tool plug-ins for Eclipse.

Table 4 shows the different statistics of the analyzed systems including the time range of the bug reports, the number of bug reports, the number of closed and resolved bugs in a project, the number of developers involved with project and the average of time spent to resolve a bug and close its corresponding bug report. The total number of collected unresolved bug reports is about 63,000 bug reports for the six open source systems. All these projects are using BugZilla tracking system and GIT as a version control system.

4.3 Pre-study Survey

The goal of the pre-study survey is to understand our participants, their background in software engineering and related experience. The list of questions were asked are:

– What is your highest level of education?
– What is your current occupation?
– How many years have you worked in software engineering?
– Choose the level (very low, low, normal, high, very high) of expertise in: (1) Software Development, (2) Software Management, (3) Software Testing, (4) JAVA, (5) Software Quality Assurance

4.3.1 Post-study Survey

The goal of the post-study survey is to gather our participants' feedback about the importance of bug prioritization and the usefulness of our tool to prioritize bug reports. The list of questions were asked are:

– Q1: How difficult was it to resolve bugs in the order that was presented?
– Q2: How difficult is it for bug prioritization tools to save developer's time to resolve multiple bugs in a particular period of time?
– Q3: How difficult was it to resolve bugs as first come first serve compared to bug prioritization tools?

4.4 Meta-heuristic Parameters Tuning

An often-omitted aspect in meta-heuristic 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 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 = 0.3 where the probability of gene modification is 0.1. [7]

The Friedman test is the non-parametric alternative to the one-way ANOVA with repeated measures. The Friedman statistical tests show that all the comparisons performed between our approach and existing ones are statistically significant based on all the metrics and the systems considered in our experiments. We used a 95% confidence level ($alpha = 5\%$) to find out whether our sample results of different approaches are significantly different.

4.5 Results

4.5.1 Results for RQ1

For this research question, we examined the number of bugs that the developers were able to resolve within the 2-hour window. Figure 5 shows the difference in performance between our multi-objective approach and the first come first serve approach. Furthermore, we measured the effectiveness of the mono-objective approaches by considering separately the score of bug priority or bug dependency. The results show that the multi-objective combining the benefits of both mono-objective approaches are presenting better results in terms of fixing bugs.

Figure 6 describes the average time spent by the developer to resolve one single defect in a certain project. This figure shows the difference in the number of minutes between our multi-objective approach and other three different approaches such as first come first serve, bug priority, and bug dependency approach. We found that the familiarity with the associated files to a bug play an important factor in the time that the developer may spend on one individual bug which explains the significant effectiveness of our approach.

Figure 7 presents the disruption cost or cognitive efforts needed to completely shift from one bug to another. We found that this cost is too high in FCFS and medium in Bug Priority but it drops significantly in Bug Dependency or multi-objective approach which shows the benefit of considering bugs dependency to improve the productivity of the developers. Cognitive effort is the time spent by the developer to make the transition between one bug to another unrelated bug. This transition involves the time to change the developer's focus to understand the information given to the developer in the new bug and the time to examine the files related to the new bugs.

To conclude, it is clear that the multi-objective approach significantly reduce the efforts spent by the developers to fix bugs when they are ranked based on a combination of their dependency and priority.

4.5.2 Results for RQ2

Figure 5 and Figure 6 confirm the efficiency of our multi-objective approach over other techniques used to prioritize bug reports based on severity or first come first served. In Figure 5, our approach shows an average of 3 defects in 2-hour window for all evaluated projects whereas first come first serve (FCFS) and Bug Priority approach shows an average of 1 defect in a given time window. Bug Dependency technique produces a promising result with an average of 2.5 which is very close to multi-objective approach's outcome and that is due to the importance of recommending the bugs that share the same set of files/classes to inspect. The complexity of the project plays an important role in localizing and fixing bugs, developers localized and fixed 2 to 3 bugs in Eclipse UI or JDT projects as opposed to 5 bugs in Jetty.

In Figure 6, the multi-objective approach has as low as 21 minutes and as high as 78 minutes on average to resolve a single defect. Bug dependency comes next in efficiency after the multi-objective approach with a low of 28 and high of 67 minutes. The third approach is Bug Priority with unremarkable results of 78 minutes on average. FCFS result is considered the worst with an average of 123 minutes since it does not follow any dynamic strategy in choosing the next bug in line to resolve. We noticed a big gap between FCFS and others as FCFS does not consider the complexity, size, severity, and urgency of the bug but rather goes from one bug to another. Our approach helps to reduce the resolution time even in the large and complicated systems, 187, 176, and 154 minutes were recorded for FCFS in Birt, Eclipse UI, and JDT respectively and 66, 44, and 78 minutes were recorded in multi-objective approach for those same projects. As a result, using a multi-objective approach saves significant time in fixing bugs compared to the FCFS approach.

Figure 7 shows an average of 6 minutes in multi-objective and 8 minutes in Bug Dependency approach. One of the reasons that make the localization and fixing time too high in FCFS is the high disruption time of 39 minutes on average. Bug Priority does slightly better than FCFS with 22 minutes but Bug Priority is still far away from Bug Dependency or multi-objective approach. Furthermore, we noticed that the disruption cost increases when the size of the project becomes larger. Birt is an example of a large project which required 10 minutes of disruption cost whereas it is around 5 minutes for other smaller projects like Jetty.

To conclude, the proposed multi-objective approach outperforms mono-objective ones which confirm the need to consider bugs dependencies when scheduling them to be repaired by developers.

4.5.3 Post-study Survey

The goal of the post-study survey is to gather our participants' feedback about the importance of bug prioritization and the usefulness of our tool to prioritize bug reports. The list of questions were asked are:

- Q1: How difficult was it to resolve bugs in the order that was presented?

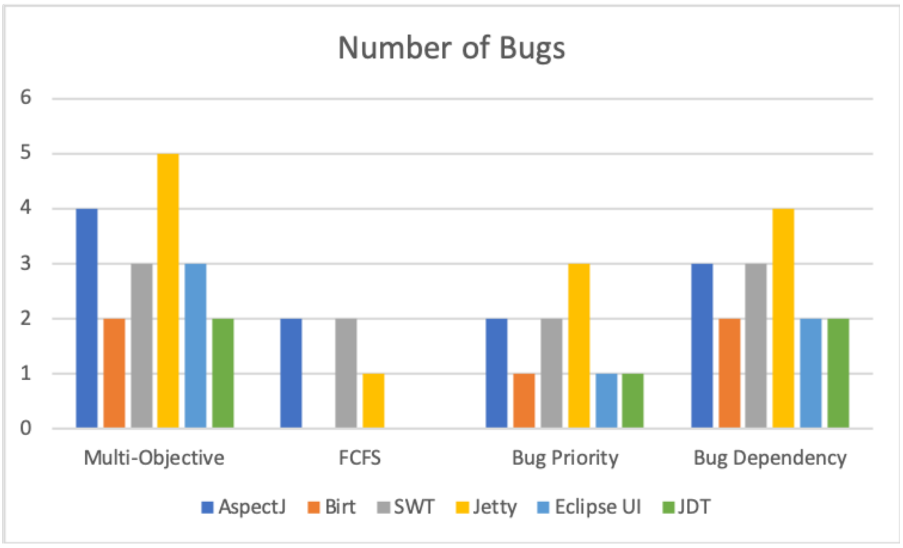

Fig. 5 Comparison of number resolved bugs in 2-hour window using our prioritization tool versus FCFS tool along with two of mono-objective approaches for each of the six projects

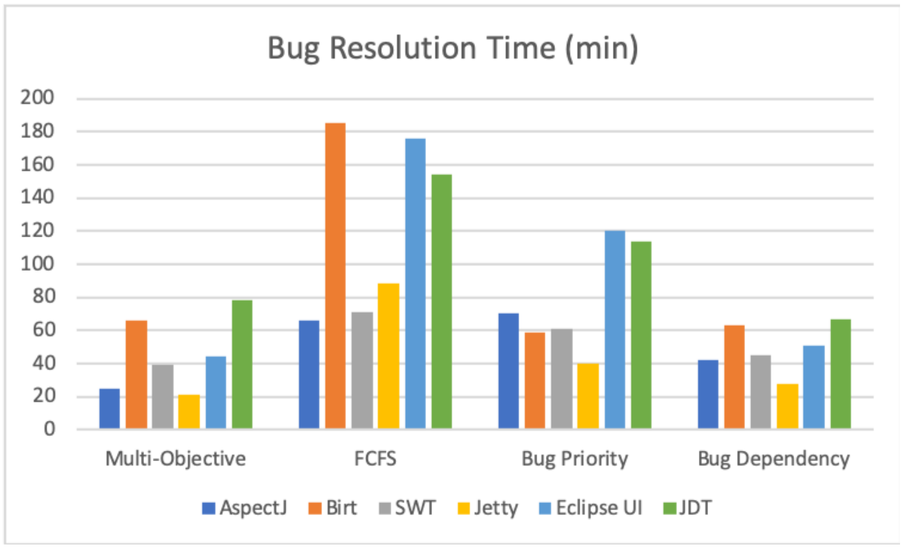

Fig. 6 Comparison of average time spent to resolve a particular bug using our prioritization tool versus FCFS tool along with two of mono-objective approaches for each of the six projects

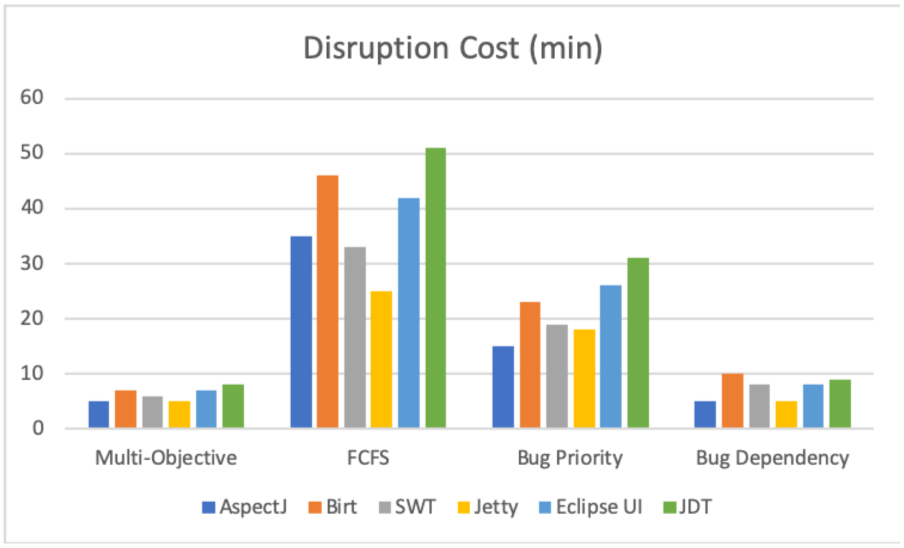

Fig. 7 Comparison of disruption cost to transit from one bug to another using our prioritization tool versus FCFS tool along with two of mono-objective approaches for each of the six projects

- Q2: How difficult is it for bug prioritization tools to save developer's time to resolve multiple bugs in a particular period of time?
- Q3: How difficult was it to resolve bugs as first come first serve compared to bug prioritization tools?

4.5.4 Pre-study Survey Results

All the participants have a job in industry as software engineer or technical lead. 87% of our participants hold a bachelor degree in computer science, Table 5 shows the list of six (6) open source software used in the study along with the number of developers who participate in each of those projects with average years of experience of those participants. Figure 8 shows the distribution of expertise for our participants regarding the 5 different categories listed in the questionnaire. 16 participants were working on software testing and bug repair tasks as part of their regular duties, which was one of the main criteria used to solicit their participation, based on our previous collaborations and contacts.

4.5.5 Post-study Survey Results

Chart 9 shows the results we gathered from our participants about the three post-study survey's questions. For Q1, we found that 72% thought that the recommended solution (the order of resolving the bugs) made the whole task easier than normal. For Q2, the majority, over 50% found that the new approach tends to save developers' time to localize bugs and resolve. For Q3, we

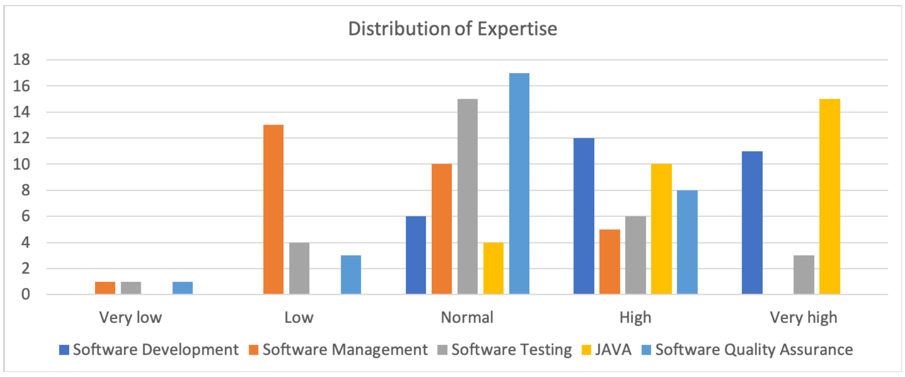

Fig. 8 Distribution of Expertise for the participants in the pre-study survey

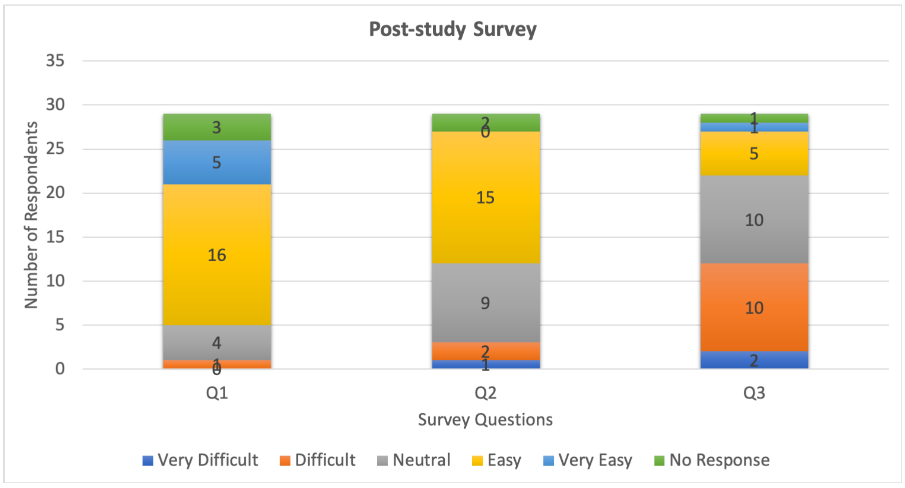

Fig. 9 Post-study survey results

found that our participants have noticed the difference between First Come First Serve (FCFS) and our approach in which 12 developers reported that task was difficult, and 10 developers found it neutral where they did not notice any improvements.

5 Threats to Validity

We want to acknowledge several threats to the validity of the paper such as the factors that can bias our empirical study. These factors can be classified into three categories: internal validity, 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 bug localization technique to generate a dependency graph among several bug reports and therefore recommend those bugs in sequential order. For that reason, we compared our proposal with different mono-objective formulations that use one metric only like the score of bug priority. The developers were asked to evaluate different systems using different tools. We did not allow developers to evaluate different tools on the same system. The developers were distributed among the systems and tools based on their background/expertise to ensure almost the same level for all systems and tools. When each developer is asked to evaluate one different tool per system, we reduce the potential bias in the experiments since they are using the tools for the first time and they are exploring each time a new system. Our results show that the productivity has gotten better for the majority of our developers regardless of their experience and skills set.

External validity refers to the fact that our survey has been conducted by 29 developers with a variety of skills and number of experience. Thus, we can affirm that our results will hold its accuracy with a different set of developers with different level of expertise or knowledge. Also, time collection was left to each individual developer who manually noted the time they started and finished localizing a defect. This could have resulted in introducing error as every developer performed differently.

Finally, External validity could be related to the type of projects we used in the survey in which we used six different widely-used open-source systems belonging to the 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.

Conclusion validity is concerned with the statistical relationship between the treatment and the outcome. 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. The parameters' values used in our experiments are found by trial-and-error, which is commonly used in the SBSE community. However, it would be an interesting perspective to design an adaptive parameter tuning strategy for our approach so that parameters are updated during the execution in order to provide the best possible performance.

6 Conclusion and Future work

We proposed an approach for bugs management by taking into consideration both the severity and dependencies between reports. Our solution is based on the use of multi-objective search to find a trade-off between these two conflicting objectives. The validation of our work shows that there were significant time savings when developers inspected bugs comparing to existing methods treating each bug individually as first come first serve or relaying on priority scores only.

As part of our future work, we envision the extension of this approach to improve the bugs management process by recommending developers to be assigned for bugs based on their background and prior expertise. The users can interact more with the suggested recommendations in order to update the assignments. In addition, we are planning to extend our current work with multiple other bug repository systems beyond Bugzilla. We also would like to validate the proposed tool on proprietary software systems to generalize the obtained results.

References

1. T. Zhang, J. Chen, G. Yang, B. Lee, and X. Luo, "Towards more accurate severity prediction and fixer recommendation of software bugs," *Journal of Systems and Software*, vol. 117, pp. 166–184, 2016.
2. J. A. Jones, "Semi-automatic fault localization," Ph.D. dissertation, Georgia Institute of Technology, 2008.
3. N. Bettenburg, R. Premraj, T. Zimmermann, and S. Kim, "Duplicate bug reports considered harmful… really?" in *2008 IEEE International Conference on Software Maintenance*. IEEE, 2008, pp. 337–345.
4. M. Fischer, M. Pinzger, and H. Gall, "Analyzing and relating bug report data for feature tracking," in *WCRE*, vol. 3, 2003, p. 90.
5. W. Zou, D. Lo, Z. Chen, X. Xia, Y. Feng, and B. Xu, "How practitioners perceive automated bug report management techniques," *IEEE Transactions on Software Engineering*, 2018.
6. W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, "A survey on software fault localization," *IEEE Transactions on Software Engineering*, vol. 42, no. 8, pp. 707–740, 2016.
7. R. Almhana, W. Mkaouer, M. Kessentini, and A. Ouni, "Recommending relevant classes for bug reports using multi-objective search," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ACM, 2016, pp. 286–295.
8. J. Uddin, R. Ghazali, M. M. Deris, R. Naseem, and H. Shah, "A survey on bug prioritization," *Artificial Intelligence Review*, vol. 47, no. 2, pp. 145–180, 2017.
9. K. Chaturvedi and V. Singh, "Determining bug severity using machine learning techniques," in *2012 CSI Sixth International Conference on Software Engineering (CONSEG)*. IEEE, 2012, pp. 1–6.
10. A. X. Zheng, M. I. Jordan, B. Liblit, M. Naik, and A. Aiken, "Statistical debugging: simultaneous identification of multiple bugs," in *Proceedings of the 23rd international conference on Machine learning*. ACM, 2006, pp. 1105–1112.
11. G. Canfora, M. Ceccarelli, L. Cerulo, and M. Di Penta, "How long does a bug survive? an empirical study," in *2011 18th Working Conference on Reverse Engineering*. IEEE, 2011, pp. 191–200.
12. Z. Li, L. Tan, X. Wang, S. Lu, Y. Zhou, and C. Zhai, "Have things changed now?: an empirical study of bug characteristics in modern open source software," in *Proceedings of the 1st workshop on Architectural and system support for improving software dependability*. ACM, 2006, pp. 25–33.
13. K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: Nsga-ii," *IEEE transactions on evolutionary computation*, vol. 6, no. 2, pp. 182–197, 2002.
14. J. Geng, S. Ying, X. Jia, T. Zhang, X. Liu, L. Guo, and J. Xuan, "Supporting many-objective software requirements decision: An exploratory study on the next release problem," *IEEE Access*, vol. 6, pp. 60 547–60 558, 2018.

15. L. Yu, W.-T. Tsai, W. Zhao, and F. Wu, "Predicting defect priority based on neural networks," in *International Conference on Advanced Data Mining and Applications.* Springer, 2010, pp. 356–367.
16. N. Goyal, N. Aggarwal, and M. Dutta, "A novel way of assigning software bug priority using supervised classification on clustered bugs data," in *Advances in Intelligent Informatics.* Springer, 2015, pp. 493–501.
17. J. Xuan, H. Jiang, Z. Ren, and W. Zou, "Developer prioritization in bug repositories," in *2012 34th International Conference on Software Engineering (ICSE).* IEEE, 2012, pp. 25–35.
18. M. Alenezi and S. Banitaan, "Bug reports prioritization: Which features and classifier to use?" in *2013 12th International Conference on Machine Learning and Applications*, vol. 2. IEEE, 2013, pp. 112–116.
19. A. Lamkanfi, S. Demeyer, Q. D. Soetens, and T. Verdonck, "Comparing mining algorithms for predicting the severity of a reported bug," in *2011 15th European Conference on Software Maintenance and Reengineering.* IEEE, 2011, pp. 249–258.
20. J. Kanwal and O. Maqbool, "Managing open bug repositories through bug report prioritization using svms," in *Proceedings of the International Conference on Open-Source Systems and Technologies, Lahore, Pakistan*, 2010, pp. 22–24.
21. W. Abdelmoez, M. Kholief, and F. M. Elsalmy, "Bug fix-time prediction model using naïve bayes classifier," in *2012 22nd International Conference on Computer Theory and Applications (ICCTA).* IEEE, 2012, pp. 167–172.
22. S. J. Dommati, R. Agrawal, S. S. Kamath *et al.*, "Bug classification: Feature extraction and comparison of event model using na\" ive bayes approach," *arXiv preprint arXiv:1304.1677*, 2013.
23. J. Kanwal and O. Maqbool, "Bug prioritization to facilitate bug report triage," *Journal of Computer Science and Technology*, vol. 27, no. 2, pp. 397–412, 2012.
24. M. Sharma, P. Bedi, K. Chaturvedi, and V. Singh, "Predicting the priority of a reported bug using machine learning techniques and cross project validation," in *2012 12th International Conference on Intelligent Systems Design and Applications (ISDA).* IEEE, 2012, pp. 539–545.
25. F. Thung, D. Lo, L. Jiang, F. Rahman, P. T. Devanbu *et al.*, "When would this bug get reported?" in *2012 28th IEEE International Conference on Software Maintenance (ICSM).* IEEE, 2012, pp. 420–429.
26. Y. Tian, D. Lo, and C. Sun, "Information retrieval based nearest neighbor classification for fine-grained bug severity prediction," in *2012 19th Working Conference on Reverse Engineering.* IEEE, 2012, pp. 215–224.
27. M. S. Zanetti, I. Scholtes, C. J. Tessone, and F. Schweitzer, "Categorizing bugs with social networks: a case study on four open source software communities," in *Proceedings of the 2013 International Conference on Software Engineering.* IEEE Press, 2013, pp. 1032–1041.
28. D. Behl, S. Handa, and A. Arora, "A bug mining tool to identify and analyze security bugs using naive bayes and tf-idf," in *2014 International Conference on Reliability Optimization and Information Technology (ICROIT).* IEEE, 2014, pp. 294–299.
29. H. Valdivia Garcia and E. Shihab, "Characterizing and predicting blocking bugs in open source projects," in *Proceedings of the 11th working conference on mining software repositories.* ACM, 2014, pp. 72–81.
30. M. Harman and B. F. Jones, "Search-based software engineering," *Information and software Technology*, vol. 43, no. 14, pp. 833–839, 2001.
31. A. Núñez, M. G. Merayo, R. M. Hierons, and M. Núñez, "Using genetic algorithms to generate test sequences for complex timed systems," *Soft Computing*, vol. 17, no. 2, pp. 301–315, 2013.
32. C. Henard, M. Papadakis, and Y. Le Traon, "Mutation-based generation of software product line test configurations," in *International Symposium on Search Based Software Engineering.* Springer, 2014, pp. 92–106.
33. J. Shelburg, M. Kessentini, and D. R. Tauritz, "Regression testing for model transformations: A multi-objective approach," in *International Symposium on Search Based Software Engineering.* Springer, 2013, pp. 209–223.

34. D. Dreyton, A. A. Araújo, A. Dantas, Á. Freitas, and J. Souza, "Search-based bug report prioritization for kate editor bugs repository," in *International Symposium on Search Based Software Engineering.* Springer, 2015, pp. 295–300.
35. D. Dreyton, A. A. Araújo, A. Dantas, R. Saraiva, and J. Souza, "A multi-objective approach to prioritize and recommend bugs in open source repositories," in *International Symposium on Search Based Software Engineering.* Springer, 2016, pp. 143–158.
36. C. Sun, D. Lo, X. Wang, J. Jiang, and S.-C. Khoo, "A discriminative model approach for accurate duplicate bug report retrieval," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1.* ACM, 2010, pp. 45–54.
37. S. T. Dumais, "Latent semantic analysis," *Annual review of information science and technology*, vol. 38, no. 1, pp. 188–230, 2004.
38. D. M. Blei, A. Y. Ng, and M. I. Jordan, "Latent dirichlet allocation," *Journal of machine Learning research*, vol. 3, no. Jan, pp. 993–1022, 2003.
39. G. Salton, A. Wong, and C.-S. Yang, "A vector space model for automatic indexing," *Communications of the ACM*, vol. 18, no. 11, pp. 613–620, 1975.
40. X. Ye, R. Bunescu, and C. Liu, "Learning to rank relevant files for bug reports using domain knowledge," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering.* ACM, 2014, pp. 689–699.
41. A. Ramirez, J. R. Romero, and S. Ventura, "A survey of many-objective optimisation in search-based software engineering," *Journal of Systems and Software*, vol. 149, pp. 382–395, 2019.
42. A. Ghannem, G. El Boussaidi, and M. Kessentini, "On the use of design defect examples to detect model refactoring opportunities," *Software Quality Journal*, vol. 24, no. 4, pp. 947–965, 2016.
43. B. Amal, M. Kessentini, S. Bechikh, J. Dea, and L. B. Said, "On the use of machine learning and search-based software engineering for ill-defined fitness function: a case study on software refactoring," in *International Symposium on Search Based Software Engineering.* Springer, Cham, 2014, pp. 31–45.
44. M. Kessentini, A. Ouni, P. Langer, M. Wimmer, and S. Bechikh, "Search-based metamodel matching with structural and syntactic measures," *Journal of Systems and Software*, vol. 97, pp. 1–14, 2014.
45. A. Ghannem, G. El Boussaidi, and M. Kessentini, "Model refactoring using examples: a search-based approach," *Journal of Software: Evolution and Process*, vol. 26, no. 7, pp. 692–713, 2014.
46. A. Ghannem, M. Kessentini, and G. El Boussaidi, "Detecting model refactoring opportunities using heuristic search," in *Proceedings of the 2011 Conference of the Center for Advanced Studies on Collaborative Research*, 2011, pp. 175–187.
47. "Bug reports data," http://bit.ly/2NUyion, accessed: 2020-01-20.
48. A. A. Keller, *Multi-Objective Optimization in Theory and Practice II: Metaheuristic Algorithms.* Bentham Science Publishers, 2019.
49. M. T. Emmerich and A. H. Deutz, "A tutorial on multiobjective optimization: fundamentals and evolutionary methods," *Natural computing*, vol. 17, no. 3, pp. 585–609, 2018.
50. K. Deb and S. Gupta, "Understanding knee points in bicriteria problems and their implications as preferred solution principles," *Engineering optimization*, vol. 43, no. 11, pp. 1175–1204, 2011.