Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

A scheduling-driven approach to efficiently assign bug fixing tasks to developers[☆]

Vahid Etemadi[a], Omid Bushehrian[a], Reza Akbari[a], Gregorio Robles[b,*]

[a] *Department of Computer Engineering and Information Technology, Shiraz University of Technology, Shiraz, Iran*
[b] *Universidad Rey Juan Carlos, Madrid, Spain*

A R T I C L E   I N F O

Article history:
Received 15 April 2020
Received in revised form 13 February 2021
Accepted 29 March 2021
Available online 17 April 2021

Keywords:
Bug assignment
Scheduling-driven
Bug fixing
Software maintenance

A B S T R A C T

The efficient assignment of bug fixing tasks to software developers is of major importance in software maintenance and evolution. When those tasks are not efficiently assigned to developers, the software project might confront extra costs and delays. In this paper, we propose a strategy that minimizes the *time* and the *cost* in bug fixing by finding the best feasible developer arrangement to handle bug fixing requests. We enhance therefore a state-of-the-art solution that uses an evolutionary bi-objective algorithm by involving a *scheduling-driven* approach that explores more parts of the search space. Scheduling is the process of evaluating all possible orders that developers can follow to fix the bugs they have been assigned. Through an empirical study we analyze the performance of the *scheduling-driven* approach and compare it to state of the art solutions. A non-parametric statistical test with four quality indicator metrics is used to assure its superiority. The experiments using two case-studies (JDT and Platform) showed that the scheduling-driven approach is superior to the state of the art approach in 71% and 74% of cases, respectively. Thus, our approach offers superior performance by assigning more conveniently bug fixing tasks to developers, while still avoiding to overload developers.

1. Introduction

Regardless of dealing with an open source or a commercial software, software evolution and maintenance teams of large software projects receive many bug reports daily (Xia et al., 2015b). If there is no systematic approach to handle those requests, the time required to close them may inevitably increase (Alencar da Costa et al., 2016). Moreover, when bug reports are not efficiently assigned to software developers, the software project might confront inefficiencies. In such cases, the amount of time required to perform bug fixing could grow. In addition, if delays have a penalization, extra costs could be incurred by the project (Wu et al., 2012).

If we model this problem, professional developers can be seen as an asset of any development team. They can be assigned to incoming bug reports in many ways (which we present in related research, in Section 2). Greedy, machine learning and information retrieval approaches can recommend the developer that fits best to a particular task, but these approaches cannot prevent developers to become overloaded because too many tasks[1] are being assigned to them. Some scholars have shown that meta-heuristic approaches, like the Non-dominated Sorting Genetic Algorithm 2 (NSGA-II), can help to achieve a balance in task assignment where tasks are addressed and developers not overloaded (Karim et al., 2016). Nonetheless, if the modeling of task assignment includes more details of the problem space, NSGA-II could be more effective in ending up with more optimal solutions.

From the problem solving perspective, we first need to represent the solutions to use NSGA-II. Typically, prior to encoding chromosomes, we have to sort the tasks coming from a Task Dependency Graph (TDG).[2] TDGs of bug fixing task assignment are mostly sparse, which means there are almost no edges among tasks (see Section 4.2). During population generation, each task gets assigned to an individual developer. Because the number of bugs is most likely to be greater than the number of developers, each developer will be assigned to more than one task. If this is the case, NSGA-II could be more effective if it can include particular encodings in a way that different orders of a task assignment get covered. This fact offers, in our point of view, room for improvement in better exploration.

Thereby, in this paper we examine the time spent by a developer when *he/she follows a different order of performing the*

[☆] Editor: Raffaela Mirandola.
[*] Corresponding author.
E-mail address: grex@gsyc.urjc.es (G. Robles).
[1] All change requests could be considered as incoming tasks or, particularly in this paper, incoming bugs.

[2] A TDG involves all dependencies that exist among the code parts comprising the incoming tasks.

assignments. Thus, the order of performing the tasks assigned to a developer matters, particularly, in terms of time and cost. We refer to this kind of orders as a *schedule*, and our approach as Scheduling-Driven (SD).

In this paper we target the problem of bug (task) assignment. We aim at minimizing the time and the cost of bug assignment using a search-based software engineering approach. Two different aspects are considered to minimize the time and the cost: (i) the best arrangement, in terms of the best match between a developer profile and the bug zone (the assignment part), and (ii) the best schedule to minimize the delay time and the delay cost (the scheduling-driven part). To include these aspects, several steps are followed: (i) creating example assignments and schedules, (ii) evaluating the candidate solutions, and (iii) selecting the best solutions. In this manner, together with evaluating different arrangements, different schedules are evaluated as well. Eventually, each final assignment comes with a schedule that offers developers a particular order for performing their assignments. After having evaluated all the arrangements and their associated schedules, a bug triager can select a particular solution with the minimum time, or the minimum cost. We thus believe that scheduling can be considered as a supplement for solving a bug assignment problem.

The gain in efficiency occurs when there is the risk of *bad scheduling*, as for instance when the workload of available developers rises because a few number of developers are always chosen by the models — this frequently happens when machine learning algorithms (Karim et al., 2016; Xia and Lo, 2017) or information retrieval techniques (Xia et al., 2015b; Kagdi et al., 2012) are applied.

The cost is computed as the wages paid to developers plus the incurred penalties due to the late delivery of a corrected version of the software. In addition, if predefined task deadlines are violated, there is an additional cost computed by a given Service Level Agreement (SLA)[3] function. Many SLAs include penalties for delays. The penalties usually depend on the quality level of the provided services (i.e., the importance/urgency of a bug) (Catolino et al., 2019; Wu et al., 2015; Yeo and Buyya, 2005; Zhang et al., 2016a). Thus, a *good scheduling* in this context means keeping the best order for handling incoming tasks by candidate developers in presence of SLAs and in consideration of the TDG (Maalej et al., 2017).

Our investigation is built on top of an approach by Karim et al. (2016), particularly *CompetenceMulti$_2$@DAB*, that proposes an efficient solution, which we will refer to as KRRGZ. KRRGZ uses NSGA-II for treating the problem of task assignment, with the aim to minimize *time* and *costs*. In more detail, cost is a dependent variable that depends (1) on the *time* developers spend fixing a bug, and (2) the penalty cost due to a delay in case of an SLA violation. Our aim is to improve KRRGZ using SD — we hypothesize that exploration can be augmented by considering *hidden* parts of the search space. The hidden parts refer to the parts of the search space representing the possible schedules of tasks assigned to a set of developers. SD covers this part of the search space through evaluating different schedules for a given assignment.

1.1. Motivating example

Let us assume a team of three developers (D_1, D_2, D_3) maintaining a software project. There are five bugs (B_1, B_2, B_3, B_4, B_5) filed in the issue tracking system that can be assigned to the three developers. Bugs are first assigned randomly to the three developers. This assignment is evaluated in terms of the time it takes and the cost that is incurred by the project. Other assignments are computed and evaluated with the help of NSGA-II, as they can lead to different delay times and to different final costs. At the end of the process, the best solution is selected.

When there are few dependencies among bugs, KRRGZ can be improved, as we have more freedom to choose the order of fixing bugs. Obviously, the order in which bugs are assigned to a developer is important and needs to be incorporated in the model. In the proposed model this shortcoming is overcome by presenting a new encoding that represents not only the assignment of tasks to developers, but also the order in which the assigned tasks are performed. In such a scenario, it is where SD offers to evaluate different possible schedules (the orders) of a given assignment that could work better for *all* developers, while KRRGZ is optimized for individual developers. Thus, **SD looks for the best solution for the whole project**. In addition, KRRGZ does not consider relevant information on the bugs (i.e., penalties due to delays as defined in an SLA), but SD does.

In detail, since developers might be assigned to several tasks, they need to be given an order (e.g., [B_2, B_3, B_5, B_4, B_1] as the best one). Our aim is to find the best possible order, i.e., that order where the combination of time and cost is the lowest (or among the lowest) among all possible orders. In the case where there are multiple orders that are equally efficient (i.e., the combination of time and cost is equivalent), project managers can choose among them, prioritizing one over the other.

1.2. Contributions and structure of the paper

In this paper, we propose a strategy that minimizes the time and the cost in bug fixing by finding the best feasible developer arrangement to handle bug fixing requests. We enhance therefore a state-of-the-art solution (KRRGZ) by involving a SD approach which explores more parts of the search space. To achieve this, we basically applied NSGA-II to solve a bi-objective task assignment problem. By using NSGA-II as the base algorithm, we need to represent the encoded solution as well as how the fitness function is formulated. Therefore, given the encoded solution and the fitness function of KRRGZ, we have made following extensions to it:

- The encoded solution has been extended with a schedule vector which keeps the best order for performing tasks by developers.
- The fitness function, comprised of the time and the cost, is extended by taking into account the delay time and the delay cost due to potentially passing the deadlines (and incurring into a penalty).
- The evaluation to achieve better schedules is done by considering a local search strategy.

The inputs for both algorithms (KRRGZ and SD) are: (i) the incoming bug requests, (ii) the number of available developers, and (iii) the source code repositories. Access to the source code repositories is needed in order to calculate the productivity of developers and to extract the TDG.

We perform an empirical study where we compare the performance of our approach with several state of the art solutions, using data by Karim et al. (2016).

The rest of the paper is organized as follows: Section 2 discusses related work. Next, the study setup is introduced in Section 3. The organization of the empirical study is offered in Section 4. Section 5 provides the results and shows the improvement over other approaches. We discuss the limitations and threats to validity in Section 6. Finally, we draw conclusions and point to further research in Section 7.

[3] An SLA is a commitment between a service provider and a client where particular aspects of the service, such as quality, availability, responsibilities, are agreed between the service provider and the client (Kearney and Torelli, 2011).

2. Related work

Bug triaging and assignment has been a widely studied topic in recent years in software engineering research. The techniques being used can be classified in three groups: (i) Machine Learning; (ii) Information Retrieval; and (iii) Metaheuristic Approaches.

"Who should fix this bug?" was one of the first attempts of using machine learning algorithms to address the problem of task assignment in software development (Anvik et al., 2006). In it, the authors present an exhaustive bug life cycle which is used for tracking the status of bugs. Xia et al. followed an incremental learning approach to propose a new version of a topic modeling algorithm, called *Topic Miner*, to find the best set of assignees for fixing incoming bugs (Xia et al., 2016). Zhang et al. (2016b) provide a useful survey on the work-flow of bug triaging. Xuan et al. address the problem of bug triaging, which supposes almost 45% of the cost in software projects (Pressman, 2010) by proposing a data reduction method (Xuan et al., 2015). The method synthesizes feature and instance selection to diminish the bugs and corresponding words leading to reducing the scale of the data. They employed a binary classifier to forecast the order of feature and instance selection which affects the quality of the reduced data. This process improves the time and the accuracy of bug triaging. Cavalcanti et al. use contextual information to provide a rule-based system to improve the assignment of change requests to developers (Cavalcanti et al., 2016). The authors address the dynamicity in the developer team structure as a main concern that needs to be taken into consideration when assigning tasks.

Shokripour et al. proposed to use the recency of using a term by a developer in determining the values of the developer expertise on top of term frequency-inverse document frequency (tf-idf) (Shokripour et al., 2015). Thus, they included time-related metadata in the process of assignment of tasks.

Kagdi et al. used Information Retrieval methods to (i) find the relevant source code for given change requests, and (ii) mine the repositories to recommend a ranked list of developers who could change those parts of the code (Kagdi et al., 2012). Xia et al. employed a paired score function to rank potential developers that could fix incoming bugs (Xia et al., 2015a). For each incoming bug, they measure (i) the distance between the bug and all previously handled bugs, and (ii) the affinity of the incoming bug and the developers' profile. Then, by summing the two values, each developer is assigned a normalized value that finally results in a ranked list of potential developers to perform a task. Although having a high accuracy, this approach does not avoid the risk of overloading a developer with too many bug assignments.

Di Penta et al. point out that resource allocation and task assignment in software projects is an NP-hard problem (Di Penta et al., 2011). They introduce a search-based optimization approach with a queuing model. Their solution focuses on providing the order in which *work packages* should be processed. The interdependencies among work packages play an important role in the formulation of the problem, as they do in this paper.

Stylianou et al. study *human resource allocation and scheduling* as a major factor in management decisions (Stylianou and Andreou, 2014). They provide a comprehensive review on the research works related to this topic. They also refer to *Software Process Simulation* as an activity commonly used to deal with problems that need to be treated over time. It uses an *evolutionary approach* to solve problems that require exploring a search space. Similarly, we use the evolutionary approach as a solution for a multi-objective task assignment problem. Moreover, Ren et al. show a *Cooperative Co-evolution* approach that provides two types of encodings per solution: one for the searching problem of staff assignment and another one for job scheduling in software projects (Ren et al., 2011). The first part is concerned with how tasks are assigned to developers. In the second part, the authors try to mitigate the waiting time for those work packages that are arranged consecutively. This is achieved by including work packages that do not have prerequisites amongst those that are consecutively dependent. In our study, we follow a similar approach, as we consider how to assign tasks to developers for a specific solution in order to encode a schedule (see Section 3.4.2 for further details).

Finally, Karim et al. addressed the problem of developers being assigned too many tasks with what we have called the KRRGZ approach (Karim et al., 2016). The authors used a multi-objectives paradigm to model the problem of task assignment in order to prevent overloading developers. They use a competency-based approach to assign tasks to developers (i.e., tasks are assigned to those developer that are more competent on them). The two main objectives of KRRGZ are (the minimization of) time and cost. As noted above, our research departs from Karim et al.'s approach and improves it with a scheduling-driven task assignment.

3. Study setup

In this Section we offer the details on how we have performed our study. We start with an abstract representation of the proposed method for task assignment in software maintenance. Since we are dealing with a bi-objective task assignment problem, we elaborate on the formulation of the objectives in detail.

3.1. Scheduling-driven task assignment

Task assignment is part of the work-flow to perform bug fixing on the software codebase. It is assumed to equip project managers with facilities to prevent the risk of using too many resources for maintaining the software.

Fig. 1 represents the entire pipeline of the incoming change tasks to be assigned to developers. The steps shown as boxes and linked together with arrows are activities that could be performed manually, semi-automated or fully automated. The focus in this paper will be on the task assignment box where we will include *schedules* that create assignment plans for developers. Schedules can be understood as an extension to the assignment plan that dictates the order of doing bug fixing by developers.

The process starts with an incoming bug fix request, in general in natural language. The request is then mapped to the code that needs to be changed. The change impact analysis box outputs the impact sets. In this paper, we assume that the impact set is composed by the packages (files) to be modified. The change impact analysis algorithm aims at automating the action of "locating part(s) of the code which need to be changed in response to a change request". There are several techniques that could be used in this regard, including (i) information retrieval-based (Zhou et al., 2012; Zhang et al., 2019), (ii) machine learning-based (Xia et al., 2016), and (iii) heuristic-based (Arrieta et al., 2018). Based on the context and the accuracy of the mapping tools presented in the literature, one or a hybrid approach of those could be selected for the change impact set generation. Depending on the type of tool, it receives different inputs, although these are usually of two types: (i) the bug report and (ii) the source code repository. For this reason, in Fig. 1, bug reports are fed into the change request analysis together with the source code repository. In this paper we use the impact set used by KRRGZ, as provided by Karim et al. (2016).

The assignment of tasks follows an SD approach upon a search-based assignment solution. The input for this step are developer profiles and a TDG, where tasks are broken down into subtasks. The proposed method is equipped with the facility of computing

Table 1
List of notations used in this paper.

D_i	$\triangleq$	Developer with id i
i	$\triangleq$	Index value
T_i	$\triangleq$	Task i
ST_{ij}	$\triangleq$	SubTask j belonging to task i
TC	$\triangleq$	Total cost of assignment
TDG	$\triangleq$	Task dependency graph
SD	$\triangleq$	Scheduling driven approach
$KRRGZ$	$\triangleq$	Karim et al.'s (2016) approach
PE	$\triangleq$	Permutation encoding
RS	$\triangleq$	Random search
HV	$\triangleq$	Hypervolume
GD	$\triangleq$	Generational distance
C	$\triangleq$	Contribution
S	$\triangleq$	Spacing
I_{HV}	$\triangleq$	Hypervolume indicator
I_{GD}	$\triangleq$	Generational distance indicator
I_C	$\triangleq$	Contribution indicator
I_S	$\triangleq$	Spacing indicator
NFE	$\triangleq$	Number of fitness evaluation

the delay time and the cost rather than the time and the cost. The final output is a set of assignments of ordered tasks along with their optimal schedule. These assignments have the property of being non-dominated assignments (i.e., no other solution is better although other solutions might exist that are equally efficient) in terms of resulting in minimum time and cost.

The system relies on a meta-heuristic approach that has stochastic nature. This means that we cannot assure the best solutions are to be found, but that the best possible ones of the algorithm set-up are achievable.

3.2. The objectives: Problem formulation

In this section we dive into the details of formulating the goals: to reduce the *time* and the *cost* in the assignment of tasks. Throughout this paper we use the notations listed in Table 1. To measure each objective we need to formulate each one precisely. We follow therefore the concept of *Work Breakdown Scheduling* (WBS) for all the incoming tasks that can be decomposed into subtasks (Sarro et al., 2017). Each task is associated with a node in a TDG and is assigned a *starting time* and an *ending time*. The starting time of each task is the maximum of the ending times of the tasks that it depends on. If we consider *assigned(ST_{ij})* as the developer assigned to subtask ST_{ij}, and task $T_i = \{ST_{ij} | 0 < i < n, 0 < j < m\}$ where n,m are respectively the number of tasks and the number of subtasks for the task at hand, then:

$$Time(T_i) = \sum_{j=1}^{m} executionTime(ST_{ij}) \tag{1}$$

$$\begin{aligned} executionTime(ST_{ij}) = \\ effort(ST_{ij})/productivity(assigned(ST_{ij}), ST_{i,j}) \end{aligned} \tag{2}$$

where *effort(ST_{ij})* is the estimated effort for ST_{ij}, and *productivity(assigned(ST_{ij}), ST_{ij})* gives an estimation of the productivity of developer *assigned(ST_{ij})* for subtask ST_{ij}. A major challenge is to calculate developer productivity, and to estimate the effort that a developer will require to complete a task. For evaluating our model, we will use the same data on effort and developer productivity used to assess KRRGZ, as provided by Karim et al. (2016). Investigating developer productivity is out of the scope of this paper, but more information on this topic can be found in the research literature (Sarro et al., 2016; Kocaguneli et al., 2012).

regularCost(ST_{ij}) is formulated as a function that depends on the *Time* and the *Hourly wage* of developers, *Wage(D_i)*,[4]

$$regularCost(T_i) = \sum_{j=1}^{m} Time(ST_{ij}) * Wage(assigned(ST_{ij})) \tag{3}$$

Furthermore, there is also another time, namely *duration*, that imposes extra costs on a software project in presence of an SLA. *duration* is defined as follows:

$$duration(T_i) = endTime(T_i) - startTime(T_i) \tag{4}$$

where *startTime(T_i)* will be the time when a change request is notified:

$$startTime(T_i) = arrivalTime(T_i) \tag{5}$$

and *endTime(T_i)* is:

$$endTime(T_i) = \max_{\forall j}(endTime(ST_{ij}))\forall ST_{ij} \in T_i \tag{6}$$

considering:

$$\begin{aligned} endTime(ST_{ij}) = startTime(ST_{ij}) \\ +executionTime(ST_{ij}) \end{aligned} \tag{7}$$

Eqs. (4) and (7) might be perceived similar in their definition. Actually, they only differ in terms of the function's arguments. In detail, in Eq. (4) we considered *tasks* as the function argument, while in Eq. (7) it is *subTasks* that are passed as argument. We have a similar situation with Eqs. (5) and (8); the former calculates the *startTime* for a task, while the latter computes the *startTime* for a subtask.

$$\begin{aligned} startTime(ST_{ij}) = arrivalTime(T_i)+ \\ max\{(endTime(ST_{ik})\forall ST_{ik} \in DEP(ST_{ij})), \\ availableTime(assigned(ST_{ij}))\} \end{aligned} \tag{8}$$

DEP(ST_{ik}) denotes all the subtasks that ST_{ik} is depended on, and *availableTime(assigned(k))* denotes the time at which developer *assigned(k)* is available to start working on subtask ST_{ik}. The delay time is defined as follow:

$$delayTime(T_i) = duration(T_i) - deadline(T_i) \tag{9}$$

where deadline is a variable that in industrial settings is usually given by multiplying *executionTime*(T_i) by a factor α (Wu et al., 2012).

Finally, *delayCost* can be formulated as follows:

$$\begin{aligned} delayCost(T_i) = delayTime(T_i) \\ *penaltyRate(class(T_i)) \end{aligned} \tag{10}$$

where *penaltyRate*(*class*(T_i)) returns a rate (i.e., the amount of penalty agreed with the customer) specified in an SLA. Thus, if *delayTime* grows, the cost of software maintenance increases.

At the end, the *total cost* and the *completion time* (See Eq. (4)) form the fitness function (with the two main objectives). The *totalCost* is a combination of *regular cost* and *delay cost*. Let *assgn* be the assignment vector, then:

$$\begin{aligned} totalCost(assgn) = regularCost(assgn)+ \\ delayCost(assgn) \end{aligned} \tag{11}$$

From considering the calculation of cost (see Eq. (3)), it might be thought that cost and time are not conflicting. However, there may be cases where a decrease in time not necessarily causes a decrease in cost. For instance, in a situation when a developer with higher experience (i.e., higher wage) requires less time than a less experienced developer (i.e., lower wage), completion time could be shorter but at the same time the cost could be higher.

[4] The hourly wage can be different for developers from different regions. Again, we use the data provided by Karim et al. (2016).

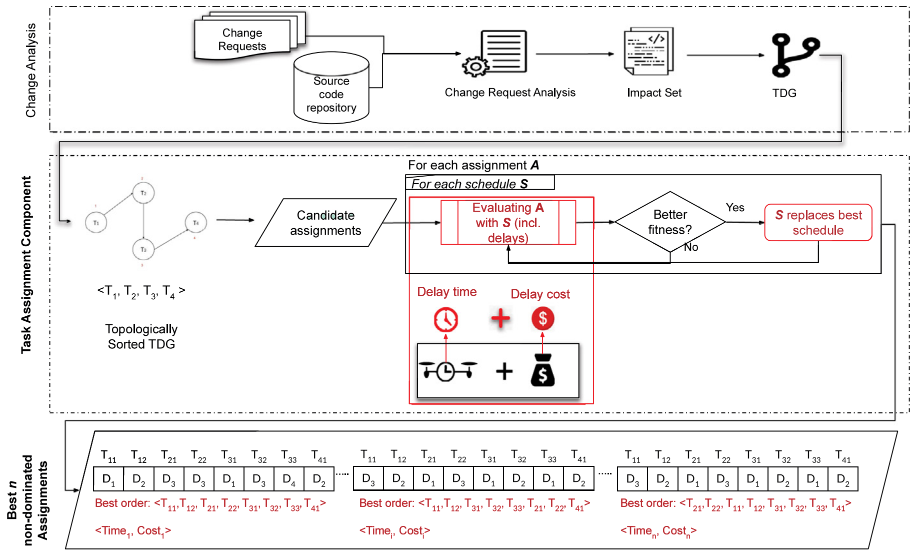


Fig. 1. The A-Z process of task assignment for several input change tasks. Our contributions, on top of KRRGZ, are highlighted in red. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

3.3. Pareto-optimality

As mentioned already, we consider two objectives: the *time* and the *cost* measured on orthogonal scales. A possibility would be to combine both objectives in a single one, using a weighted vector. The challenge is then the lack of deterministic values for the weighted vector. Thus, to deal with the bi-objective problem, we follow the principle of Pareto-optimality.

To clarify this decision, we will provide an example. Consider two solutions: TA_1 and TA_2. Solution TA_2 is said *to be dominated* by solution TA_1 when in at least one of the objectives, solution TA_1 has a better value than TA_2, while for the other objectives the values of TA_1 are still higher or equal to the ones of TA_2. In this manner, with solutions with two objectives, TA_1 strictly dominates TA_2 if:

$$Cost(TA_2) > Cost(TA_1) \text{ and } Time(TA_2) \geq Time(TA_1)$$
or
$$Cost(TA_2) \geq Cost(TA_1) \text{ and } Time(TA_2) > Time(TA_1)$$

During population evaluation, there is always a set of solutions where we cannot find relations like the ones above. If those solutions belong to the final generation, they create a set called Pareto-optimal or Pareto-front. That is, a set of non-dominated solutions which could each be a candidate for the final solution. In our case, the final solutions can be represented in a two-dimensional space, each one associated with an objective. Hence, the trade-off between time and cost can be observed in a two-dimensional plot (see Section 5.2.3).

3.4. Using search-based solutions

We have followed a search-based software engineering (SBSE) approach to provide solutions for the addressed problem. As the task assignment problem gets modeled as a bi-objective problem, we need to first justify the contrary relationship between the objectives. To do this, we need to follow several steps, including: present the solution, define the fitness function and explain the algorithms used for the computational search.

3.4.1. Computational search

Given a representation of the final solution (see Fig. 3), we need to find an evolutionary algorithm that, while sufficient for that discrete space, can provide better performance in terms of exploration and exploitation (Črepinšek et al., 2013). For those problems that deal with multi-objective trade-offs, studies so far have used metaheuristic approaches and subsequently computational search (Deb et al., 2002; Sarro et al., 2017; Karim et al., 2016; Sarro et al., 2016).

In this paper, we use NSGA-II as the evolutionary algorithm. NSGA-II performs well when the final solutions are non-dominated sets that need to be sorted for exploitation. It is commonly used in SBSE, specifically in those problems that are modeled as multi-objective (Karim et al., 2016; Sarro et al., 2016). This algorithm ends up with a non-dominated set that contains a number of solutions. We show how NSGA-II, which is extended by a domain-specific algorithm called *greedy local search (gls)* (we will thus call it $NSGA-II_{gls}$), can provide better exploration in terms of quality indicator metrics. *gls* gets run per solution evaluation and intends to modify the schedule vector.

Computational complexity has always been of major importance in studies in the area of evolutionary algorithms. In this study, applying the greedy local search imposes some extra running time. To address this shortcoming, a possibility would be to define a new parameter called k to cut the number of link evaluations (Ishibuchi and Murata, 1996). However, as we intend to enlarge the search space, such a restriction is better not to be taken into account.

KRRGZ also uses NSGA-II as the optimization algorithm. In this work, we have parameterized KRRGZ with the best settings reported in Karim et al. (2016) to compare our approach with the best results that KRRGZ can offer.

3.4.2. Solution encoding

In this subsection we aim at describing the encoding process of the candidate solutions. Each solution in our case comes with two vectors. The main vector is the encoded chromosome that determines who is assigned to which subtask. The second vector, the *schedule*, specifies the order in which those tasks should be performed by developers.

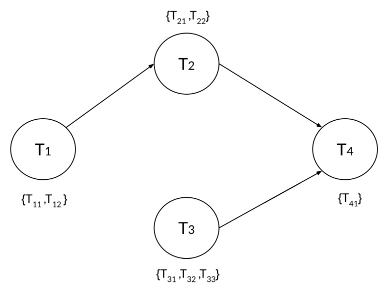


Fig. 2. A sample of task dependency graph (TDG) of the impact set $T = \{T_1, T_2, T_3, T_4\}$.

$$[T_1, T_2, T_3, T_4]$$

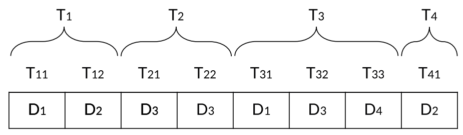


Fig. 3. A candidate chromosome for the TDG used as example.

The main vector. The solution to the addressed task assignment problem determines the developers assigned to each subtask. The chromosome is the main element of the encoded solution of size n, where n is the total number of subtasks. The tasks create together a task dependency graph called TDG. The TDG reveals all the dependencies among tasks. For example, if task T_1 is dependent on task T_2 then the *start time* of T_1 is at least greater than the *end time* of T_2. The TDG, on the other hand, is a Directed Acyclic Graph (DAG), so it helps to keep the order of performing tasks. We use a topological sorting algorithm (Cormen et al., 2009) to sort the prepared DAG and to obtain the order of tasks. Once the tasks are topologically sorted, the candidate solutions (i.e., the assignment of subtasks to developers) are encoded.

Assuming the developer set $D = \{D_1, D_2, D_3, D_4\}$ and the impact set $T = \{T_1, T_2, T_3, T_4\}$, Fig. 2 shows an example of a simple TDG which is used to encode feasible candidate solutions. The links in the TDG show the technical dependencies that exist among the artifacts of the tasks. These dependencies are sometimes labeled as work dependencies (Blincoe et al., 2015). Moreover, each task T_i comes with several subtasks (e.g., subtask ST_{ij} is the subtask j of task T_i).

For example, given the impact set T, the list of subtasks is listed as follow:

$T_1 = \{T_{11}, T_{12}\}$

$T_2 = \{T_{21}, T_{22}\}$

$T_3 = \{T_{31}, T_{32}, T_{33}\}$

$T_4 = \{T_{41}\}$

A candidate chromosome of the impact set T has a length of 8, i.e., the sum of all the subtasks of the members of set T. We have considered the topological sorting of TDG during chromosome encoding. A candidate topological sorting of the presented TDG graph could be as follow:

$[T_1, T_2, T_3, T_4]$

The encoded chromosome for this set has a representation as depicted in Fig. 3. The values for each gene in the chromosome are the *ids* of the developers (equivalent to D_i from Table 1). The initial population is of size n.

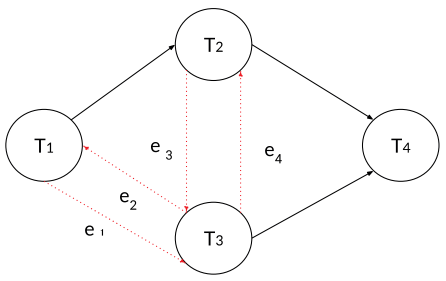


Fig. 4. Links e_1, e_2, e_3 and e_4 are the potential links that need to be iterated to create the schedule.

KRRGZ uses this encoding to represent the solution. In our method, however, the chromosome is extended by an encoded part called *schedule*.

The extended part. The schedule part, along with its associated chromosome, offers a particular order for developers to perform the tasks which is different from its chromosome alone.

The schedule part has to consider if there are tasks assigned to a developer that could be performed in parallel. In this case the potential links between the tasks play an important role in finding the best schedule. The potential links are the candidate links which can connect each pair of parallel nodes in the TDG. Given the TDG in Fig. 2, all potential links are highlighted with dashed lines in Fig. 4. In this case, e_1, e_2, e_3 and e_4 are all the potential links that need to be iterated to find the best schedule. The schedule has a fixed size of all potential links (let us denote it as m), which is 4 for the TDG graph shown in Fig. 2.

For each candidate solution, the assignment of parallel tasks to developers is analyzed to see whether there are intersections between parallel tasks.[5] So, given a candidate solution (e.g., the solution in Fig. 3), the following steps have to be taken to find the best schedule:

1. Pick a link from the potential links pool and check whether the tail nodes have developers in common (the tail nodes are already parallel).
2. Add the candidate link to the TDG and check if the link does not incur in a cycle in the graph.
3. Reevaluate the solution with a new order coming from a new topological sorting of the updated TDG.
4. Check if the solution with the new values dominates its former state.
5. In case of domination, turn the value of the potential link from 0 to 1 in the *schedule* vector; otherwise remove the link from the TDG.
6. In case the link pool is not empty, jump to step (1).

As it is obvious, during the schedule generation, if adding a link results in a cycle in the TDG, then that link is automatically discarded (see step 2). In this way, the TDG is kept *acyclic* to avoid loops. Fig. 5 shows a final solution in which the link e_1 incurs in finding a better optimum for the encoded solution.

It is important to notice that KRRGZ only considers one order of performing tasks by a developer, while we take into account different orders in our approach. This means that the candidate solutions, along with the corresponding schedules, allow to explore the parts of the search space which might have been overlooked by KRRGZ. If the TDG is sparse, our approach will potentially have more possibilities to find a better solution than KRRGZ. Being *sparse* means that dependencies among tasks do not occur very frequently.

[5] An intersection means a developer has been assigned two parallel tasks.

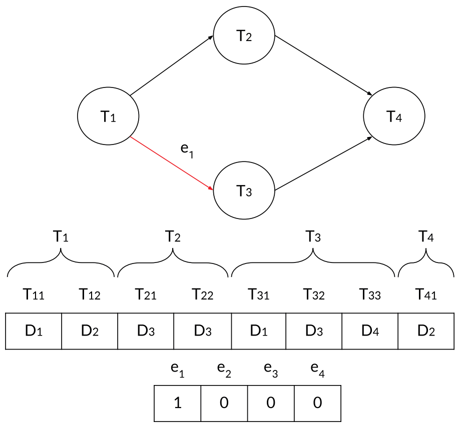


Fig. 5. A candidate final solution. Updated TDG with a new edge along with the associated chromosome and its schedule.

3.4.3. Fitness evaluation

Fitness evaluation is the process of computing the *time* and the *cost* for each candidate solution. It is done before turning into the next generation. The algorithm turns into the new generation by adding the new generated offspring to the current population. The over-sized population is then truncated to fit the predefined population size.

Algorithm 1 describes how the greedy local search gets involved during fitness evaluation. The output of the algorithm is a schedule associated to its corresponding solution. Line 1 in the algorithm denotes the schedule which is initialized with *0* for all its elements. The algorithm follows a greedy approach to pick the best links for scheduling. Therefore, if examining a particular link does not lead to a better optimum (i.e., a minimum), then the element with the corresponding index is kept untouched. There is also the possibility that none of the links gets selected. In this case all the values for the schedule are left to *0*.

The local search algorithm in this paper works on top of a *stochastic greedy local search* (SLS) (Dechter, 2003). For each assignment, the adapted local search evaluates different schedules. Given a fixed length of the schedule, the new schedules could be generated using *value flipping* (Dechter, 2003). Since each element in the schedule vector is valued with a *0* or a *1*, flipping means switching between these two values. Because of using an SLS-based approach, the algorithm advances to the next evaluation for a particular assignment if there is the possibility of having (i) valid schedules to be evaluated and (ii) the chance to find better schedules.

3.5. Other approaches

Our main aim in this research is to compare SD with KRRGZ, as the latter is supposed to be the state-of-the-art approach. However, for the sake of completeness, we will also compare SD and KRRGZ with two other approaches: (i) Random Search (RS) and (ii) Permutation Encoding (PE).

RS (Bergstra and Bengio, 2012) follows a uniform distribution for generating a solution; we will use it basically to make a sanity check of SD.

PE (Ronald, 1995) is a well-known type of solution encoding. It is a method that preserves the orders during the evaluation of a solution. Applying PE is a very efficient and simple way of modeling ordering solutions based on NSGA-II. The advantages and disadvantages of PE *vs.* SD can be summarized as follows:

1. Generating random solutions for PE is faster and easier due to the fact that a feasibility check is not required, while in SD the feasibility check is an obligation to avoid cycles in the dependency graph.
2. The search space is augmented by a factor of *n!* in PE, whereas SD expands the search space by $2^{n(n-1)}$ in the worst case. Usually the search space in SD will be much smaller than $2^{n(n-1)}$ because there might be dependencies, and some possibilities may become invalid and do not require to be evaluated. All in all, the feasibility check is usually more resource consuming for SD than for PE.

4. Case study design

In this section we discuss the organization of the empirical research study done to evaluate our approach.

4.1. Research questions

First, we present the research questions that this study aims to answer. As this study aims at minimizing the time and the cost of task assignment during bug fixing, the questions address the procedure and the efficiency of the proposed approach in comparison to the other approaches.

RQ0: What are the best NSGA-II parameter values for KRRGZ and SD?
RQ1: Does SD dominate KRRGZ in terms of quality indicator metrics?
RQ2: Does Permutation Encoding dominate KRRGZ and SD?
RQ3: How close to the Ground Truth is SD in terms of the accuracy?

4.2. Data-set

For examining the approaches and their applicability, we have to validate our approach within a real context. Therefore, we use the data-sets employed (and made available) by Karim et al. (2016). These data-sets have been preprocessed so that they are ready for being analyzed with our implementation. In particular, they contain milestones of two Eclipse IDE project components: JDT and Platform. For each of these components, we have access to the following information:

1. Available developers
2. The productivity of each developer
3. The hourly wage for each developer
4. The packages that have to be changed for each incoming bug report.

The dataset associates developers to a set of packages, with a given competence on them. Following KRRGZ's model, these competencies do not change over time. Thus, each developer has a value of productivity (in LOC/hr) for a particular package. Depending on this productivity, the developer will need more or less time to fix a bug.

Table 2 provides some relevant statistics of the projects. For each component the number of bugs, the number of dependencies and the time interval for each bug set are given. The dependencies are used to create the TDG, which allows to infer the order of the tasks. In particular, the packages that need to be changed for each bug are denoted — this is known as the *impact set*. An impact set includes all the artifacts that should be changed to accomplish a task (Ye et al., 2016). The artifacts define the subtasks that should be worked on by developers.

As shown in Table 2, the number of dependencies is very low in comparison to the number of tasks in each milestone. This holds for all the milestones of JDT and Platform. So, we can assume a scenario where the TDG of each milestone is *sparse*. This

Algorithm 1: Schedule-driven greedy local search applied to an individual solution.

```
Input: solution; TDG; schedule; sortedBugList (topologically sorted )
Output: modified schedule (includes links used for scheduling)
initialize schedule;
for $i \leftarrow 0$ to $sortedBugList.size - 2$ do
    for $j \leftarrow i + 1$ to $sortedBugList.size - 1$ do
        $numOfDevsInCommon \leftarrow getNumberOfDevsInCommon(solution, sortedBugList[i], sortedBugList[j])$;
        /* checks whether the number of developers in common between two tasks is more than one */
        if $numOfDevsInCommon > 0$ then
            $b \leftarrow isParallel(sortedBugList[i], sortedBugList[j])$ /* checks whether no path exists between these two nodes */
            if $b$ then
                $tempEdgeList \leftarrow candidateEdgesBetweenSortedBugList[i]\&sortedBugList[j]$;
                for edge e in tempEdgeList do
                    add edge e to TDG;
                    provide a topological sort of TDG;
                    evaluate(solution);
                    if new solution dominates former solution then
                        update schedule[indexOf(e)]
                    else
                        remove edge e from TDG
                    end
                end
            end
        end
        j++;
    end
    i++;
end
```

Table 2
List of JDT and Platform milestones and some of their characteristics.

Milestone	#Bugs	#Deps	Time period
JDT 3.1	9	0	2004-06-24 18:00–2005-06-26 14:09
JDT 3.1.1	86	0	2005-05-27 12:56–2005-09-27 12:08
JDT 3.1.2	30	0	2005-10-13 12:55–2006-01-09 05:42
JDT M1	73	0	2004-02-05 15:37–2004-08-12 05:29
JDT M2	82	0	2004-08-19 04:34–2005-09-16 13:41
JDT M3	142	0	2004-06-15 10:34–2004-03-11 18:26
JDT M4	244	1	2004-11-12 10:07–2004-12-16 03:51
JDT M5	259	2	2004-12-10 17:16–2005-02-18 10:29
JDT M6	172	1	2005-02-11 19:22–2005-04-01 11:36
Platform 3.0	20	0	2003-06-17 11:56–2004-08-10 18:20
Platform 3.1	18	0	2003-10-06 06:57–2004-12-03 14:39
Platform M2	51	0	2003-06-12 11:47–2004-07-17 06:19
Platform M3	101	5	2003-06-11 22:08–2003-08-27 11:32
Platform M4	179	2	2003-06-06 09:54–2003-10-09 04:44
Platform M5	109	2	2003-06-27 14:32–2003-11-20 13:35
Platform M6	88	0	2003-06-23 23:59–2003-12-18 14:41
Platform M7	156	0	2003-02-20 04:16–2004-02-12 14:05
Platform M8	124	1	2002-12-17 09:36–2004-02-13 01:48
Platform M9	99	0	2002-06-01 11:48–2004-05-21 14:56

is due to the nature of bug fixing, as bug fixes seldom require another bug to be fixed first.

Table 3 lists the priority levels assigned to bugs, and the associated penalty rates in case bugs are not fixed in time. As already mentioned, these priority levels are usually specified in SLAs. Tasks with high priorities will penalize more (i.e., cost more) than those with lower priorities. For example, suppose that task T_1 with priority *P1* has not been solved in time (i.e., the value of *delayTime* is DT_1, which is greater than zero). In this situation, the *delayCost* would be higher than if its priority would have been *P2* or *P3*.

4.3. Validity procedure

In this Section, we first introduce the metrics used to evaluate the approaches. Then, we present the methods used for the comparison. We also describe the Ground Truth (GT), and how we are going to compare SD with it. Finally, we discuss how we have configured NSGA-II in our implementation.

Table 3
Priority level and associated penalty rate (as specified in an SLA).

Priority list	
Priority level	Penalty rate
P1	0.9
P2	0.6
P3	0.3

4.3.1. Quality indicator metrics

Two approaches may be evaluated with the help of algorithmic metrics (i.e., the quality indicator metrics). These metrics show if using an approach results in solutions that dominate the ones by another approach. For the comparisons, we use four quality indicator metrics: (i) Hypervolume, (ii) Contribution, (iii) Generational Distance (Sarro et al., 2017) and (iv) Spacing (Du and Swamy, 2016).

Hypervolume. Given the objective space of the final solutions, Hypervolume refers to the portion of that space which is dominated by the non-dominated solutions of the underlying algorithm. If an approach provides non-dominated solutions that cover a larger part of the objective space relative to the reference point, then it will have a higher value of Hypervolume. Globally, if every solution provided by algorithm B is weakly dominated by at least one solution of algorithm A's Pareto-optimal set, then $Hypervolume_A > Hypervolume_B$ (Zitzler et al., 2003). Moreover, diversity and convergence, two of the main features for comparison, can be revealed by Hypervolume (Hadka and Reed, 2012). However, according to Zitzler et al. (2007), Hypervolume might suffer from bias — preventing us from recognizing the best possible solutions. Thus, using Hypervolume as the only quality indicator does not guarantee the convergence to the best solutions in a 2-dimensional objective space.

Contribution. To address the superiority of an algorithm in case one of its candidates already exists in the reference front, we use the Contribution quality indicator. Contribution is defined as the ratio of a specific algorithm's non-dominated solutions to all those already existing in the reference front set. So, if an algorithm generates more candidates in the reference front set, then that algorithm will result in more Contribution. The main drawback concerning Contribution is that algorithms that have good but few solutions score low (Sarro et al., 2017).

Generational distance. The Generational Distance quality indicator is computed as the average distance of all the solutions provided by an algorithm to those that belong to the reference front. This metric can be used as a representation of the measurement error: it indicates how far the obtained Pareto-optimal set member is from its nearest in the reference front (Van Veldhuizen and Lamont, 1998). The distance is computed in an n-dimensional space with n being the number of objectives. When Generational Distance is zero, all solutions are already in the reference front (i.e., the approaches are fully convergent). The value of Generational Distance is important when an approach has a Pareto-optimal set with few, but better non-dominated solutions. Generational Distance usually is intended for comparing the algorithm in terms of convergence (Hadka and Reed, 2012), so other metrics, such as Hypervolume, need to be taken into account to include the diversity provided by each algorithm.

Spacing. Spacing refers to the distribution of non-dominated solutions over the objective space (Du and Swamy, 2016). The non-dominated solutions belong to an individual algorithm. This metric is computed as the squared of the overall distance between the approximation of the members of a set and their nearest solution in the reference front divided by the number of solutions in the Pareto-front. For those algorithms that result in smaller spacing, final solutions are more requested. That is, the solutions are more uniformly distributed over the objective space.

4.3.2. Methods used in the comparison

We will use several methods to compare the quality indicator metrics.

Reference front. The reference front is the union of all non-dominated Pareto-optimal solutions (Sarro et al., 2017). In this study, we use the MOEA Framework (Hadka and Reed, 2012) to run the NSGA-II algorithm and then compute the quality indicator metrics. This framework provides an API (Application Programming Interface) for normalizing the objective values for each solution, which is required to be done in advance for the computation of the reference front. A normalization process, performed by the MOEA Framework, is required to avoid the scaling effects in the computation of the reference front (Sarro et al., 2017; Durillo and Nebro, 2011).

Statistical method. There is always a random variation in the output of evolutionary algorithms, due to their stochastic nature (Coello et al., 2007), that can be mitigated by repeating the experiment several times. Several authors propose to run experiments at least 30 times in such scenarios (Harman et al., 2012; Arcuri and Briand, 2014). Therefore, we need a statistical test that is able to justify the superiority of one approach over the other (in a paired test). Inspired by the research conducted by Sarro et al. we use the Wilcoxon statistical significance test to measure the difference in the performance of the employed approaches (Sarro et al., 2017). Wilcoxon is a non-parametric inferential statistical test where there is no assumption of the distribution of the sample under analysis (Arcuri and Briand, 2011). The Wilcoxon test has the property of being robust in treating data-sets with outliers (Kocaguneli et al., 2012). The final output of the test is the sum of the ranks with a particular *p-value*. The research literature recommends to use a *win–tie–loss* method to count the number of times an approach performs better against the other one (Sarro et al., 2017; Kocaguneli et al., 2012). We use this approach to compare the performance among the three examined algorithms.

Table 4
Range of effect sizes.

E_{AB}	Effect size
$0.6 \leq E_{AB} < 0.7$	Small
$0.7 \leq E_{AB} < 0.8$	Medium
$E_{AB} \geq 0.8$	Large

Moreover, usually a non-parametric test comes with a criterion called effect size (Arcuri and Briand, 2011). To precisely report the difference in performance of the heuristic approaches, we compute the effect size of the obtained statistical test using the Vargha–Delaney approach (Arcuri and Briand, 2011). The effect size allows to measure to what extent two approaches under comparison differ in terms of their quality indicator metrics. In this paper, we denote the effect size as E_{AB}, where A and B refer to the algorithms. The definition of the effect size is as follow:

$$E_{AB} = (R_1/n - (n+1)/2)/n \tag{12}$$

where R_1 is the rank sum of all the results of algorithm A and n is the total number of all items compared.

The effect size is usually reported on the basis of some level of confidence. The level of confidence specifies a threshold, α, where only effect sizes α >70% are accepted. Given that the effect size is a complement to the Wilcoxon test, bringing robustness to the test, we use the effect size rather than the number of *wins* to report statistical difference. So, in case of domination in terms of *wins*, the value of the effect size is the one that determines the significance of the difference.

We have used the ranges proposed in Sarro et al. (2017) as small, medium and large (see Table 4) to see how the three approaches perform when considering the effect size: small effect sizes mean a lose, medium mean a tie and large are a win. For instance, we label an effect-size as a win for A only if $E_{AB} \geq 0.80$.

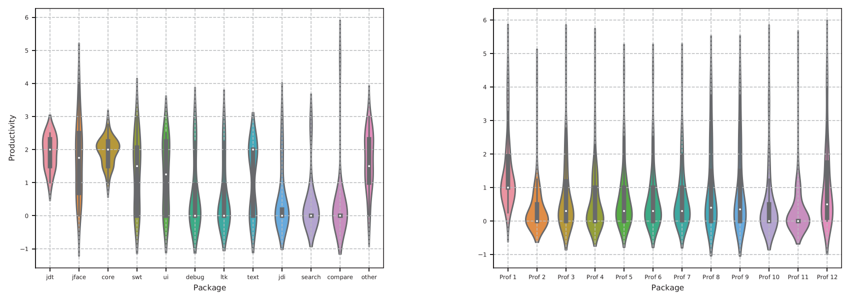
4.3.3. Comparison with the ground truth

Using algorithmic performance metrics is a way of evaluating the proposed approach. However, we need to examine how accurate our proposed approach is against the confirmed *Ground Truth (GT)*. As GT, we will use original assignments (that resulted in successful fixes). Given the GT, we will compute the accuracy of SD and KRRGZ.

A way to obtain the value of accuracy is using an *identity-based* comparison, which implies comparing the recommended developer with the actual fixer. A match is found *iff* developer A is the one who fixed the bug both in GT and the other approach.

A more robust computation of accuracy requires to consider more elements, as team members in our data-set may hold similar profiles. The algorithms could identify a similar developer as a good fit for performing a task. We refer to this approach as *profile-based*, as the assignment is based on the characteristics (i.e., the profile) of developers. In this scenario, if developer A has the same characteristics as developer B, assigning any of them is equally valid. Profile-based techniques have been used before in the research literature on automatic bug assignment (Tamrawi et al., 2011; Kagdi et al., 2012; Xia et al., 2016).

In our research, we have computed the profile-based accuracy using information on the productivity of developers in a given package. Hence, two developers who have identical productivity in a package will be interchangeable. Fig. 6 shows the distribution of developer productivity in JDT and Platform packages. As it can be observed from the violin plots, although some dispersion exists, developers with the same value of productivity in a given package can be found.



(a) JDT developers (b) Platform developers

Fig. 6. Distribution of productivity among developers by package.

Table 5
Configuration parameters used during the quantitative evaluation of the approaches. NFE stands for number of fitness evaluation.

Configuration parameters				
Approach	NFE	Crossover rate	Mutation rate	Population size
KRRGZ	250,000	0.90	0.05	500
Scheduling-Driven	250,000	0.90	0.01	500

4.4. Configuration of NSGA-II

This Section, together with Section 5.1, provides the details of the operators used in NSGA-II. We elaborate on how we have determined the best values for both SD and KRRGZ.

4.4.1. Genetic operators used in the experiment

When NSGA-II is applied to find a solution set for a multi-objective problem, it is important to use the right genetic operators in the exploration and exploitation. Following Sarro et al. (2017), we use (i) a crowding distance measure to sort solutions, and (ii) a binary tournament to select the members that should be conveyed to the next generation. For each solution, the *crowding distance* is given by the normalized distance to its closest adjacents along with each objective axis. The crowding distance measurement is preceded by sorting all the solutions based on the value of their objectives. In this way the solutions that ranked as the last ones result in an infinite distance in computation. All the objective values for each solution are normalized before being sorted.

To provide more diversity, NSGA-II operates on those solutions that have more crowding distance. Therefore, a binary tournament selection is used upon the crowding distance during each tournament to pick a solution for crossover. Binary tournament is a selection method commonly used by NSGA-II to select a member from a set of solutions in each round of the tournament. Alongside this operator, we use a one-point crossover function and a uniform mutation function to provide better exploration while the algorithm is running (Črepinšek et al., 2013). Many configurations can be used to tune the function operators; our aim is to evaluate them all to find the best one. To do so, we tried with the range of values presented in Table 5.

Solution convergence has always been a challenge for multi-objective problems which are solved through evolutionary methods (Durillo and Nebro, 2011). Convergence could be measured with Generational Distance (Hadka and Reed, 2012) (as presented in 4.3.1). Moreover, a fixed number of evaluations is set before running the algorithms. In this way, all the algorithms evaluate the fitness function equally. Given the same number of evaluations, the amount of computational effort is similar (Sarro et al., 2017). The different population sizes are set in the configurations as well. While the population size of each configuration might be different, the algorithms will end up after the same number of fitness evaluations (NFE) which is 250,000 times (Sarro et al., 2017).

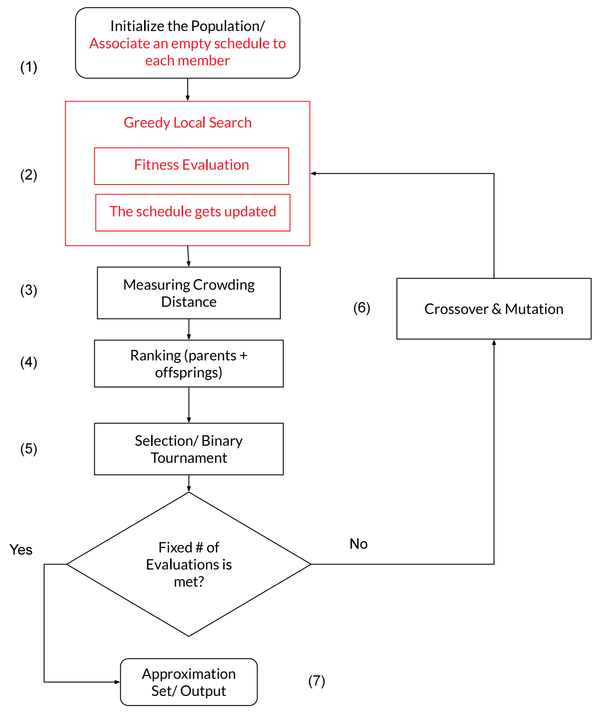

Initialize the Population/
Associate an empty schedule to each member
(1)
Greedy Local Search
Fitness Evaluation
The schedule gets updated
(2)
Measuring Crowding Distance
(3)
Ranking (parents + offsprings)
(4)
Selection/ Binary Tournament
(5)
Crossover & Mutation
(6)
Yes
Fixed # of Evaluations is met?
No
Approximation Set/ Output
(7)

Fig. 7. Flowchart of applying the SD approach. In red, our contribution on top of NSGA-II as used by KRRGZ. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

Fig. 7 offers a detailed overview of the steps followed. First, each population member is given an empty schedule. When either mutation or crossover are applied on each chromosome, the value of genes gets changed. As the schedule creation relies on the developer for each subtask, the altered chromosome needs to be reevaluated to prevent invalid schedules. This is because after the crossover and mutation operators are applied on the selected solution, the associated schedule might not be valid anymore, e.g., because of the existence of cycles in the TDG. The schedule is completely rewritten as the fitness evaluation gets applied on the mutated solution. We do this to ensure each schedule is always compatible with its associated candidate solution.

With the configuration from Table 5, both approaches (KRRGZ and SD) have their best value in terms of quality indicator metrics. The population size refers to the number of candidate solutions initialized when the algorithm starts. The number of fitness evaluations (NFE) is the same in all experiments. We use a one-point

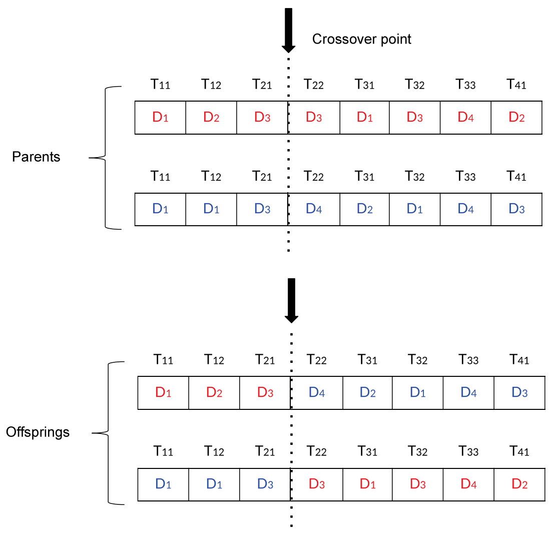


Fig. 8. One-point crossover is applied on a candidate solution on task assignment.

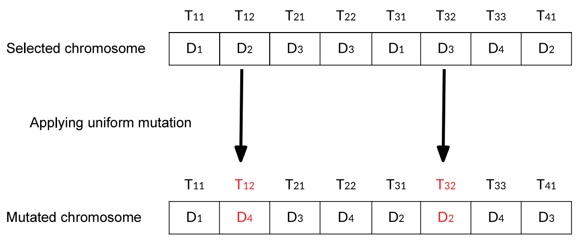


Fig. 9. A solution sample which is mutated by uniform mutation. The higher the mutation probability is, the more genes are mutated through this function.

crossover together with the uniform mutation as the genetic operators during the implementation. Thus, we need to first set the arguments for these functions. The arguments are the rate used when the functions are operating on top of the solutions. For the one-point crossover function, the rate determines the probability of choosing a point on both parent chromosomes to exchange the values of their genes. The rate for mutation limits the probability of selecting one of the possible values which is used for the offspring's genes (i.e., the variables of the candidate solution initialized by the number of developers). In NSGA-II, each candidate solution is considered as a chromosome which is formed by combining a fixed number of genes. The offspring here is equal to a chromosome in the implemented *NSGA-II*. The operators help to reform existing chromosomes in the current population aiming to better exploration.

Figs. 8 and 9 respectively show how the one-point crossover and uniform mutation are applied on the candidate solutions, particularly on the main vector.

5. Results

This Section aims to answer the research questions presented in Section 4.1. We start looking for the best parameter values of NSGA-II for the given approaches. Then, we examine if there is any simple and more realistic state of the art approach that solves the bi-objective task assignment problem in a more efficient way. We proceed then comparing SD and KRRGZ in terms of quality indicator metrics. Finally, we examine how close SD is to GT in terms of accuracy.

Table 6
Range of parameters that have been examined for genetic parameter tuning.

Operator	# of values examined	List of values
Crossover	5	[0.5, 0.6, 0.7, 0.8, 0.9]
Mutation	5	[0.01, 0.05, 0.10, 0.20, 0.30]

5.1. RQ0: What are the best NSGA-II parameter values for KRRGZ and SD?

Parameter tuning is an activity that usually is overlooked in search-based software investigations (Sarro et al., 2017). Even though we could have configured the experiment with the values used in KRRGZ, we need to look for the best parameters that fit our model best. We therefore pose this preliminary research question, and refer to it as RQ0.

We provide several settings associated to the genetic operators used in our study, and have further investigated them. To this end, we examine 5 values for each parameter. Table 6 lists the values selected in the tuning process.

Each pair of values for a particular operator is compared to one another. Table 7 shows the effect size of the comparisons among the range of values of crossover for JDT 3.1. Table 8, on the other hand, offers the comparisons among candidate values for Platform 3.1. As can be seen, the larger a value is, the better the effect size is. So, we select the maximum of the examined values, which is 0.9, as the crossover rate (the same values are used for KRRGZ, as in Karim et al., 2016).

The other parameter that needs to be tuned is mutation. To tune the mutation parameter, we again experimented with JDT 3.1 and Platform 3.1. Tables 9 and 10 report respectively on the effect size of the comparisons: the best performance belongs to the setting with lowest mutation, in our case 0.01.

These values of crossover and mutation, together with those for NFE, crossover and mutation rate and population size (already discussed in Table 5), offer the best setting for KRRGZ and SD. We have used them to run the experiments 30 times.

5.2. RQ1: Does SD dominate KRRGZ in terms of quality indicator metrics?

RQ1 examines if there is any improvement of SD over KRRGZ, and is the central part of our research. Therefore, we will first further elaborate on the motivating example presented in Section 1 to explain why we think SD might outperform KRRGZ (Section 5.2.1). Then, we will carry out several comparisons that show that SD offers better solutions. We will do so at the macro level (Section 5.2.2); using Pareto-fronts (Section 5.2.3); using boxplots to compare the medians and the dispersiveness of the quality indicators (Section 5.2.4); and performing a statistical test on 30 runs of the approaches (Section 5.2.5).

5.2.1. Elaborating on the motivating example

We have selected a specific example from the real dataset to clarify our approach. In particular, we have chosen three bugs from the JDT 3.1.1. Table 11 offers information on the bugs, such as their priority and an estimation of the effort to fix the bug in each of the packages that it affects. The source of the data used in this example comes from KRRGZ.[6]

Table 12, on the other side, lists three JDT developers and some information relative to them, in particular their productivity (for each of the packages affected by the bug, in lines of code per hour — LOC/h), and their hourly wage.

6 https://sites.google.com/site/mrkarim/data-sets (file JDTMilestone3.1.1.txt).

Table 7
Effect size when comparing different values of *crossover* in terms of quality indicator metrics for JDT 3.1.

	Hypervolume					Generational distance					Spacing					Contribution				
	0.5	0.6	0.7	0.8	0.9	0.5	0.6	0.7	0.8	0.9	0.5	0.6	0.7	0.8	0.9	0.5	0.6	0.7	0.8	0.9
0.5	–	0.51	0.50	0.46	0.42	–	0.52	0.34	0.51	0.30	–	0.35	0.43	0.48	0.39	–	0.51	0.45	0.43	0.33
0.6	0.49	–	0.49	0.51	0.42	0.48	–	0.52	0.44	0.44	0.65	–	0.48	0.60	0.54	0.49	–	0.42	0.46	0.33
0.7	0.50	0.51	–	0.53	0.51	0.66	0.48	–	0.63	0.55	0.57	0.52	–	0.47	0.54	0.55	0.58	–	0.60	0.50
0.8	0.54	0.49	0.47	–	0.56	0.49	0.56	0.37	–	0.54	0.52	0.40	0.53	–	0.52	0.57	0.54	0.40	–	0.65
0.9	0.58	0.58	0.49	0.44	–	0.70	0.56	0.45	0.46	–	0.61	0.46	0.46	0.48	–	0.67	0.67	0.50	0.35	–

Table 8
Effect size when comparing different values of *crossover* in terms of quality indicator metrics for Platform 3.1.

	Hypervolume					Generational distance					Spacing					Contribution				
	0.5	0.6	0.7	0.8	0.9	0.5	0.6	0.7	0.8	0.9	0.5	0.6	0.7	0.8	0.9	0.5	0.6	0.7	0.8	0.9
0.5	–	0.47	0.46	0.42	0.44	–	0.53	0.24	0.42	0.49	–	0.51	0.41	0.44	0.56	–	0.52	0.21	0.35	0.29
0.6	0.53	–	0.48	0.47	0.44	0.47	–	0.51	0.43	0.37	0.49	–	0.50	0.49	0.49	0.48	–	0.46	0.28	0.25
0.7	0.54	0.52	–	0.43	0.51	0.76	0.49	–	0.38	0.38	0.59	0.50	–	0.51	0.34	0.79	0.54	–	0.33	0.26
0.8	0.58	0.53	0.57	–	0.46	0.58	0.57	0.62	–	0.43	0.56	0.51	0.49	–	0.60	0.65	0.72	0.67	–	0.45
0.9	0.56	0.56	0.49	0.54	–	0.51	0.63	0.62	0.57	–	0.44	0.51	0.66	0.40	–	0.71	0.75	0.74	0.55	–

Table 9
Effect size when comparing different values of *mutation* in terms of quality indicator metric for JDT 3.1.

	Hypervolume					Generational distance					Spacing					Contribution				
	0.01	0.05	0.1	0.2	0.3	0.01	0.05	0.1	0.2	0.3	0.01	0.05	0.1	0.2	0.3	0.01	0.05	0.1	0.2	0.3
0.01	–	0.54	0.77	1	1	–	0.48	0.95	1	1	–	0.57	0.92	1	1	–	0.75	1	1	1
0.05	0.46	–	0.87	1	1	0.52	–	0.99	1	1	0.43	–	0.97	1	1	0.25	–	1	1	1
0.10	0.23	0.13	–	1	1	0.05	0.01	–	1	1	0.08	0.03	–	1	1	0	0	–	1	1
0.20	0	0	0	–	1	0	0	0	–	1	0	0	0	–	0.95	0	0	0	–	1
0.30	0	0	0	0	–	0	0	0	0	–	0	0	0	0.05	–	0	0	0	0	–

Table 10
Effect size when comparing different values of *mutation* in terms of quality indicator metrics for Platform 3.1.

	Hypervolume					Generational distance					Spacing					Contribution				
	0.01	0.05	0.1	0.2	0.3	0.01	0.05	0.1	0.2	0.3	0.01	0.05	0.10	0.20	0.30	0.01	0.05	0.1	0.2	0.3
0.01	–	0.95	1	1	1	–	1	1	1	1	–	0.91	1	1	0.93	–	1	1	1	1
0.05	0.05	–	1	1	1	0	–	1	1	1	0.09	–	0.91	1	1	0	–	1	1	1
0.10	0	0	–	1	1	0	0	–	1	1	0	0.09	–	0.92	0.88	0	0	–	1	1
0.20	0	0	0	–	0.99	0	0	0	–	1	0	0	0.08	–	0.70	0	0	0	–	1
0.30	0	0	0	0.01	–	0	0	0	0	–	0.07	0	0.12	0.30	–	0	0	0	0	–

Table 13
Comparison of SD against KRRGZ for all snapshots. It contains the number of bugs fixed for each snapshots, the number of developers involved in bug fixing activities depending on the approach used, the resulting time and cost, and the relative improvement in percentage achieved by SD over KRRGZ. To measure improvement, the final solution with minimum cost is selected for each approach.

		Bugs	Unique Devs		Time (in *hours*)		Cost		% Improvement	
			SD	KRRGZ	SD	KRRGZ	SD	KRRGZ	Time	Cost
JDT	3.1	9	7	8	81	83	11,401	11,403	**2.30**	**0.02**
	3.1.1	86	20	20	2285	1851	359,897	379,057	−**19.01**	**5.05**
	3.1.2	30	20	15	907	1435	85,127	85,577	**36.82**	**0.53**
	M1	73	19	20	1281	1335	153,853	154,178	**4.00**	**0.21**
	M2	82	20	20	1233	1039	250,752	264,933	−**15.68**	**5.35**
	M3	142	20	20	962	649	424,320	474,233	−**32.55**	**10.52**
	M4	244	20	20	851	1732	579,238	671,101	**50.86**	**13.69**
	M5	259	20	20	1383	2437	660,538	780,692	**43.23**	**15.39**
	M6	172	20	20	775	1125	419,551	458,848	**31.14**	**8.56**
Platform	3.0	20	29	49	588	464	159,981	212,354	−**21.17**	**24.66**
	3.1	18	19	42	1687	1541	153,348	171,240	−**8.65**	**10.45**
	M2	51	36	51	1429	1443	101,037	108,379	**1.01**	**6.77**
	M3	101	70	77	6361	8028	766,426	789,205	**20.76**	**2.89**
	M4	179	78	78	6782	5093	1,262,998	1,586,467	−**24.91**	**20.39**
	M5	109	78	78	5610	4755	807,259	997,332	−**15.24**	**19.06**
	M6	88	78	78	1486	1311	452,254	582,555	−**11.77**	**22.37**
	M7	156	78	78	6411	7976	1,705,809	2,043,133	**19.62**	**16.51**
	M8	124	78	78	1359	1974	1,320,192	1,757,590	**31.16**	**24.89**
	M9	99	76	78	3034	1873	1,113,984	1,531,975	−**38.29**	**27.28**

KRRGZ would offer the following final sample solution (*CandidateSolution* is given in Box I):

This means that B_1 would be addressed by D_1 in packages *jdt* and *ltk*, by D_2 in package *jface* and by D_3 in package *core*. The cost

$$CandidateSolution = \begin{array}{c} \\ \\ \end{array}\left[\begin{array}{cccccccccc} B_1 & & & & B_2 & & & B_3 & & \\ jdt & jface & core & ltk & jface & core & swt & jface & core & swt \\ D_1, & D_2, & D_3, & D_1, & D_3, & D_1, & D_2 & D_1, & D_3, & D_1 \end{array}\right]$$

Box I.

Table 11
Effort (in hours) needed to change a particular package (jdt, jface, core, swt, ltk) to fix bugs B_1, B_2 and B_3 in JDT 3.1.1. The priority column gives information on the importance of the bug.

Bug priority	Bug	Packages				
		jdt	jface	core	swt	ltk
P1	B_3 (#109,104)	0	51.16	38.88	13.30	0
P2	B_2 (#108,258)	0	40.80	3	16.20	0
P3	B_1 (#97,027)	79.97	2.61	4.36	0	2.61

Table 12
Productivity of developers per package (in LOC/h).

Developer	Packages					Wage
	jdt	jface	core	swt	ltk	
D_1	2	2	2	1	1.50	90
D_2	1.50	3.50	1.50	2.75	0.75	100
D_3	2	2	2.25	2	2.75	100.25

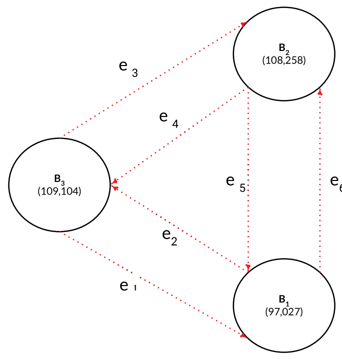


Fig. 10. Potential links that could result in different orders for developers when fixing bugs. In our case, the three bugs (B1 #97,027, B2 #108,258 and B3 #109,104) have no inter-dependencies.

and the time computed for this sample solution are: *total time = 179 h* and *total cost = 12,357*.

As already mentioned, our approach builds on top of KRRGZ, equipping it with the possibility to have a larger search space due to the inclusion of scheduling. Therefore, we build the directed acyclic graph (DAG) of the three bugs (see Fig. 10).

Given the DAG of the three bugs, several valid schedules are identified. An example schedule could be as follow:

$$Candidateschedule = \left[\begin{array}{cccccc} e_1 & e_2 & e_3 & e_4 & e_5 & e_6 \\ 0, & 0, & 1, & 0, & 1, & 0 \end{array}\right]$$

In this example, we can see which of the edges of the DAG are considered in the ordering. The value of 1 means the associated *edge* is included, whereas 0 means the absence of that edge. The candidate schedule has the property of dictating a particular order of performing tasks by the developers (in this case, [B_3, B_2, B_1]). With this new order, the time and the cost are: *Total time = 170 h* and *total cost = 11,637*. This means that with the new schedule we have achieved a 5% improvement in terms of time (9 h less) and almost a 6% improvement in cost (720 less).

The reason for the improvement lies in the fact that the rearrangement results in many more orders of performing the tasks. These new arrangements are evaluated for their time and cost. In our case, as (at least) one of these rearrangements is better than the solution given by KRRGZ, our approach is superior and bugs are addressed in less time and with less cost.

5.2.2. *Comparison of time and cost*

Table 13 compares the results of SD and KRRGZ for all the snapshots considered in our study. Given a snapshot, SD and KRRGZ are compared based on the time and the cost of the two best solutions for each approach, i.e., they are the best assignments by SD and KRRGZ selected from their final Pareto-fronts. The improvement in terms of time and cost in relative terms (columns *Time* and *Cost* Improvement) are also shown in Table 13. Non-negative improvements are found when SD performs better than KRRGZ. On the other hand, the negative percentages denote the superiority of KRRGZ over SD.

As can be seen, there are snapshots where SD is better than KRRGZ in time and cost. There are, however, some snapshots where SD is beaten by KRRGZ in one of the two objectives. This happens when SD and KRRGZ are both Pareto-optimal, so they offer solutions that equally optimize time and cost. In those cases, project managers can define a priority between the two objectives. For instance, in Table 13 we have opted uniformly for the solution for SD that is of lower cost — resulting in more time being invested. Depending on the available solutions, project managers could choose instead to minimize time. It should be noted that as SD builds on top of KRRGZ, a project manager could easily go for a solution offered by KRRGZ.

5.2.3. *Visualizing Pareto-fronts*

We have randomly selected 8 milestones (4 from JDT and 4 from Platform) and offer the Pareto-fronts visually in a diagram. Random Search (RS) is included in this comparison as a sanity check. Figs. 11 and 12 show the scatter plots with the results for the selected milestones from JDT and Platform, respectively. Particularly, Fig. 11 demonstrates how the final Pareto-optimal sets for JDT dominate those of KRRGZ. The same occurs for the milestones from Platform. Fig. 12 shows that in 3 out of 4 provided Pareto-front plots, SD dominates KRRGZ and RS. Throughout all the cases it never happens that SD gets dominated by KRRGZ, but there are some cases, like the one in Fig. 12(b), where both SD and KRRGZ end up with almost the same Pareto-front.

5.2.4. *Performance comparison using boxplots*

In addition to the Pareto-fronts, the approaches can be compared with respect to the median and the dispersiveness of the evaluated quality indicators. To do so, we use boxplot diagrams. The comparison between SD and KRRGZ has been performed in terms of the 4 quality indicator metrics. The plots provided for each data-set make the comparisons over all the milestones of a particular project. So, the plots shown in Fig. 13 represent how the final non-dominated solutions for JDT dominate the other

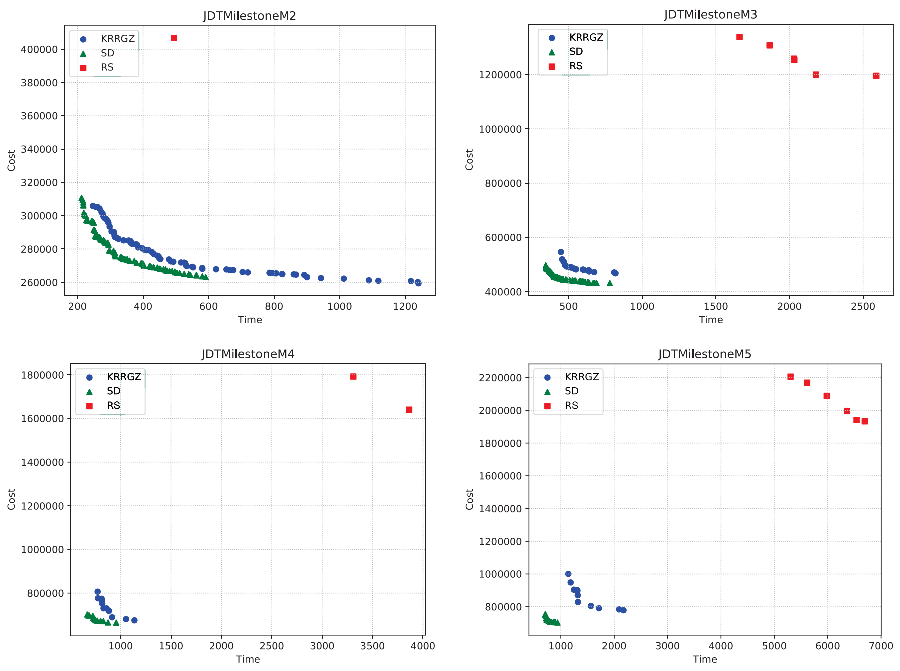


Fig. 11. Final Pareto-fronts for JDT milestones. (a), (b), (c) and (d) respectively depict JDT M2, M3, M4 and M5. Being closer to the origin shows better performance.

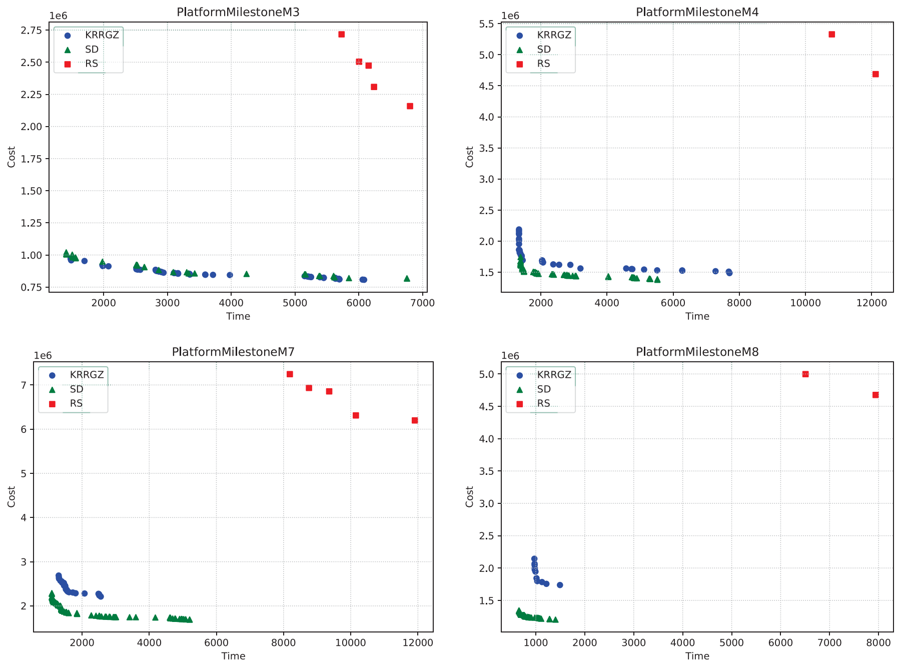


Fig. 12. Final Pareto-fronts for Platform milestones. Respectively, (a), (b), (c) and (d) depict Platform M3, M4, M7 and M8. Being closer to the origin shows better performance.

ones. We also show the comparison of the quality indicators for Platform in Fig. 14.

For all the plots shown in Figs. 13 and 14, superiority of I_{HV} and I_C means having higher values, while for I_{GD} and I_S the lower the better. In Fig. 13(a) and (c), where I_{HV} and I_C are compared for JDT milestones, SD comes up with higher median values and less outliers. The same happens for Platform, as it can be seen in Fig. 14(a) and (c). In addition, Figs. 13(b) and 14(b) show how SD dominates KRRGZ in terms of median of I_{GD} for JDT and Platform, respectively.

Moreover, Table 14 lists the mean value and standard deviation (after 30 runs) of the quality indicator metrics for each milestone of the two projects under study. Significant differences in performance are highlighted in the table, showing that SD performs better than KRRGZ in terms of all four quality indicators, but in particular for Hypervolume and Contribution.

5.2.5. *Statistical comparison*

In this subsection, we report on the output of applying the paired non-parametric statistical test, i.e., the Wilcoxon signed rank test. Let us denote the average of 30 runs (for the four quality indicator metrics) of SD per data-set as SD_i, for KRRGZ as $KRRGZ_i$

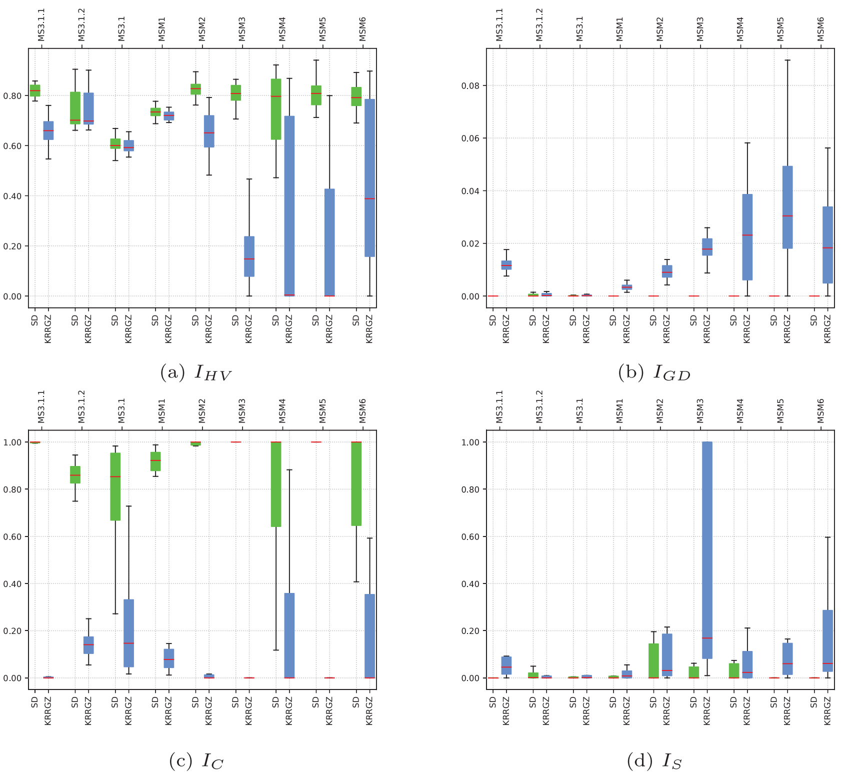


Fig. 13. Boxplots comparing the performance of quality indicators for JDT milestones. (a), (b), (c) and (d) are respectively the boxplots for Hypervolume, Generational Distance, Contribution and Spacing. Green boxplots correspond to SD, while blue ones to KRRGZ; in both, the median is given by a red horizontal line. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

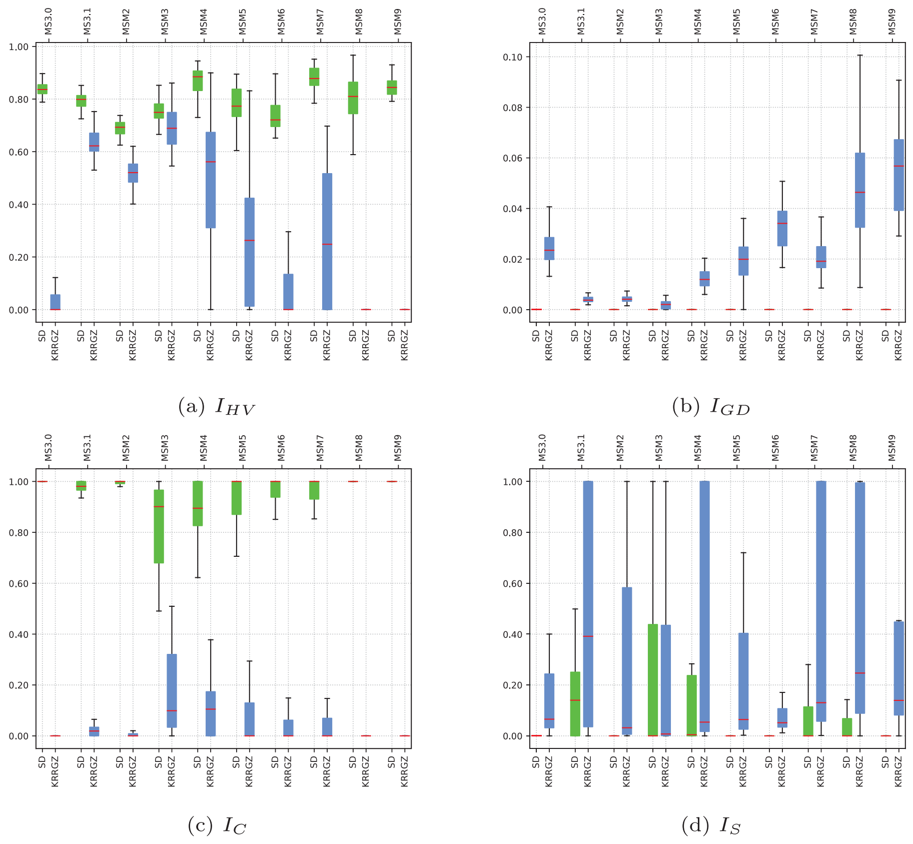


Fig. 14. Boxplots comparing the performance of quality indicators for Platform milestones. (a), (b), (c) and (d) are respectively the boxplots for Hypervolume, Generational distance, Contribution and Spacing. Green boxplots correspond to SD, while blue ones to KRRGZ; in both, the median is given by a red horizontal line. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

Table 14
Mean and standard deviation of the quality indicators for all JDT and Platform milestones. The entries have been normalized. The cell background color scheme corresponds to quintiles; the darker the better. Note that for Hypervolume and Contribution, the more the better; while for Generational Distance and Spacing, the merrier the better.

		Hypervolume			Generational distance			Spacing			Contribution		
		SD	KRRGZ	RS	SD	KRRGZ	RS	SD	KRRGZ	RS	SD	KRRGZ	RS
JDT	MS3.1.1	0.82; 0.03	0.66; 0.06	0.00; 0.00	0.00; 0.00	0.01; 0.01	1.00; 0.00	0.06; 0.16	0.16; 0.30	0.90; 0.31	0.99; 0.03	0.01; 0.03	0.00; 0.00
	MS3.1.2	0.75; 0.07	0.75; 0.07	0.01; 0.05	0.00; 0.01	0.00; 0.00	1.00; 0.00	0.03; 0.05	0.01; 0.03	1.00; 0.00	0.85; 0.07	0.15; 0.07	0.00; 0.00
	MS3.1	0.61; 0.03	0.60; 0.03	0.00; 0.00	0.00; 0.00	0.00; 0.00	1.00; 0.00	0.19; 0.37	0.22; 0.41	0.77; 0.43	0.75; 0.27	0.25; 0.27	0.00; 0.00
	MSM1	0.72; 0.05	0.72; 0.02	0.00; 0.00	0.00; 0.00	0.00; 0.00	1.00; 0.00	0.01; 0.02	0.02; 0.02	1.00; 0.00	0.87; 0.16	0.13; 0.16	0.00; 0.00
	MSM2	0.82; 0.03	0.65; 0.08	0.00; 0.00	0.00; 0.00	0.01; 0.00	1.00; 0.00	0.12; 0.22	0.26; 0.42	0.77; 0.43	0.99; 0.02	0.01; 0.02	0.00; 0.00
	MSM3	0.81; 0.04	0.16; 0.11	0.00; 0.00	0.00; 0.00	0.02; 0.01	1.00; 0.00	0.08; 0.17	0.43; 0.43	0.67; 0.47	1.00; 0.00	0.00; 0.00	0.00; 0.00
	MSM4	0.67; 0.30	0.28; 0.36	0.00; 0.00	0.04; 0.18	0.02; 0.02	1.00; 0.02	0.13; 0.29	0.13; 0.26	0.90; 0.29	0.76; 0.40	0.24; 0.40	0.00; 0.00
	MSM5	0.76; 0.19	0.20; 0.29	0.00; 0.00	0.03; 0.16	0.03; 0.02	1.00; 0.00	0.07; 0.21	0.21; 0.34	0.89; 0.29	0.85; 0.33	0.15; 0.33	0.00; 0.00
	MSM6	0.78; 0.09	0.44; 0.31	0.00; 0.00	0.00; 0.00	0.02; 0.02	1.00; 0.00	0.09; 0.26	0.21; 0.31	0.90; 0.30	0.80; 0.33	0.20; 0.33	0.00; 0.00
Platform	MS3.0	0.84; 0.05	0.06; 0.12	0.00; 0.00	0.00; 0.00	0.02; 0.01	1.00; 0.00	0.07; 0.20	0.26; 0.37	0.80; 0.40	1.00; 0.00	0.00; 0.00	0.00; 0.00
	MS3.1	0.79; 0.04	0.62; 0.09	0.00; 0.00	0.00; 0.00	0.00; 0.00	1.00; 0.00	0.24; 0.33	0.50; 0.46	0.47; 0.51	0.98; 0.02	0.02; 0.02	0.00; 0.00
	MSM2	0.69; 0.03	0.52; 0.05	0.00; 0.00	0.00; 0.00	0.00; 0.00	1.00; 0.00	0.12; 0.24	0.28; 0.43	0.75; 0.43	0.99; 0.01	0.01; 0.01	0.00; 0.00
	MSM3	0.76; 0.05	0.69; 0.08	0.00; 0.00	0.00; 0.00	0.00; 0.00	1.00; 0.00	0.26; 0.40	0.26; 0.40	0.70; 0.47	0.76; 0.30	0.24; 0.30	0.00; 0.00
	MSM4	0.86; 0.06	0.48; 0.25	0.00; 0.00	0.00; 0.00	0.01; 0.01	1.00; 0.00	0.17; 0.28	0.37; 0.45	0.67; 0.48	0.86; 0.21	0.14; 0.21	0.00; 0.00
	MSM5	0.75; 0.16	0.26; 0.26	0.00; 0.00	0.00; 0.00	0.02; 0.01	1.00; 0.00	0.07; 0.20	0.28; 0.40	0.83; 0.38	0.89; 0.25	0.11; 0.25	0.00; 0.00
	MSM6	0.72; 0.15	0.09; 0.16	0.00; 0.00	0.00; 0.00	0.04; 0.02	1.00; 0.00	0.01; 0.05	0.15; 0.25	0.93; 0.25	0.92; 0.19	0.08; 0.19	0.00; 0.00
	MSM7	0.88; 0.04	0.26; 0.24	0.00; 0.00	0.00; 0.00	0.02; 0.01	1.00; 0.00	0.07; 0.13	0.39; 0.43	0.72; 0.45	0.95; 0.08	0.05; 0.08	0.00; 0.00
	MSM8	0.79; 0.11	0.02; 0.07	0.00; 0.00	0.00; 0.00	0.05; 0.02	1.00; 0.00	0.12; 0.25	0.42; 0.41	0.73; 0.43	1.00; 0.00	0.00; 0.00	0.00; 0.00
	MSM9	0.84; 0.05	0.00; 0.00	0.00; 0.00	0.00; 0.00	0.06; 0.02	1.00; 0.00	0.03; 0.08	0.35; 0.38	0.81; 0.38	1.00; 0.00	0.00; 0.00	0.00; 0.00

and for RS as RS_i. The Wilcoxon test examines how pairs of these approaches are statistically significantly different. $Z_i = SD_i - KRRGZi$ ($i \in$ run number) describes the difference between the mean values of *SD* and *KRRGZ* for each data-set (this relation also stands for all the pairs of approaches). i in Z_i denotes the result of run number i.

The results of the Wilcoxon and the effect size computation are finally summarized as win–tie–loss stats shown through stack charts. We will use two scenarios to obtain the number of wins, ties and loses: (i) the output of the Wilcoxon test together with its *p*-value, and (ii) a threshold of the effect size.

Using the Wilcoxon test: Charts shown in Fig. 15 summarize the win–tie–loss statistics considering only Wilcoxon sum rank tests with a *p*-value of *0.05*. Given a particular approach, a fixed number of comparisons is made with the two other approaches: for JDT this number is 72, while for Platform it is 80. For JDT, SD is the winner in 68 (71%), and KRRGZ wins 28 (29%) out of 96 cases. For Platform, SD wins 79 (74%), and KRRGZ 28 (26%) out of 107 cases. It is worth mentioning that RS does not win in any situation, and only gets some equal performances for some comparisons.

Effect size threshold: The stack charts in Fig. 16 summarize the win–tie–loss in terms of effect sizes. To get these results, we ran Vargha–Delaney upon the values of the quality indicators. As shown in Fig. 16(a), SD gets 54 out of 82 (66%) wins for the JDT project. On the other hand, the number of wins for KRRGZ is 28 out of 82 (34%). Fig. 16(b) visualizes the number of wins that each approach obtains during the comparison for the Platform project. The numbers for SD and KRRGZ are 71 out of 99 (72%) and 28 out of 99 (28%), respectively. The results with large effect-sizes show that SD is significantly better than KRRGZ in terms of quality indicators.

5.3. RQ2: Does permutation encoding dominate KRRGZ and SD?

To answer RQ2, we examine the efficiency of SD (and KRRGZ) compared to Permutation Encoding (PE). We have therefore performed experiments to find the optimal task assignments in four milestones of JDT (3.1.2 and M1) and Platform (3.1 and M2). Table 15 offers some descriptive information, such as the number of bugs and the number of unique developers (*Unique Devs*) that fixed bugs for each approach. In addition, Table 15 presents as well the results that allow us to compare SD, PE and KRRGZ for time and cost. We have chosen those solutions that hold minimum cost in the related Pareto-front. That is why there are some cases where PE outperforms SD in terms of time. This happens when SD and PE are both Pareto-optimal, so they offer solutions that equally optimize time and cost. Even though, we can see that SD clearly outperforms PE in half of the studied cases. For the other cases where PE beats SD in terms of minimum time (but not cost), we can find out which one dominates the other by visualizing the Pareto-fronts.

Figs. 17 and 18 compare the Pareto-fronts of the implementations of SD, KRRGZ and PE for two milestones of JDT and Platform, respectively. It can be seen that SD performs in general better than PE, as being closer to the origin in indicative for better performance. This is clearly the case for JDT M1, Platform 3.1 and Platform M2. Only in the case of JDT 3.1.2, the performance of PE is better than SD for a few, low values of time.

We use box-plots to further compare the results using the four quality indicator metrics. Figs. 19 and 20 compare the results for the selected milestones after 30 runs. As it can be seen, SD outperforms PE in all cases.

It is worth mentioning that PE, while dominated by SD, performs very similar to KRRGZ — the Pareto-fronts in Figs. 17 and 18 offer evidence in this regard.

5.4. RQ3: How close to the ground truth is SD in terms of the accuracy?

This section is dedicated to answer RQ3, evaluating the proposed approach in term of its accuracy, particularly in comparison with GT. To this end, we compared the assignments made with our approach (and KRRGZ) with those in the original data-set. As discussed earlier (see Section 4.3.3), accuracy will be reported in two different ways: *Identity-based* and *Profile-based*.

As it can be observed from Table 16, there is not much conformance between the assignments made by SD and KRRGZ and the real assignments in the identity-based comparison with values that range mainly from 0.0 to 0.2. Moreover, considering the profile-based accuracy, SD and KRRGZ offer an acceptable level of accuracy compared to GT. The values for this type of accuracy range from 0.51 to 0.93, with most of them in the 0.80's for the SD approach.

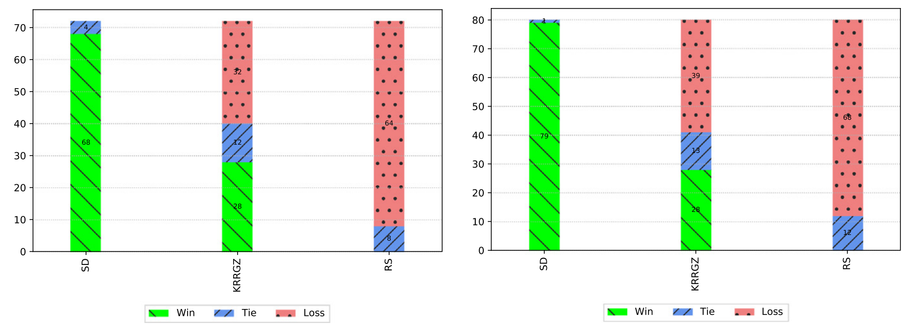
All in all, Table 16 also illustrates that the assignment used in GT is not the best one. If we look at the values of time and cost, only in three (JDT 3.1.2, JDT M1, Platform M2) they are better for GT than for SD and KRRGZ. In these cases, nonetheless, the cost is significantly higher. We conjecture that this is because the most experienced (but as well the ones with the highest salaries) were the ones who fixed the bugs in those milestones. In general, we can proclaim that the GT does not offer an optimal solution, at least in terms of the time and the cost. Both KRRGZ and SD outperform GT clearly, even achieving relatively high values of profile-based accuracy. SD in this regard offers both higher values of profile-based accuracy than KRRGZ compared to GT and at the same time (as shown with RQ1) offers better time-cost solutions than KRRGZ.

6. Discussion, limitations and threats to validity

As we have built our approach on top of Karim et al.'s approach (Karim et al., 2016), our work *inherits* most of the limitations from that work. KRRGZ and SD are based on a set of assumptions, being the most relevant: (i) developers do not work on more than one bug simultaneously, (ii) the sets of bug reports, developers, and competencies are fixed, (iii) the effort reported for bug fixing, and the resulting developer productivity is accurate, (iv) the complexity of bugs is not taken into consideration, and (v) the competency of developers is proportional to their productivity. Given that the assumptions are the same, they affect both approaches equally, resulting in a *fair* comparison between the two. It is against the other approaches where these assumptions might have an effect.

On the other hand, it should be noted that our approach does not lose some of the benefits of Karim et al.'s solution. For instance, KRRGZ addressed the problem of developers being assigned too many tasks, in order to prevent overloading them. We have examined if this is still true with SD. Fig. 21 shows several violin plots with developer participation in four randomly selected milestones of JDT and Platform. As it can be observed, the number of tasks is more or less evenly distributed among developers, in a similar fashion as with KRRGZ. In short, while SD optimizes better for time and cost compared to KRRGZ, it still avoids overloading developers.

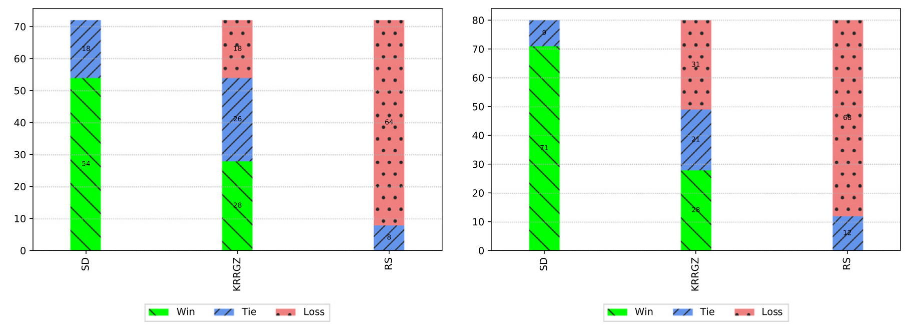
There may be other aspects (in addition to the time and the cost) that could influence the decision of a manager when selecting a particular solution. For instance, team leaders may prefer to select an assignment plan that involves a higher number of developers, so that the knowledge of the code base is more widely spread among its team. For this purpose, we inspired ourselves again in some criteria used by Karim et al. (2016), resulting in following preferences (referred as problem-specific metrics):



(a) JDT

(b) Platform

Fig. 15. Win–tie–loss summary considering Wilcoxon rank sum test among the three approaches. (a) and (b) respectively show the stats of win–tie–loss for JDT and Platform.



(a) JDT

(b) Platform

Fig. 16. Win–tie–loss summary considering the effect size among all three approaches. (a) and (b) respectively show the stats of win–tie–loss for JDT and Platform, respectively.

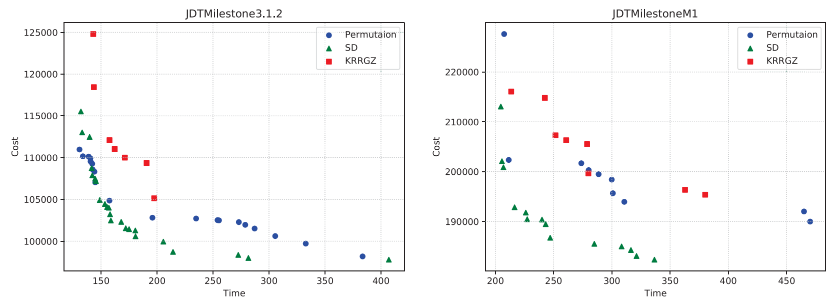


Fig. 17. Pareto-fronts for KRRGZ, SD and PE for JDT milestones, in particular (a) for JDT 3.1.2, and (b) for JDT M1.

Table 15
Comparison between SD, KRRGZ and PE in terms of *time* and *cost*. solutions with minimum cost have been chosen.

Snapshot	Bugs	Unique devs			Time (in *hours*)			Cost			% Superiority (SD over PE)	
		SD	PE	KRRGZ	SD	PE	KRRGZ	SD	PE	KRRGZ	Time	Cost
JDT 3.1.2	30	18	20	18	407	384	197	97,787	98,174	105,130	**−6.12**	**0.42**
JDT M1	73	19	20	20	337	470	380	182,326	189,970	195,371	**39.72**	**4.19**
Platform 3.1	18	38	43	42	570	620	455	218,610	304,044	284,270	**8.14**	**28.09**
Platform M2	51	42	54	51	1147	1075	724	147,284	172,083	196,895	**−6.34**	**14.41**

(i) number of developers involved; (ii) mean tasks per developer; and (iii) number of developers not assigned to any bug. Table 17 shows values for these problem-specific metrics for a subset of the scenarios from the Pareto-front (i.e., they do not dominate the others regarding time and cost) for Platform *M2*. As a result, managers would be able to select a solution not only based on the time and the cost, but also on these other characteristics. So, a manager wanting to maximize the number of developers involved could choose a solution with the highest number of developers (42, as in row #1 or in row #10), knowing that these solutions are optimized as well for their time and cost.

We have also examined if the number of dependencies and the size of the bug fixing team affects the performance of SD in relation to KRRGZ. As expected, with TDGs becoming less sparse, the performance of SD tends to be that of KRRGZ. The size of the bug fixing team does, however, not affect the dominance of

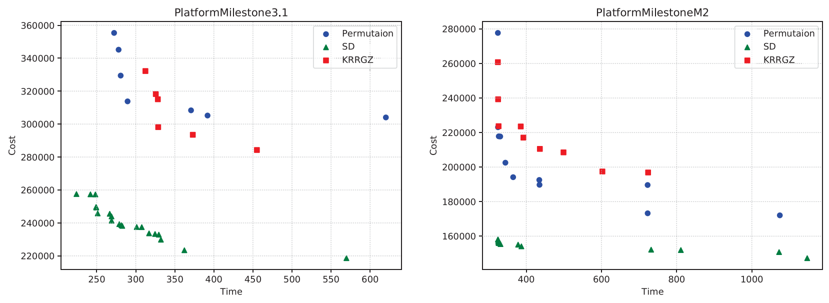
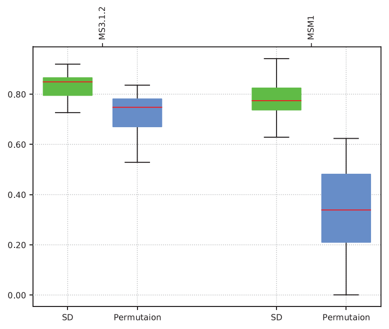
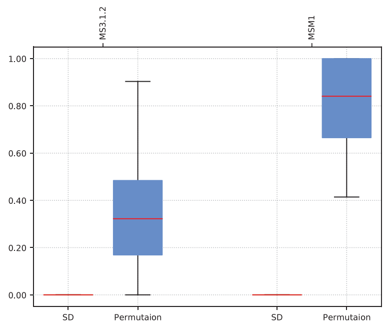


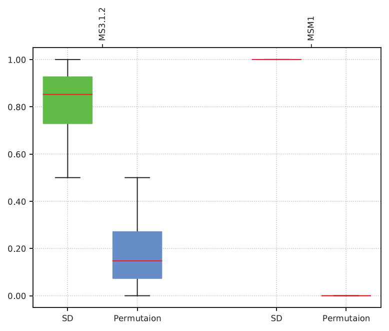
Fig. 18. Pareto-fronts for KRRGZ, SD and PE for Platform milestones, in particular (a) for Platform 3.1, and (b) Platform M2.



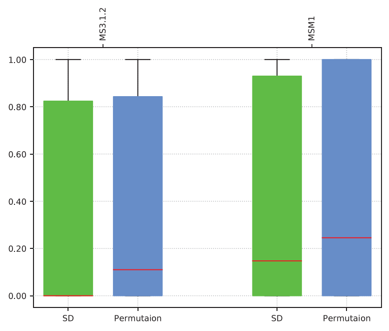
(a) I_{HV}



(b) I_{GD}



(c) I_C



(d) I_S

Fig. 19. Boxplots comparing the performance of the quality indicators for JDT milestones. (a), (b), (c) and (d) are the boxplots for Hypervolume, Generational Distance, Contribution and Spacing, respectively. Green boxplots correspond to SD, while blue ones to PE; in both, the median is given by a red horizontal line. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

SD over KRRGZ. When the teams are smaller both approaches perform worse, but SD outperforms KRRGZ consistently.

Researchers who want to further investigate with our data or address the limitations can benefit from Karim et al.'s (2016) data, as we have done. In addition, the scripts used in our study can be accessed in an on-line replication package[7]

In the following subsections, we address the three main threats concerning the validity of the introduced approach, specifically, *Construct*, *Internal* and *External* validity (Runeson and Höst, 2009).

6.1. Construct validity

Construct validity addresses the doubts surrounding the methodology used for empirical research. To address the construct validity, we have followed well-known methods employed in literature (Anvik and Murphy, 2011; Karim et al., 2016; Sarro et al., 2017) for empirical research. To make a comparative study, we have used the same data-sets supplied and used by Karim et al. to evaluate KRRGZ (Karim et al., 2016). There are some concerns about the data-sets used in KRRGZ that apply to our case, too (Karim et al., 2016).

6.2. Internal validity

Internal validity points to the potential bias that might be incurred by the methods used to solve the problem. Here, the stochastic nature of the evolutionary approach used as the solution imposes some biases. To tackle the internal threats we followed state of the art approaches (Sarro et al., 2017), and took four different quality indicators. To get rid of the bias in the experimental results, we run each experiment 30 times with the fixed configuration (Harman et al., 2012) and took the average values of the quality indicators. The Wilcoxon test together with effect size are then applied to statistically compare the output of both approaches. In this way, the results can be considered to have an acceptable level of safety.

[7] The on-line replication packages is publicly available at https://github.com/anonauthors/SCM_TA.

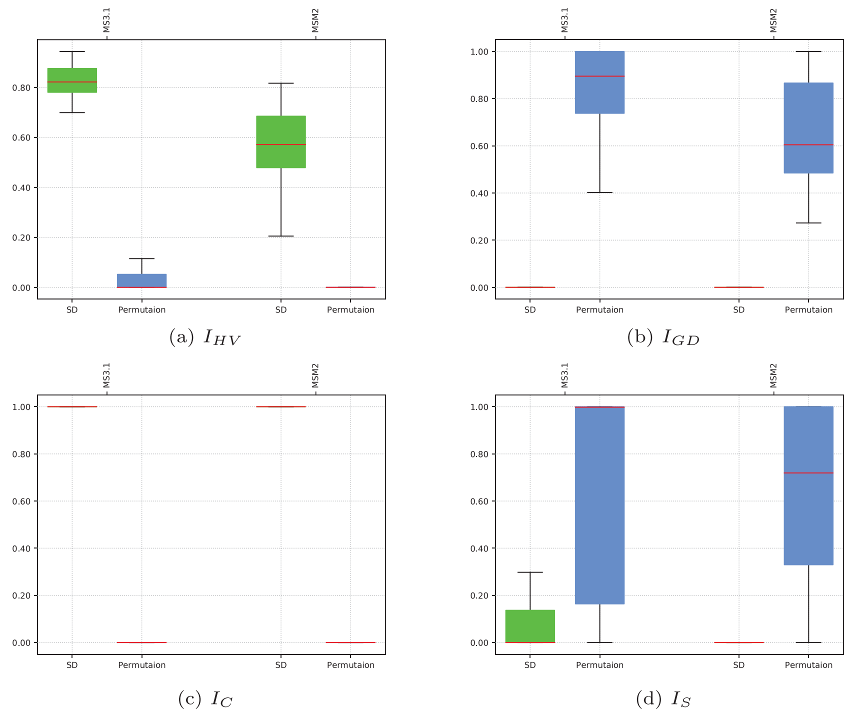


Fig. 20. Boxplots comparing the performance of quality indicators for Platform milestones. (a), (b), (c) and (d) are the boxplots for Hypervolume, Generational distance, Contribution and Spacing, respectively. Green boxplots correspond to SD, while blue ones to PE; in both, the median is given by a red horizontal line. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

Table 16
Time and cost for the ground truth (GT) are compared with SD and KRRGZ. The accuracy is presented in two different ways: *Identity-based* and *Profile-based*.

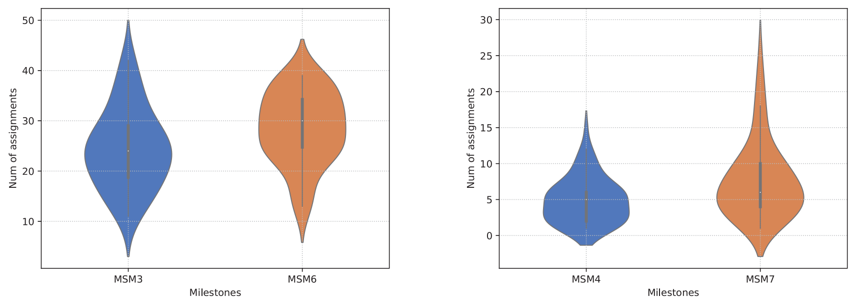
		Time			Cost			Identity-based Accu.		Profile-based Accu.	
		GT	KRRGZ	SD	GT	KRRGZ	SD	KRRGZ	SD	KRRGZ	SD
JDT	3.1	511	83	81	66,109	11,404	11,401	0.33	0.22	0.80	0.87
	3.1.1	4370	1851	2285	922,491	379,057	359,897	0.09	0.12	0.79	0.82
	3.1.2	535	1435	907	180,037	85,577	85,127	0.07	0.07	0.78	0.84
	M1	550	1335	1281	219,869	154,178	153,853	0.10	0.11	0.84	0.88
	M2	1301	1039	1233	578,609	264,933	250,752	0.12	0.17	0.76	0.78
	M3	2873	649	962	1,302,079	474,233	424,320	0.12	0.12	0.78	0.76
	M4	5520	1732	851	1,859,578	671,101	579,238	0.14	0.15	0.80	0.83
	M5	5722	2437	1383	2,238,684	780,692	660,538	0.16	0.19	0.81	0.79
	M6	4327	1125	775	1,168,936	458,848	419,551	0.17	0.18	0.81	0.80
Platform	3.0	1755	464	588	476,827	212,354	159,981	0.00	0.20	0.72	0.93
	3.1	2332	1541	1687	495,839	171,240	153,348	0.06	0.17	0.83	0.89
	M2	440	1443	1429	174,487	108,379	101,037	0.04	0.06	0.49	0.60
	M3	8298	8028	6361	1,783,080	789,205	766,426	0.08	0.05	0.49	0.51
	M4	7997	5093	6782	2,519,522	1,586,467	1,262,998	0.11	0.09	0.53	0.53
	M5	3823	4755	5610	1,530,535	997,332	807,259	0.09	0.09	0.58	0.59
	M6	3520	1311	1486	1,228,003	582,555	452,254	0.07	0.19	0.59	0.62
	M7	7238	7976	6411	3,358,271	2,043,133	1,705,809	0.10	0.08	0.63	0.71
	M8	8087	1974	1359	2,924,566	1,757,590	1,320,192	0.13	0.10	0.67	0.71
	M9	5767	1873	3034	2,827,907	1,531,975	1,113,984	0.08	0.07	0.68	0.72

6.3. External validity

The generalizability of the proposed approach is directly discussed by external validity. We have used just two software projects in our empirical study, so we cannot claim it is valid for all software systems.

In this study we used a set of data obtained from real projects. These data originate from issue tracking systems which potentially threat the generalizability concern. Moreover, in machine-learning-based solutions there is always the problem of overfitting and underfitting. By using a metaheuristic approach, we could prevent those issues with the right solution encoding and a proper configuration for initializing the experiment.

The employed data-sets were almost free of dependencies among bug fixes. We think that this is a common situation in bug fixing tasks, but we cannot assure that this is true for all software projects. When dealing with a TDG which is very connected, the scheduling approach might lose its advantage and perform similar to KRRGZ. But the role of dependencies should not be limited to intra-module dependencies, the only ones we have taken into account. There exists the risk of being dependent on a third-party library, referenced in the code. The dependency network at the ecosystem level could be used to address this problem, and the scheduling-driven approach could not show itself beneficial if a large number of dependencies are identified.



(a) JDT (b) Platform

Fig. 21. Task distribution among developers with SD for two random milestones of JDT (a) and Platform (b).

Table 17
Problem-specific metrics for a selected subset of final Pareto-optimal solutions for Platform M2. Note that total number of developers is 78 (which is the sum of "Devs. Involved" and "Devs. Without Tasks").

	Time	Cost	Devs. Involved	Mean tasks per Dev.	Devs. Without Tasks
1	324	132,938	42	3	36
2	670	116,344	32	4	46
3	382	124,128	39	3	39
4	563	116,712	33	4	45
5	380	127,438	34	3	44
6	1291	102,178	32	4	46
7	1395	102,132	34	3	44
8	1152	105,743	38	3	40
9	1075	105,796	31	4	47
10	325	132,442	42	3	36
11	386	123,017	35	3	43
12	475	119,645	34	3	44
13	500	118,673	36	3	42
14	767	113,921	36	3	42
15	434	122,469	40	3	38

7. Conclusions and future works

In this work, we have conducted an empirical study to find a better solution to the developer bug-fixing assignment problem compared to other state-of-the-art solutions. The task assignment is formulated as a multi-objective problem where the *time* and the *cost* are considered as the two objectives. A new paradigm, *Scheduling-Driven*, that extends solution encoding to improve efficiency, was introduced. Specifically, our approach helps to explore more parts of the search space, using an embedded greedy search which operates over the *schedules*.

Four different quality indicator metrics were used to perform the comparisons. We hypothesized that having better exploration and exploitation would result in better values of these quality indicators metrics, and have shown an improvement of the *scheduling-driven* approach compared to KRRGZ and other approaches.

The scheduling-driven approach can be employed in both proprietary and open source system development. Bug fixing teams could benefit from the proposed approach to manage the cost of change management with respect to finding an efficient way for bug fixing assignment. As an advantage, it could be beneficial for handling assignments without any specific presumptions about the codebase or the number of bug reports.

The proposed approach could be implemented as part of an issue tracking system that receives many issues on a regular basis. To work properly, tasks associated with bug issues should have few dependencies on other bugs. Examining possible orders of tasks assigned to a developer is a time-consuming activity. To this end, we have used a greedy local search during solution evaluation to iterate over those orders. Our research has shown that the SD approach is an efficient solution for supporting ordering issues in bug assignment.

As future work, other goals that we have not be taken into account in our research could be examined. As an example, we could think of a way for optimizing developer *knowledge* of the project, i.e., the system would assign tasks to developers in such a way that in the long term the experience of the development team as a whole is maximized. For this to be possible, we would need at least data (or a model) of developer turnover and of the learning curves, among others. Turnover usually causes losing high experienced developers and consequently puts the project at risk. Therefore, task assignments which results in more *knowledge diffusion* would be of high interest.

CRediT authorship contribution statement

Vahid Etemadi: Software, Conceptualization, Investigation, Writing - original draft. **Omid Bushehrian:** Conceptualization, Supervision. **Reza Akbari:** Conceptualization, Validation. **Gregorio Robles:** Validation, Writing - review & editing, Supervision.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

When writing this article, the first author was visiting the lab of the last author for eight months for a doctoral research visit. GR acknowledges the support of the Government of Spain through project "BugBirth" (RTI2018-101963-B-100).

References

Alencar da Costa, D., McIntosh, S., Shang, W., Kulesza, U., Coelho, R., Hassan, A., 2016. A framework for evaluating the results of the SZZ approach for identifying bug-introducing changes. IEEE Trans. Softw. Eng. 1. http://dx.doi.org/10.1109/TSE.2016.2616306, URL http://ieeexplore.ieee.org/document/7588121/.

Anvik, J., Hiew, L., Murphy, G.C., 2006. Who should fix this bug? In: Proceeding of the 28th International Conference on Software Engineering - ICSE '06. ACM, ACM Press, New York, New York, USA, p. 361. http://dx.doi.org/10.1145/1134285.1134336, URL http://portal.acm.org/citation.cfm?doid=1134285.1134336.

Anvik, J., Murphy, G.C., 2011. Reducing the effort of bug report triage. ACM Trans. Softw. Eng. Methodol. 20 (3), 1–35. http://dx.doi.org/10.1145/2000791.2000794, URL http://dl.acm.org/citation.cfm?doid=2000791.2000794.

Arcuri, A., Briand, L., 2011. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In: Proceeding of the 33rd International Conference on Software Engineering - ICSE '11. ACM, ACM Press, New York, New York, USA, p. 1. http://dx.doi.org/10.1145/1985793.1985795, URL http://portal.acm.org/citation.cfm?doid=1985793.1985795.

Arcuri, A., Briand, L., 2014. A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. Softw. Test. Verif. Reliab. 24 (3), 219–250. http://dx.doi.org/10.1002/stvr.1486, arXiv:arXiv:1709.04631v1, URL http://onlinelibrary.wiley.com/doi/10.1002/stvr.450/pdf http://doi.wiley.com/10.1002/stvr.1486.

Arrieta, A., Segura, S., Markiegi, U., Sagardui, G., Etxeberria, L., 2018. Spectrum-based fault localization in software product lines. Inf. Softw. Technol. 100, 18–31.

Bergstra, J., Bengio, Y., 2012. Random search for hyper-parameter optimization. J. Mach. Learn. Res. 13 (2).

Blincoe, K., Valetto, G., Damian, D., 2015. Facilitating coordination between software developers: A study and techniques for timely and efficient recommendations. IEEE Trans. Softw. Eng. 41 (10), 969–985. http://dx.doi.org/10.1109/TSE.2015.2431680, URL http://ieeexplore.ieee.org/document/7105409/.

Catolino, G., Palomba, F., Zaidman, A., Ferrucci, F., 2019. Not all bugs are the same: Understanding, characterizing, and classifying bug types. J. Syst. Softw. 152, 165–181.

Cavalcanti, Y.C., Machado, I.d.C., Neto, P.A.D.M.S., Almeida, E.S.D., 2016. Towards semi-automated assignment of software change requests. J. Syst. Softw. 115, 82–101. http://dx.doi.org/10.1016/j.jss.2016.01.038, URL http://linkinghub.elsevier.com/retrieve/pii/S0164121216000352.

Coello, C.A., Lamont, G.B., Van Veldhuizen, D.A., et al., 2007. Evolutionary Algorithms for Solving Multi-Objective Problems, Vol. 5. Springer.

Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C., 2009. Introduction to Algorithms, third ed. MIT Press, Cambridge, USA, p. 1292.

Črepinšek, M., Liu, S.-H., Mernik, M., 2013. Exploration and exploitation in evolutionary algorithms. ACM Comput. Surv. 45 (3), 1–33. http://dx.doi.org/10.1145/2480741.2480752, arXiv:arXiv:1502.07526v1, URL http://dl.acm.org/citation.cfm?doid=2480741.2480752.

Deb, K., Pratap, A., Agarwal, S., Meyarivan, T., 2002. A fast and elitist multiobjective genetic algorithm: NSGA-II. IEEE Trans. Evol. Comput. 6 (2), 182–197. http://dx.doi.org/10.1109/4235.996017, URL http://ieeexplore.ieee.org/document/996017/.

Dechter, R., 2003. Constraint Processing. Morgan Kaufmann.

Di Penta, M., Harman, M., Antoniol, G., 2011. The use of search-based optimization techniques to schedule and staff software projects: an approach and an empirical study. Softw. - Pract. Exp. 41 (5), 495–519.

Du, K.-L., Swamy, M.N.S., 2016. Search and optimization by metaheuristics. In: Search and Optimization by Metaheuristics. Springer International Publishing, Cham, http://dx.doi.org/10.1007/978-3-319-41192-7, URL http://link.springer.com/10.1007/978-3-319-41192-7.

Durillo, J.J., Nebro, A.J., 2011. Jmetal: A java framework for multi-objective optimization. Adv. Eng. Softw. 42 (10), 760–771. http://dx.doi.org/10.1016/j.advengsoft.2011.05.014, URL http://linkinghub.elsevier.com/retrieve/pii/S0965997811001219.

Hadka, D., Reed, P., 2012. Diagnostic assessment of search controls and failure modes in many-objective evolutionary optimization. Evol. Comput. 20 (3), 423–452, URL http://www.mitpressjournals.org/doi/10.1162/EVCO_a_00053.

Harman, M., McMinn, P., De Souza, J.T., Yoo, S., 2012. Search based software engineering: Techniques, taxonomy, tutorial. In: Empirical Software Engineering and Verification. Springer, pp. 1–59.

Ishibuchi, H., Murata, T., 1996. Multi-objective genetic local search algorithm. In: Proceedings of IEEE International Conference on Evolutionary Computation. IEEE, pp. 119–124. http://dx.doi.org/10.1109/ICEC.1996.542345, URL http://ieeexplore.ieee.org/document/542345/.

Kagdi, H., Gethers, M., Poshyvanyk, D., Hammad, M., 2012. Assigning change requests to software developers. J. Softw.: Evol. Process 24 (1), 3–33. http://dx.doi.org/10.1002/smr.530, URL http://doi.wiley.com/10.1002/smr.530.

Karim, M.R., Ruhe, G., Rahman, M.M., Garousi, V., Zimmermann, T., 2016. An empirical investigation of single-objective and multiobjective evolutionary algorithms for developer's assignment to bugs. J. Softw.: Evol. Process 28 (12), 1025–1060. http://dx.doi.org/10.1002/smr.1777, URL http://doi.wiley.com/10.1002/smr.1777.

Kearney, K.T., Torelli, F., 2011. The SLA model. In: Service Level Agreements for Cloud Computing. Springer, pp. 43–67.

Kocaguneli, E., Menzies, T., Keung, J.W., 2012. On the value of ensemble effort estimation. IEEE Trans. Softw. Eng. 38 (6), 1403–1416. http://dx.doi.org/10.1109/TSE.2011.111, URL http://ieeexplore.ieee.org/document/6081882/.

Maalej, W., Ellmann, M., Robbes, R., 2017. Using contexts similarity to predict relationships between tasks. J. Syst. Softw. 128, 267–284. http://dx.doi.org/10.1016/j.jss.2016.11.033, URL http://linkinghub.elsevier.com/retrieve/pii/S0164121216302357.

Pressman, R.S., 2010. Software Engineering : A Practitioner's Approach, seventh ed. McGraw-Hill, New York, NY, USA, p. 880.

Ren, J., Harman, M., Di Penta, M., 2011. Cooperative co-evolutionary optimization of software project staff assignments and job scheduling. In: Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). In: LNCS, vol. 6956, Springer, pp. 127–141, URL http://link.springer.com/10.1007/978-3-642-23716-4_14.

Ronald, S., 1995. Genetic Algorithms and Permutation-Encoded Problems: Diversity Preservation and a Study of Multimodality (Ph.D. thesis). University of South Australia.

Runeson, P., Höst, M., 2009. Guidelines for conducting and reporting case study research in software engineering. Empir. Softw. Eng. 14 (2), 131.

Sarro, F., Ferrucci, F., Harman, M., Manna, A., Ren, J., 2017. Adaptive multi-objective evolutionary algorithms for overtime planning in software projects. IEEE Trans. Softw. Eng. 1. http://dx.doi.org/10.1109/TSE.2017.2650914, URL http://ieeexplore.ieee.org/document/7814340/.

Sarro, F., Petrozziello, A., Harman, M., 2016. Multi-objective software effort estimation. In: Proceedings of the 38th International Conference on Software Engineering - ICSE '16. ACM, ACM Press, New York, New York, USA, pp. 619–630. http://dx.doi.org/10.1145/2884781.2884830, URL http://dl.acm.org/citation.cfm?doid=2884781.2884830.

Shokripour, R., Anvik, J., Kasirun, Z.M., Zamani, S., 2015. A time-based approach to automatic bug report assignment. J. Syst. Softw. 102, 109–122.

Stylianou, C., Andreou, A.S., 2014. Human resource allocation and scheduling for software project management. In: Software Project Management in a Changing World. Springer, pp. 73–106.

Tamrawi, A., Nguyen, T.T., Al-Kofahi, J.M., Nguyen, T.N., 2011. Fuzzy set and cache-based approach for bug triaging. In: Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering. pp. 365–375.

Van Veldhuizen, D.A., Lamont, G.B., 1998. Multiobjective Evolutionary Algorithm Research: A History and Analysis. Tech. rep., Citeseer.

Wu, L., Garg, S.K., Buyya, R., 2015. Service level agreement (SLA) based saas cloud management system. In: 2015 IEEE 21st International Conference on Parallel and Distributed Systems (ICPADS), Vol. 2016-Janua. IEEE, IEEE, pp. 440–447. http://dx.doi.org/10.1109/ICPADS.2015.62, URL http://ieeexplore.ieee.org/document/7384325/.

Wu, L., Kumar Garg, S., Buyya, R., 2012. SLA-based admission control for a Software-as-a-Service provider in Cloud computing environments. J. Comput. System Sci. 78 (5), 1280–1299. http://dx.doi.org/10.1016/j.jcss.2011.12.014, URL http://linkinghub.elsevier.com/retrieve/pii/S0022000011001590.

Xia, X., Lo, D., 2017. An effective change recommendation approach for supplementary bug fixes. Autom. Softw. Eng. 24 (2), 455–498. http://dx.doi.org/10.1007/s10515-016-0204-z, URL http://link.springer.com/10.1007/s10515-016-0204-z.

Xia, X., Lo, D., Ding, Y., Al-Kofahi, J.M., Nguyen, T.N., Wang, X., 2016. Improving automated bug triaging with specialized topic model. IEEE Trans. Softw. Eng. 5589 (c), 1. http://dx.doi.org/10.1109/TSE.2016.2576454, URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=7484672 http://ieeexplore.ieee.org/document/7484672/.

Xia, X., Lo, D., Wang, X., Yang, X., 2015a. Who should review this change?: Putting text and file location analyses together for more accurate recommendations. In: 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME). IEEE, IEEE, pp. 261–270. http://dx.doi.org/10.1109/ICSM.2015.7332472, URL http://ieeexplore.ieee.org/document/7332472/.

Xia, X., Lo, D., Wang, X., Zhou, B., 2015b. Dual analysis for recommending developers to resolve bugs. J. Softw.: Evol. Process 27 (3), 195–220. http://dx.doi.org/10.1002/smr.1706, URL http://doi.wiley.com/10.1002/smr.1706.

Xuan, J., Jiang, H., Hu, Y., Ren, Z., Zou, W., Luo, Z., Wu, X., 2015. Towards effective bug triage with software data reduction techniques. IEEE Trans. Knowl. Data Eng. 27 (1), 264–280. http://dx.doi.org/10.1109/TKDE.2014.2324590, URL http://ieeexplore.ieee.org/document/6815966/.

Ye, X., Bunescu, R., Liu, C., 2016. Mapping bug reports to relevant files: A ranking model, a fine-grained benchmark, and feature evaluation. IEEE Trans. Softw. Eng. 42 (4), 379–402. http://dx.doi.org/10.1109/TSE.2015.2479232, URL http://ieeexplore.ieee.org/document/7270328/.

Yeo, C.S., Buyya, R., 2005. Service level agreement based allocation of cluster resources: Handling penalty to enhance utility. In: 2005 IEEE International Conference on Cluster Computing. IEEE, IEEE, pp. 1–10. http://dx.doi.org/10.1109/CLUSTR.2005.347075, URL http://ieeexplore.ieee.org/document/4154118/.

Zhang, T., Chen, J., Yang, G., Lee, B., Luo, X., 2016a. Towards more accurate severity prediction and fixer recommendation of software bugs. J. Syst. Softw. 117, 166–184. http://dx.doi.org/10.1016/j.jss.2016.02.034.

Zhang, T., Chen, J., Zhan, X., Luo, X., Lo, D., Jiang, H., 2019. Where2change: Change request localization for app reviews. IEEE Trans. Softw. Eng. http://dx.doi.org/10.1109/TSE.2019.2956941.

Zhang, T., Jiang, H., Luo, X., Chan, A.T.S., 2016b. A literature review of research in bug resolution: Tasks, challenges and future directions. Comput. J. 59 (5), 741–773. http://dx.doi.org/10.1093/comjnl/bxv114.

Zhou, J., Zhang, H., Lo, D., 2012. Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports. In: 2012 34th International Conference on Software Engineering (ICSE). IEEE, pp. 14–24.

Zitzler, E., Brockhoff, D., Thiele, L., 2007. The hypervolume indicator revisited: On the design of Pareto-compliant indicators via weighted integration. In: Evolutionary Multi-Criterion Optimization. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 862–876.

Zitzler, E., Thiele, L., Laumanns, M., Fonseca, C., da Fonseca, V., 2003. Performance assessment of multiobjective optimizers: an analysis and review. IEEE Trans. Evol. Comput. 7 (2), 117–132. http://dx.doi.org/10.1109/TEVC.2003.810758, URL http://ieeexplore.ieee.org/document/1197687/.