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

Code Reviewer Recommendations as a Multi-Objective Problem: Balancing Expertise, Availability and Collaborations

Soumaya Rebai · Abderrahmen Amich · Somayeh Molaei · Marouane Kessentini · Rick Kazman

Received: date / Accepted: date

Abstract Modern Code review is one of the most critical tasks in software maintenance and evolution. A rigorous code review leads to fewer bugs and reduced overall maintenance costs. Most existing studies focus on automatically identifying the most qualified reviewers, based on their expertise, to review pull-up requests. However, the management of code reviews is a complex problem in practice due to a project's limited resources, including the availability of peer reviewers. Furthermore, the history of collaborations between developers and reviewers could affect the quality of the reviews, in positive or negative ways. In this paper, we formulate the recommendation of code reviewers as a multi-objective search problem to balance the conflicting objectives of expertise, availability, and history of collaborations. Our validation confirms the effectiveness of our multi-objective approach on 9 open source projects by making better recommendations, on average, than the state of the art.

S. Rebai
University of Michigan, USA E-mail: srebal@umich.edu

A. Amich
University of Michigan, USA E-mail: aamich@umich.edu

S. Molaei
University of Michigan, USA E-mail: smolaei@umich.edu

M. Kessentini
University of Michigan, USA E-mail: marouane@umich.edu

R. Kazman
University of Hawaii, USA E-mail: kazman@hawaii.edu

Keywords Modern code review · search based software engineering · reviewers recommendation

1 Introduction

The source code review process has always been one of the most important software maintenance and evolution activities [14]. Several studies show that a careful code inspection can significantly reduce defects and improve the quality of software systems. Recently this process has become informal, asynchronous, light-weight and facilitated by tools [4] [33]. A survey with practitioners, performed by Bacchelli et al. [3], show that code review nowadays is expanding beyond just looking for defects but to also provide alternatives to improve the code and transfer knowledge among developers.

Despite recent progress [30, 46] code reviews are still time-consuming, expensive, and complex involving a large amount of effort by managers, developers and reviewers. Thongtanunam et al. [37] found on four open source projects with 12 days as the average to approve a code change. The automated recommendation of peer code reviewers may help to reduce delays by finding the best reviewers who will then spend less time in reviewing the assigned files.

The majority of existing tools and techniques for automated recommendation of code reviewers are based on the level of reviewer expertise [4,36,37,46]. Expertise is mainly defined as the prior knowledge of the changes under review. For instance, a selected peer reviewer with high expertise should have reviewed the same files [36, 37], or even the same lines of code in the files [4]. An empirical study at Microsoft found that selected reviewers with high expertise can provide valuable and rapid feedback to the author of the code under review [3]. However, reviewers with high expertise may not be always available in practice, or at least assigning them may create delays.

To address the above challenges we propose to formulate the selection of peer code reviewers as a multi-objective problem. The goal is to balance the conflicting objectives of expertise, availability and history of collaborations. The multi-objective approach tries to find a trade-off between multiple objectives and minimizing the former collaborations on reviewing the same files is just one component between many objectives. We adopted one of the widely used multi-objective search algorithms, NSGA-II [16], to find a trade-off depending on current context and available resources. For instance, our formulation can slightly sacrifice expertise to avoid a delay caused by limited resources (e.g. low availability of peer reviewers). In another context, the reviewer(s) with the highest expertise can be selected when the goal is to inspect high priority code changes such as critical buggy files. Thus, our approach enables navigation between the three different dimensions by generating multiple non-dominated peer reviewer recommendations instead of one solution as is done in existing work.

Our validation on 9 open source confirms the effectiveness of our multi-objective approach by making better recommendations than the state of the art.

The remainder of this paper is organized as follows. Section 2 presents the relevant background related to this research and the problem statement. Section 3 describes our approach overview and the adaptation steps. Empirical study and results are provided in Section 4 while threats to validity are discussed in Section 5. Section 6 is dedicated to related work. Finally, we conclude and provide our future research directions in Section 7.

2 Background

2.1 Review Process

We begin by defining the key concepts related to the modern code review process supported nowadays by many tools such as Gerrit[1]. A code review includes all the interactions between the submitter of a pull-request and one or more reviewers of that change including comments on the code and discussions with reviewers. The owner is the programmer making the changes to the code and then submitting the review request. A peer reviewer is a developer assigned to contribute in reviewing the set of code changes. These reviewers write review comments as feedback to the owner about the introduced changes.

Figure 1 shows the *code review* process in a version-control repository. A code review process starts with a new branch (①). In this new branch, each commit should correspond to a code-level change (②). After developers commit all the code-level changes, developers make a pull request, in which they write a description of the code changes (③). After a pull request has been sent out, it appears in the list of pull requests for the project in question, visible to anyone who can see the project. Then, other collaborators can check the changes made in the branch and discuss the changes (code reviews ④). During the code review, developers may make more changes to the branch. Finally, if the collaborators accept these code changes, this branch is merged into the master branch (⑤).

Figure 2 shows one example of code reviews where many possible reviewers can be assigned to review the changes. Thus, dealing with a large number of possible reviewers for multiple pull requests is a management problem which is under-studied in the research literature. This management process requires handling multiple competing criteria including expertise, availability and previous collaborations with the owners and reviewers. We will describe, in the next section, our formulation of code reviewer recommendations as a multi-objective problem.

[1]https://www.gerritcodereview.com/intro-quick.html

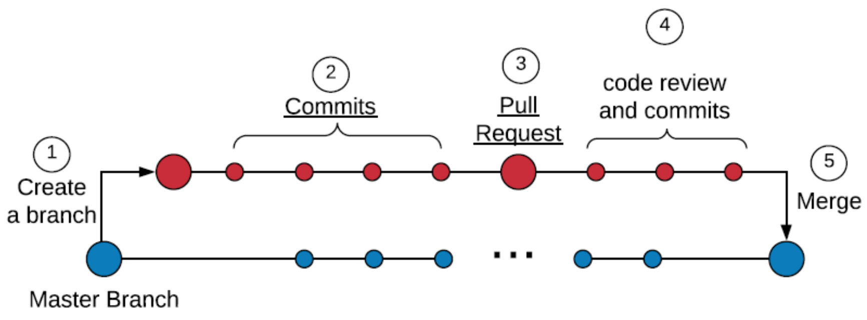
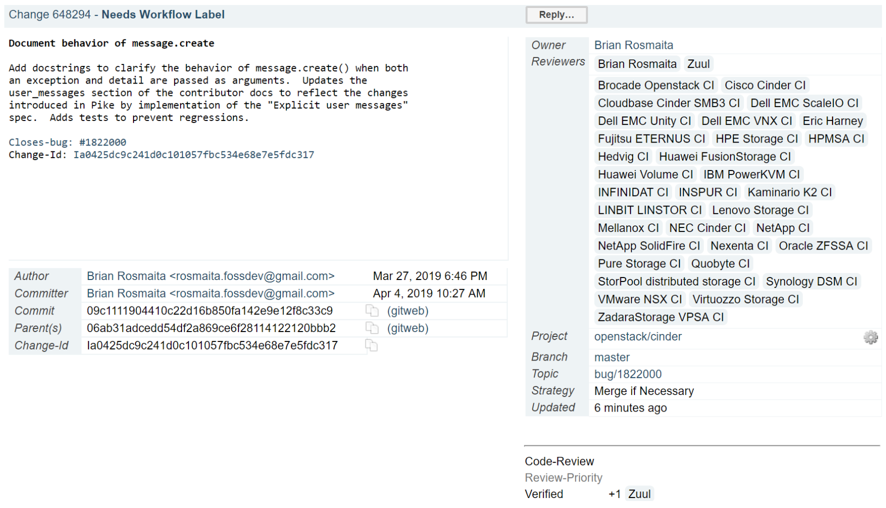


Fig. 1 A summary of the code review process.



Change 648294 - **Needs Workflow Label**

```
Document behavior of message.create

Add docstrings to clarify the behavior of message.create() when both
an exception and detail are passed as arguments.  Updates the
user_messages section of the contributor docs to reflect the changes
introduced in Pike by implementation of the "Explicit user messages"
spec.  Adds tests to prevent regressions.

Closes-bug: #1822000
Change-Id: Ia0425dc9c241d0c101057fbc534e68e7e5fdc317
```

Author	Brian Rosmaita <rosmaita.fossdev@gmail.com>	Mar 27, 2019 6:46 PM
Committer	Brian Rosmaita <rosmaita.fossdev@gmail.com>	Apr 4, 2019 10:27 AM
Commit	09c1111904410c22d16b850fa142e9e12f8c33c9	(gitweb)
Parent(s)	06ab31adcedd54df2a869ce6f28114122120bbb2	(gitweb)
Change-Id	Ia0425dc9c241d0c101057fbc534e68e7e5fdc317	

Owner	Brian Rosmaita
Reviewers	Brian Rosmaita Zuul
	Brocade Openstack CI Cisco Cinder CI Cloudbase Cinder SMB3 CI Dell EMC ScaleIO CI Dell EMC Unity CI Dell EMC VNX CI Eric Harney Fujitsu ETERNUS CI HPE Storage CI HPMSA CI Hedvig CI Huawei FusionStorage CI Huawei Volume CI IBM PowerKVM CI INFINIDAT CI INSPUR CI Kaminario K2 CI LINBIT LINSTOR CI Lenovo Storage CI Mellanox CI NEC Cinder CI NetApp CI NetApp SolidFire CI Nexenta CI Oracle ZFSSA CI Pure Storage CI Quobyte CI StorPool distributed storage CI Synology DSM CI VMware NSX CI Virtuozzo Storage CI ZadaraStorage VPSA CI
Project	openstack/cinder
Branch	master
Topic	bug/1822000
Strategy	Merge if Necessary
Updated	6 minutes ago

Code-Review
Review-Priority
Verified +1 Zuul

Fig. 2 An example of a code review extracted from OpenStack

2.2 Preliminary Study

As part of preliminary work of this paper, we performed unstructured survey with 6 senior managers and 11 senior developers actively involved in code reviews to assign reviewers or/and review pull-requests. We decided to perform an unstructured survey to encourage the participants to think-aloud and avoid biasing them with our opinions. Furthermore, the goal of our surveys is get insights about the current challenges in code reviews rather than a large empirical study. We found that 10 days is the average to approve a code change at eBay. The main reason based on the surveys for the delay is the challenging task of identifying the right reviewers which is aligned with existing studies[40, 44].

A senior manager confirmed that "We don't actually need more tools to just suggest reviewers based on expertise. We need better support to manage code reviews especially with short deadlines and limited resources while not sacrificing a lot of expertise. It is a complex problem." In addition, the participants highlighted that it is critical to consider the priority of the files to be inspected as part of the management of the code review process. Furthermore, we found in our interviews that the social interactions between code authors and reviewers is another critical aspect to consider to ensure high quality reviews.

Existing studies assume that peer reviewers with high interactions with authors/owners of the code under review are the best to select [30]. However, this aspect may be considered negative with extensive mutual peer reviews and/or quick approval of code changes as suggested by the participants. The diversity of peer code reviews is important, as pointed out by the eBay senior managers and peer reviewers, especially when frequent patterns of code authors/reviewers are observed.

3 Approach

In this section, we describe our proposed approach for recommending the most appropriate set of reviewers for pull-requests to be reviewed using multi-objective search.

3.0.1 Multi-Objective Optimization

Multi-Objective search considers more than one objective function to be optimized simultaneously. It is hard to find an optimal solution that solves such problems because the objectives to be optimized are conflicting. For this reason, a multi-objective search-based algorithm could be suitable to solve this problem because it finds a set of alternative solutions, rather than a single solution as result. One of the widely used multi-objective search techniques is NSGA-II [1, 16, 31] that has shown good performance in solving several software engineering problems [22].

A high-level view of NSGA-II is depicted in Algorithm 1. The algorithm starts by randomly creating an initial population P_0 of individuals encoded using a specific representation (line 1). Then, a child population Q_0 is generated from the population of parents P_0 (line 2) using genetic operators (crossover and mutation). Both populations are merged into an initial population R_0 of size N (line 5). *Fast-non-dominated-sort* [16] is the technique used by NSGA-II to classify individual solutions into different dominance levels (line 6). Indeed, the concept of non-dominance consists of comparing each solution x with every other solution in the population until it is dominated (or not) by one of them. According to Pareto optimality: "A solution x_1 is said to dominate another solution x_2, if x_1 is no worse than x_2 in all objectives and x_1 is strictly better than x_2 in at least one objective". Formally, if we consider a set of objectives f_i , $i \in 1..n$, to maximize, a solution x_1 dominates x_2 :

$$\textit{iff } \forall i,\ f_i(x_2) \leqslant f_i(x_1) \text{ and } \exists j \mid f_j(x_2) < f_j(x_1)$$

Algorithm 1 High level pseudo code for NSGA-II

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

The whole population that contains N individuals (solutions) is sorted using the dominance principle into several fronts (line 6). Solutions on the first Pareto-front F_0 get assigned dominance level of 0. Then, after taking these solutions out, *fast-non-dominated-sort* calculates the Pareto-front F_1 of the remaining population; solutions on this second front get assigned dominance level of 1, and so on. The dominance level becomes the basis of selection of individual solutions for the next generation. Fronts are added successively until the parent population P_{t+1} is filled with N solutions (line 8). When NSGA-II has to cut off a front F_i and select a subset of individual solutions with the same dominance level, it relies on the crowding distance [16] to make the selection (line 9). This parameter is used to promote diversity within the population. This front F_i to be split, is sorted in descending order (line 13), and the first (N- $|P_{t+1}|$) elements of F_i are chosen (line 14). Then a new population Q_{t+1} is created using selection, crossover and mutation (line 15). This process will be repeated until reaching the last iteration according to stop criteria (line 4).

3.1 Approach Overview: A multi-objective Code Reviewer Recommendation Framework

The ultimate goal of our Code Reviewer Recommendation framework is to automatically assign the most appropriate reviewers to newly opened pull-requests. The assignment is performed by balancing three important competing criteria: the expertise of the reviewers, their availability (considering their current workload) and their social connections (collaborations) with the submitter of the open pull request(s). Thus, we propose to use multi-objective search, based on NSGA-II [16], to find a tradeoff between the different competing objectives. An overview of the approach is illustrated in Figure 3.

Our approach takes as input: 1) the pull-request(s) to be reviewed; 2) the pull-request(s) under review and the involved reviewers; and 3) the detailed

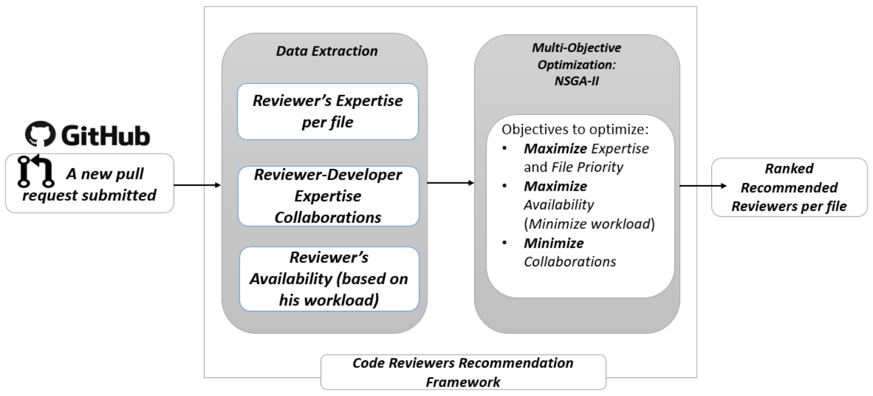

GitHub
A new pull request submitted
Data Extraction
Reviewer's Expertise per file
Reviewer-Developer Expertise Collaborations
Reviewer's Availability (based on his workload)
Multi-Objective Optimization: NSGA-II
Objectives to optimize:
Maximize Expertise and File Priority
Maximize Availability (Minimize workload)
Minimize Collaborations
Ranked Recommended Reviewers per file
Code Reviewers Recommendation Framework

Fig. 3 Overview of our multi-objective search-based approach for code reviewer recommendation.

history of closed pull-requests. The extraction of these 3 required inputs is easy and straightforward by simply providing the GitHub link of the project to our tool. Using our integrated parser, we automatically analyze the GitHub repository to collect the code review history, commit messages and source code. Next, from the collected data, we extract three clusters of interaction information: a File-Reviewer interaction matrix (FR), a Developer-Reviewer interaction matrix (DR) and a File-Developer interaction matrix (FD). From the open pull-requests to be reviewed we can automatically extract the files that need to be reviewed and evaluate the expertise of assigned reviewers in our solution representation, as detailed later.

As an output, our multi-objective algorithm generates a set of trade-off solutions where each solution consists of assigning one or more reviewers per pull-request. Thus, the solution can be represented as a matrix matching reviewers to the files of the pull-request(s). For each file, the reviewers are ranked based on their level of expertise to review the file, their availability, and their past collaboration with the developer of that file, all while reducing the number of reviewers per pull-request as much as possible.

To find a trade-off between the different objectives, we used NSGA-II [16] since it was used for similar discrete problems in software engineering and performed well. The use of a metaheuristic algorithm to deal with conflicting objectives is justified by the large search space to explore. Let M be the number of total reviewers and P number of total files submitted to be reviewed for code changes. The size of the search space to explore in order to find the best subset of m reviewers among a set of M reviewers to review p files is of $\begin{pmatrix} m \\ M \end{pmatrix} \times p = \frac{m!}{m!(M-m)!} \times p$. This is a very fast growing function and as M grows the search space becomes prohibitively large to the point where exhaustive search is not practical. We propose the use of metaheuristic search to explore this combinatorial search space to find near-optimum reviewer recommendations.

he multi-objective approach proposed in this paper generates as output a set of non-dominated solutions (Pareto front). It is upto the team manager to select the reviewers assignment solution based on their preferences

Thus, the final output of the algorithm is a set of solutions (Pareto front) representing trade-offs between the three objectives. It is up to the manager to select the reviewers assignment (choose a solution) based on their preferences. In general, the preferences are defined based on the current context: urgency to release code quickly, available resources, speedy growth phase of the project, etc. These different contexts are not changing daily and they are not related to only one or few pull-requests but more related to the situation of the whole project. The preferred solution can be quickly selected by looking at the distribution of the solutions in the Pareto front or ranking the solutions based on the most preferred fitness function based on the current context. The two common ways to extract a solution from the Pareto front are the use of the reference point and the knee point [15, 17, 25, 32]. The knee point corresponds to the solution with the maximal trade-off between all fitness functions, i.e., a vector of the best objective values for all solutions. In order to find the maximal trade-off, we use the trade-off worthiness metric proposed by [32] to evaluate the worthiness of each solution in terms of objective value compromise. While the knee point selection may not be the perfect way, it is the only strategy to ensure a fair comparison with the mono-objective and deterministic approaches since they generate only one solution as output.

The manager may select a reference point with high expertise, if s(he) cares about finding knowledgeable reviewers of the files while accepting some delays in the review process. Thus, the selected solution will be the closest one to the specified reference point. This scenario happens, for example, when a pull-request is modifying some security critical files. However, it is not required that the managers specify the reference point for each pull-request since the preferences usually depend on the context of the whole project and they do not change daily. Moreover, the knee point can be automatically calculated based on the distribution of the solutions in the Pareto front [25] and it represents the maximum trade-off between the objectives.

3.2 Main Components of the Approach

3.2.1 Reviewer's Expertise Model

This model aims at exploring reviewer-file connections: Who are the peer reviewers who worked on the same file? From the previous commits and closed pull-requests, we can automatically extract a matrix that represents the expertise of reviewers. Expertise value is defined as the number of times that the reviewer reviewed the same file. In fact, for every file, the matrix keeps track of reviewers who reviewed that specific file and how many times every reviewer reviewed that particular file.

FR is a $P \times M$ matrix where each entry $fr_{k,i}$ represents the number of times reviewer r_i reviewed or modified file f_k where $i \in \{1, 2, \ldots, M\}$, $k \in \{1, 2, \ldots, P\}$, P is total number of files requested to be reviewed and M total number of reviewers working on the project. This matrix represents how familiar is each reviewer with each file, which is used as a proxy measure for expertise.

$$FR = (fr_{(k,i)})\varepsilon^{P \times M} \tag{1}$$

3.2.2 Reviewer-Developer Collaboration Model

To take the socio-technical factor into account when searching for the best reviewers to review a code change, we extracted the collaborations between reviewers and developers from the history of closed pull-requests. In fact, for every potential recommended reviewer, we extract both the list of developers and the files per pull-request that he/she reviewed or modified in the past. Then, we calculated for each pair (reviewer,developer) the total number of commonly modified files. Note that the reviewer can be found in the comments of the pull-requests of the submitter (developer). Thus, a "Collaborations" matrix DR is automatically created.

To sum up, DR is a $N \times M$ matrix where each entry $dr_{j,i}$ represents the number of times reviewer r_i reviewed a file changed by developer d_j where $i \in \{1, 2, \ldots, M\}$, $j \in \{1, 2, \ldots, N\}$, N is total number of developers working on the project and M total number of reviewers working on the project. In fact, $dr_{j,i}$ is defined as the number of files that the reviewer and the developer collaborated together (reviewed or modified) in the past. This matrix represents the social connections between reviewers and developers.

$$DR = (dr_{(j,i)})\varepsilon^{N \times M} \tag{2}$$

3.2.3 Availability Model

To estimate the availability of peer reviewers, we considered of the number of files per open pull-requests and numbers of commits where they are currently involved. We represented the availability (workload) in a vector $A = [a_1, a_2, \ldots, a_M]$ where a_i represents the total number of files of open pull requests and commits for a reviewer r_i.

Data. For *expertise* and *collaborations*, we considered all the data since the start of the project because we believe that more information about the expertise and collaborations of the developers is useful in assigning the appropriate reviewer. Regarding the *availability model*, we considered the last 7 days of open pull requests because we wanted to have an estimate of the current workload of the reviewers.

3.3 Problem Formulation

3.3.1 Solution Representation

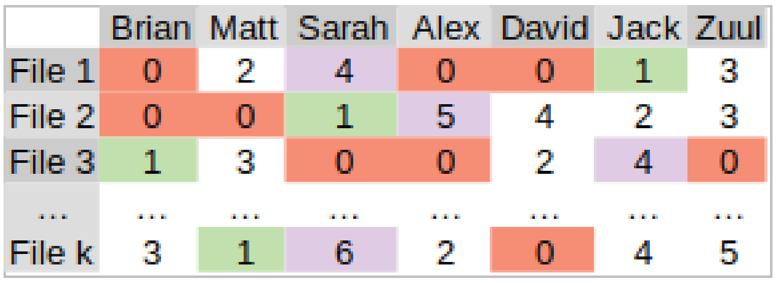
The solution of the optimization problem is a matrix S that contains an integer value $o \in \{0, 1, 2, \ldots, M\}$ for entry $s_{k,i}$ denoting the recommended order (rank) for the reviewer r_i to review file f_k. This matrix contains P rows and M columns. P is the number of files that contains code changes to be reviewed and M is the number of potential reviewers. To initialize the matrix S, we first extract the number M because it represents the number of candidate reviewers for the files to be reviewed in the submitted pull-request. Second, we extract the files to be reviewed in the pull-request to review. Then, initially, each S[k,i] will take a distinct random number. Assigning 0 to S[k,i] means that the kth developer is not assigned to review the ith file and assigning an integer 0<o<=M means that the developer is assigned to review the ith changed file and his rank is o within the list of appropriate reviewers.

After each iteration, the genetic algorithm decides if a reviewer is suitable for a review assignment for a specific file or not. If yes, it will decide the rank of that reviewer, compared to other candidate reviewers for the same file, based on our three objectives (defined in the section 3.3.2).

An example of a two-dimensional solution representation is illustrated in Figure 4. Let say we have seven reviewers who are working on the project: Brian, Matt, John, Alex, David, jack and Zuul, and there are k files with code changes. Based on our solution representation, we suggest which reviewers are appropriate for reviewing which file(s) and in what order. In this example, Brian is not recommended to review $file1$ and $file2$, but he is the most appropriate reviewer to review the changes in $filek$. To review $file1$, Matt is the second best reviewer and Zuul is the third best one. To sum up, our multi-objective algorithm outputs reviewer-file matrix (as shown in Figure 4) which assigns reviewers to all the files changed in the submitted pull-request. Thus, for each pull-request (PR) we rank the reviewers based on how many files in that PR he/she is able to review taking into consideration the different fitness functions.

3.3.2 Fitness Functions

In our approach, we aim to optimize three fitness functions. The first and the second ones are formulated to maximize the *expertise* and *the availability* of the reviewers. While the third fitness function is formulated to minimize the social connections between reviewers and developers in the hope of reducing human bias. The motivation of our multi-objective approach is aligned with the observation of a recent study at Microsoft [11] highlighting that promoting diversity depends on the norms of the team, i.e., some teams prefer diverse, some teams prefer close connections. While previous collaborations between developers and reviewers could reduce the tension around the review task, the extensive former interactions/collaborations can be an indication of light/weak



	Brian	Matt	Sarah	Alex	David	Jack	Zuul
File 1	🟥 0	2	🟪 4	🟥 0	🟥 0	🟩 1	3
File 2	🟥 0	🟥 0	🟩 1	🟪 5	4	2	3
File 3	🟩 1	3	🟥 0	🟥 0	2	🟪 4	🟥 0
...	...	...	...	...	...	...	...
File k	3	🟩 1	🟪 6	2	🟥 0	4	5

Fig. 4 An example of our solution representation. Red: this reviewer is not recommended to review the file; green: the most appropriate reviewer for the file; and purple: recommended, but the least appropriate reviewer for the file.

review to approve code quickly to meet release deadlines especially when associated with low expertise. The multi-objective approach proposed in this paper generates as an output a set of non-dominated solutions (Pareto front). It is up to the team manager to select the reviewers assignment solution based on their preferences. If the team prefers close connection then the selected/preferred solution from the Pareto front will be in the region of interest where the objective of collaborations is high otherwise the selected solution will be in the area of the Pareto front where the value of collaboration is low. Our goal is to provide a diverse set of good reviewers assignment solutions rather than only one solution then the user can select the preferred one based on his/her preferences.

We present in the following our three fitness functions: availability, expertise and collaborations.

Availability The availability is the inverse of the estimated wait until reviewers that are selected to work on a selected set of file S become available. In our case, the waiting period is deducted from the workload that the reviewer has. We considered the workload as the combination of the number of commits submitted recently (during the last 7 days) and the total number of files for all open pull-requests.

$$Availability = \frac{1}{\sum_{k=1}^{P} \sum_{i=1}^{M} a_i * S[k,i]}, s_{k,i} > 0 \qquad (3)$$

Where $a = \{a_1, a_2, \ldots, a_M\}$ is an array that contains the tasks queued for a reviewer. a_i represents the number of tasks in the queue for the reviewer r_i. P is total number of files requested to be reviewed and M is total number of reviewers working on the project.

Expertise Considering File Priority PR is a vector of weights that defines how urgently a file needs to be reviewed. For a file f_k, the priority score will take 1 if the tag "priority" is used in the pull-request, otherwise, the priority will be 0. We used both FR and PR to formulate the reviewer expertise as an objective.

$$Expertise = \sum_{k=1}^{P} \sum_{i=1}^{M} \frac{FR[k,i] + PR[k]}{S[k,i]}, s_{k,i} > 0 \tag{4}$$

Where M is total number of reviewers working on the project and P is total number of developers working on the project. FR is a File-Reviewer matrix and $S[k,i]$ represents the rank of the reviewers in the solution S. In fact, We are ranking the reviewers from 0 to P. For instance, if we have $P = 7$ developers (potential reviewers), a reviewer with rank 2 would be more appropriate than a reviewer with rank 4 to review the assigned file.

Both fitness functions "availability" and "expertise" are to be maximized. Thus, a lower rank (more suitable reviewer) would result in a higher fitness function (availability or expertise) since the rank ($S[k,i]$) is in the denominator. Therefore, the top ranked developers with high expertise/availability would be more likely to survive for the next evaluations of the multi-objective algorithm.

Collaboration Collaboration is computed as the sum of all connections between recommended reviewers selected to work with a selected set of developers:

$$Collaboration = \sum_{k=1}^{N} \sum_{j=1}^{P} \sum_{i=1}^{M} DR[j,i] * FD[k,j] * (S[k,j] > 0) \tag{5}$$

Where $(s[k,j] > 0)$ is a binary mask for $S[k,j]$, meaning each entry with value 0 will remain 0 and each entry with value greater than 0 will become 1. P is total number of files requested to be reviewed, M is total number of reviewers working on the project and N is total number of developers working on the project. DR is a Developer-Reviewer matrix and FD is a File-Developer matrix where $FD[i,j]$ represents the number of times that the developer i worked on the file j. Therefore, the developer who changed the file under review (one or many times) can be assigned as a reviewer. The two matrix DR and FD are created during the data extraction step.

3.3.3 Change Operators

We applied single point crossover and swap mutation to explore and exploit the search space. Regarding crossover, we deploy a single random cut-point crossover. This operator is performed by generating a random crossover point. The cut-point is a binary block from crossover point K, which is a row-index and a column- index of a solution, to the end of the solution is copied from one parent, the rest is copied from the second parent. Then, it exchanges the subsequences before and after K between two parent individuals to create two offspring. In case we generate any infeasible offspring we apply a repair mechanism.

Our mutation—bit inversion changes the new offspring by swapping two rows in the matrix of the solution. Mutation can occur at each row in the matrix with some probability. The purpose of mutation is to prevent all solutions in the population falling into a local optimum.

4 Experiment and results

To evaluate our approach for recommending relevant peer reviewers, we conducted a set of experiments based on different versions of 9 open source systems. Due the stochastic nature of search algorithms, each experiment was repeated 30 times and the results were subsequently and statistically analyzed with the aim of comparing our multi-objective approach with both a mono-objective search technique based on an aggregation of expertise and collaborations [30] and also all the three objectives (AEC GA), and existing tools not based on heuristic search cHRev[45], REVFINDER[37], and ReviewBot[4] that only use expertise models without considering collaborations and availability of peer reviewers. Furthermore, we conducted an ablation study to compare our approach with three multi-objective variants considering two out of the three objectives (AC NSAG-II, AE NSGA-II and EC NSGA-II). All these existing studies were already evaluated in the literature on the same projects considered in this validation and the associated data is available thus we did not find a need to re-implement them. In this section, we present our research questions followed by experimental settings and parameters. Finally, we discuss our results for each of those research questions.

4.1 Research Questions

We focused on the following three research questions to evaluate the efficiency of our approach:

- **RQ1.** (Efficiency) Can the proposed approach precisely identify relevant peer reviewers?
- **RQ2.** (Comparison to search-based techniques) Does the proposed multi-objective approach perform significantly better than an existing mono-objective formulation aggregating expertise and collaboration [30], a mono-objective aggregation of all the three objectives (AEC GA) and variants of our multi-objective search considering two out of the three objectives (NSGA-II, AE NSGA-II and EC NSGA-II)?
- **RQ3.** (Comparison to state-of-the-art) Does our approach perform significantly better than existing peer reviewer recommendation techniques not based on heuristic search?

To answer RQ1, we validated the proposed multi-objective technique on 9 medium to large-size open-source systems, as detailed in the next section, to evaluate the correctness of our code-reviewer recommendation framework.

To ensure a fair comparison with existing techniques, we followed a similar evaluation procedure by taking the most recent 1000 reviews and the reviewers assigned to these pull-requests as the ground truth. We built the different expertise, availability and collaborations models based on the review data just before the pull-request to evaluate in order to assign peer reviewers. We used GitHub API to extract the information about the pull request. From the information extracted, there is a tag 'reviewer' which contains the name of the reviewer. The name of the reviewer is also extracted from the comments under the pull request and this information is also provided by GitHub API.To this end, we used the following evaluation metrics:

– **Precision@k** denotes the number of correct recommended peer reviewers in the top k of recommended ones by the solution divided by the total number of peer reviewer recommendations to inspect.
– **Recall@k** denotes the number of correct recommended peer reviewers in the top k of recommended ones by the solution divided by the total number of expected reviewers to be recommended based on the ground truth.
– **MMR@k** measures the mean reciprocal rank which is an average rank of correct reviewers in the recommendation list. The higher the value the better.

Since the number of involved reviewers in each pull-request evaluation is limited in general to a few developers, we calculate these precision and recall metrics with different k values, 1, 3, 5 and 10.

To answer RQ2, we compared, using the above metrics, the performance of our multi-objective approach with an existing mono-objective formulation, based on a Genetic Algorithm, aggregating the two objectives of expertise and collaboration into one objective as the sum of them with equal weight [30]. We selected that mono-objective approach since it is the closest one to our work and already outperformed random search and other metaheuristic algorithms (simulated annealing and Particle Swarm Optimization) based on the results presented in [30]. Furthermore, we implemented a mono-objective approach aggregating all the three objectives (AEC GA) in one fitness function to evaluate the impact of adding the availability objective on the quality of the results by comparing with [30]. In addition, we compared different variants of our multi-objective approach including only two out of the three objectives (NSGA-II AE, AC and EC) to evaluate the contribution of each objective to the quality of the assignment results. The comparison between NSGA-II EC and the mono-objective search using only expertise and collaboration [30] can confirm the impact of the conflicting nature of the two objectives on the quality of the results.

To answer RQ3, we compared our multi-objective approach to different existing techniques not based on heuristic search:

– REVFINDER [37] uses the paths of the files to be reviewed to find reviewers who evaluated files in the same location.
– cHRev [45] is a hybrid approach using the frequency and recency of the history of the reviews to find relevant peer reviewers.

– ReviewBot [4] uses static analysis tools to find experienced reviewers

We limited the evaluation in RQ2 and RQ3 to Android, OpenStack, and Qt to ensure a fair comparison based on an existing benchmark [30, 37, 42]. More details about these projects will be presented in the next section.

4.2 Studied Projects

As described in Table 1, we used a data set of 9 open-source systems including 3 projects (OpenStack, Android and Qt) from existing code review benchmarks [30, 37, 42]. We used our tool to collect the data about Atomix, Tablesaw, Vavr, Takes, Dkpro-core, and Pac4j. In fact, our tool is implemented in a way that it takes a link to the project repository on GitHub and extracts all the needed data automatically similar to the existing public dataset for OpenStack, Android and Qt. To collect the data, we used GitHub API to send multiple queries to GitHub to get the needed information about the project under study. Actually, GitHub API provides different queries to extract the information about the pull requests, its reviewers, its changed files and all the committer names. The response to each query is a JSON file. Thus, we had to perform some cleaning and extracting steps to keep only the needed pieces of information.

– **Atomix:** A fault-tolerant distributed coordination framework.
– **Tablesaw:** A data science platform that includes a data-frame, an embedded column store, and hundreds of methods to transform, summarize, or filter data.
– **Vavr:** A functional component library that provides persistent data types and functional control structures.
– **Takes:** Opinionated web framework which is built around the concepts of True Object-Oriented Programming and immutability.
– **Dkpro-core:** A collection of reusable NLP tools for linguistic pre-processing, machine learning, lexical resources, etc.
– **Pac4j:** A security engine.
– **Android:** A software stack for mobile devices developed by Google.
– **OpenStack:** A large platform for cloud computing to manage a data-center.
– **Qt:** A widget toolkit for creating graphical user interfaces.

Table 1 shows statistics for the analyzed systems including the number of reviewers, the number of reviews in a project, the size, etc. All collected reviews are from closed pull-requests and contain at least one file. We selected these open source projects for our experiments since they contain a large number of code reviews and they have been studied in the software review literature [4, 37, 45] to ensure a fair comparison with the current state of the art.

Table 1 Summary of Studied Systems

Project (Studied Period)	Number of classes	Number of reviewers	Number of files	Number of reviews
Atomix (04/2017-11/2018)	1459	136	182280	4237
Tablesaw (06/2016-03/2018)	224	12	52837	1930
Vavr (04/2016-08/2018)	301	123	126683	4188
Takes (07/2015-05/2018)	472	264	50369	2687
Dkpro-core (03/2015-08/2018)	376	411	54695	4564
Pac4j (08/2014-10/2017)	302	29	31916	2282
Android (10/2008-01/2012)	563	94	26840	5126
OpenStack (07/2011-05/2012)	539	82	16953	6586
Qt (05/2011-05/2012)	782	202	78401	23810

4.3 Parameter Tuning and Statistical Tests

Since metaheuristic algorithms are stochastic optimizers, they can provide different results for the same problem instance from one run to another. For this reason, our experimental study was performed based on 30 independent simulation runs for each problem instance and the obtained results were statistically analyzed using the Friedman test with a 95% confidence level ($\alpha = 5\%$). Since the Friedman test results were significant, we used the Wilcoxon rank sum test [39] in a pairwise fashion (AEC NSGA-II versus each of the competitor approaches) in order to detect significant performance differences between the algorithms under comparison based on 30 independent runs. For deterministic techniques, we did not perform 30 independent runs. The Wilcoxon test allows testing the null hypothesis H0 that states that both algorithms medians' values for a particular metric are not statistically different against H1 which states the opposite. The Wilcoxon test does not require that the data sets follow a normal distribution since it operates on values' ranks instead of operating on the values themselves. Since we are comparing more than two different algorithms, we performed several pairwise comparisons based on Wilcoxon test to detect the statistical difference in terms of performance. To compare two algorithms based on a particular metric, we record the obtained metric's values for both algorithms over 30 runs. For deterministic techniques, we considered one value of each metric on each system. After that, we compute the metric's median value for each algorithm. Besides, we executed the Wilcoxon test with a 95% confidence level ($\alpha = 5\%$) on the recorded metric's values using the Wilcoxon MATLAB routine. If the returned p-value is less than 0.05 then we reject H0 and we can state that one algorithm outperforms the other, otherwise we cannot say anything in terms of performance difference between the two algorithms.

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

An often-omitted aspect in metaheuristic search is the tuning of algorithm parameters. In fact, parameter setting significantly influences the performance of a search algorithm on a particular problem. For this reason, for each search algorithm and each system, we performed a set of experiments using several population sizes: 10, 20, 30, 40 and 50. The stopping criterion was set to 100,000 fitness evaluations for all search algorithms to ensure fairness of comparison. We used a high number of evaluations as a stopping criterion since our approach requires multiple objectives. Each algorithm was executed 30 times with each configuration and then the comparison between the configurations was performed based on different metrics described previously using the Friedman test. The other parameter values were fixed by trial and error and are as follows: (1) crossover probability = 0.5; mutation probability = 0.4 where the probability of gene modification is 0.2. We used the same parameters of the existing work of Ouni et al., called RevRec, [30] for a fair comparison.

4.4 Results

Results for RQ1. The results of Tables 2-3 and Figure 5 confirm the efficiency of our multi-objective approach, based on NSGA-II, to identify relevant peer reviewers for pull-requests from all the 9 open source systems. Tables 2 and 3 show the average precision@k and recall@k results of our NSGA-II AEC technique on the various systems, with k equal to 1, 3, 5 and 10. For example, most of the recommended peer reviewers in the top 3 (k=3) are relevant (compared to the expected results) with precision over 60% on all the 9 systems. The lowest precision is around 47% for k=10 which still could be considered acceptable due to a large number of possible reviewers in the selected systems.

In terms of recall, Table 3 confirms that the majority of the expected peer reviewers to recommend are located in the top 10 (k=10) with a recall score over 53%. The highest recall is 78% for k=10 (Qt project). Since several pull-requests may require more than one peer reviewer, most of the highest recall scores are obtained for k=5 and k =10.

Figure 5 shows that NSGA-II was able to efficiently rank the recommended peer-reviewers. In fact, the median MMR on the different systems is higher than 68% with the highest score of 79% for the Open Stack project. This outcome is important since the efficient ranking of the recommended peer reviewer is one of the main motivations of our approach that consider not only the expertise but also the availability and the collaborations among reviewers. The availability in our case is considered based on the number of commits and files that a programmer is working on in the time period closest to the evaluated pull-request. We noticed that our technique does not have a bias toward the evaluated system. We had almost consistent average scores of precision, recall and the mean reciprocal rank.

Table 2 Median Precision@k results for the search algorithms (multi-objective variants) including RevRec (mono-objective search) on all the systems based on 30 runs. All the results are statistically significant using the Friedman test and Wilcoxon with a 95% confidence level ($\alpha = 5\%$)

Project	k	**Precision@k**					
		AEC (NSGA-II)	RevRec (GA)	AEC (GA)	AC (NSGA-II)	AE (NSGA-II)	EC (NSGA-II)
Atomix	1	0.62	0.56	0.60	0.52	0.58	0.60
	3	0.58	0.44	0.47	0.41	0.44	0.51
	5	0.52	0.38	0.43	0.36	0.40	0.47
	10	0.47	0.41	0.41	0.38	0.41	0.45
Tablesaw	1	0.57	0.49	0.54	0.44	0.52	0.54
	3	0.64	0.52	0.56	0.41	0.52	0.60
	5	0.61	0.44	0.51	0.38	0.48	0.56
	10	0.55	0.41	0.46	0.40	0.44	0.50
Vavr	1	0.62	0.53	0.56	0.46	0.53	0.58
	3	0.58	0.47	0.52	0.41	0.44	0.54
	5	0.64	0.56	0.59	0.47	0.52	0.61
	10	0.66	0.51	0.56	0.44	0.53	0.60
Takes	1	0.57	0.48	0.52	0.42	0.50	0.52
	3	0.62	0.56	0.59	0.48	0.52	0.59
	5	0.55	0.46	0.50	0.40	0.43	0.52
	10	0.53	0.44	0.47	0.37	0.44	0.50
Dkpro-core	1	0.63	0.52	0.56	0.41	0.50	0.59
	3	0.57	0.47	0.51	0.34	0.43	0.54
	5	0.66	0.55	0.59	0.42	0.55	0.61
	10	0.59	0.43	0.49	0.37	0.47	0.52
Pac4j	1	0.61	0.52	0.56	0.41	0.54	0.58
	3	0.56	0.43	0.47	0.38	0.45	0.49
	5	0.59	0.39	0.46	0.33	0.42	0.51
	10	0.54	0.42	0.46	0.36	0.40	0.49
Android	1	0.68	0.58	0.62	0.51	0.60	0.64
	3	0.62	0.47	0.53	0.44	0.51	0.56
	5	0.53	0.39	0.43	0.37	0.41	0.45
	10	0.47	0.34	0.39	0.31	0.36	0.41
OpenStack	1	0.72	0.59	0.64	0.52	0.61	0.64
	3	0.61	0.51	0.54	0.46	0.52	0.56
	5	0.64	0.43	0.5	0.39	0.48	0.52
	10	0.54	0.36	0.39	0.33	0.36	0.43
Qt	1	0.58	0.49	0.51	0.46	0.47	0.53
	3	0.61	0.45	0.50	0.43	0.43	0.55
	5	0.54	0.41	0.45	0.39	0.38	0.48
	10	0.46	0.34	0.39	0.31	0.32	0.39

Results for RQ2. Tables 2-3 and Figure 5 confirm that our multi-objective approach (AEC NSGA-II) is better, on average, than the existing mono-objective technique, RevRec [30], based on the 3 metrics of precision, recall and MMR on all the 9 systems. The median precision and recall values of the RevRec tool on the 9 systems are lower than 56% as described in Table 2 for all values of k (1, 3, 5 and 10). Furthermore, the EC NSGA-II variant of our approach outperformed the mono-objective search aggregating the same objectives [30] based on the metrics on almost all the systems. Thus, an interesting observation is the clear conflicting objectives of expertise and collaborations which confirms our observation in the eBay survey that collaborations does not mean qualified reviewers (with high expertise) are assigned to review the pull-requests. The same observation is valid for the ranking of recommended peer reviewers based on the MMR measure as described in Figure 5. For in-

Table 3 Median Recall@k results for the search algorithms (multi-objective variants) including RevRec (mono-objective search) on all the systems based on 30 runs. All the results are statistically significant using the Friedman test and Wilcoxon test with a 95% confidence level ($\alpha = 5\%$)

Project	k	**Recall@k**					
		AEC (NSGA-II)	RevRec (GA)	AEC (GA)	AC (NSGA-II)	AE (NSGA-II)	EC (NSGA-II)
Atomix	1	0.56	0.43	0.48	0.39	0.46	0.51
	3	0.52	0.39	0.44	0.36	0.44	0.47
	5	0.61	0.46	0.53	0.41	0.50	0.58
	10	0.58	0.34	0.45	0.39	0.43	0.56
Tablesaw	1	0.51	0.43	0.48	0.37	0.46	0.48
	3	0.55	0.41	0.46	0.36	0.43	0.52
	5	0.52	0.38	0.44	0.35	0.40	0.50
	10	0.59	0.33	0.50	0.36	0.42	0.56
Vavr	1	0.53	0.41	0.48	0.38	0.43	0.50
	3	0.62	0.39	0.52	0.35	0.46	0.59
	5	0.55	0.42	0.50	0.40	0.44	0.52
	10	0.59	0.38	0.46	0.34	0.41	0.54
Takes	1	0.49	0.41	0.46	0.38	0.44	0.46
	3	0.53	0.44	0.47	0.39	0.42	0.50
	5	0.62	0.37	0.43	0.31	0.40	0.59
	10	0.66	0.34	0.51	0.32	0.39	0.62
Dkpro-core	1	0.54	0.47	0.44	0.40	0.42	0.51
	3	0.51	0.41	0.46	0.39	0.43	0.48
	5	0.58	0.39	0.49	0.36	0.46	0.53
	10	0.67	0.35	0.59	0.31	0.56	0.63
Pac4j	1	0.56	0.41	0.49	0.38	0.44	0.53
	3	0.62	0.36	0.53	0.31	0.50	0.58
	5	0.51	0.31	0.39	0.28	0.35	0.47
	10	0.63	0.38	0.49	0.31	0.47	0.60
Android	1	0.57	0.38	0.51	0.36	0.48	0.54
	3	0.72	0.51	0.63	0.48	0.60	0.67
	5	0.76	0.61	0.66	0.53	0.63	0.71
	10	0.79	0.71	0.77	0.66	0.71	0.77
OpenStack	1	0.59	0.41	0.49	0.38	0.45	0.56
	3	0.68	0.54	0.62	0.51	0.60	0.65
	5	0.76	0.61	0.68	0.53	0.64	0.72
	10	0.81	0.74	0.77	0.68	0.69	0.77
Qt	1	0.56	0.41	0.48	0.38	0.43	0.50
	3	0.66	0.50	0.58	0.47	0.50	0.61
	5	0.68	0.59	0.63	0.53	0.61	0.63
	10	0.76	0.65	0.68	0.57	0.65	0.71

stance, the MMR score for AEC NSGA-II is 78% on the Takes project while it is limited to 61% for RevRec.

The outperformance of NSGA-II can be explained as well by the consideration of the new objective of availability which may reflect the reality of how peer reviewers are manually assigned to reduce delays. In fact, the aggregation of all the three objectives in a mono-objective search (AEC GA) is performing better than [30] which confirms the positive contribution of the availibility objective on the quality of the results. The least performance of our multi-objective approach in terms of MMR (slightly less than RevRec) was observed for the Dkpro-core and pac4j projects. While investigating the reasons behind this decreased performance, we found out that the main reason is that these projects have a large enough number of contributors comparing to their sizes(in terms of files, commits and pull-request). In fact, the ratio *'contributors to size'* is larger than the other projects. Thus, the availability objective may not represent a big concern for these projects unlike the others since they have enough contributors to review the changed files/pull-requests.

All these results were statistically significant on 30 independent runs using the Friedman test and Wilcoxon test (pairwise comparison) with a 95% confidence level ($\alpha < 5\%$). We also found the results of the Vargha Delaney A_{12} statistic are higher than 0.8 (large) on all the systems which confirms the significant outperformance of AEC NSGA-II comparing to the mono-objective formulation. The detailed effect size results can be found in Tables 4 and 5.

Table 4 The effect size for Precision based on 30 runs when comparing AEC NSGA-II versus each of the search algorithms.

Project	Effect Size-RevRec (GA)	Effect Size-AEC (GA)	Effect Size-AC (NSGA-II)	Effect Size-AE (NSGA-II)	Effect Size-EC (NSGA-II)
Atomix	0.52	0.61	0.82	0.76	0.58
Tablesaw	0.39	0.72	0.79	0.73	0.63
Vavr	0.87	0.63	0.86	0.78	0.71
Takes	0.64	0.68	0.91	0.83	0.68
Dkpro-core	0.92	0.77	0.83	0.71	0.72
Pac4j	0.86	0.72	0.72	0.84	0.66
Android	0.52	0.64	0.77	0.92	0.74
OpenStack	0.76	0.68	0.84	0.81	0.63
Qt	0.94	0.71	0.92	0.83	0.61

Table 5 The effect size for Recall based on 30 runs when comparing AEC NSGA-II with each of the search algorithms.

Project	Effect Size-RevRec (GA)	Effect Size-AEC (GA)	Effect Size-AC (NSGA-II)	Effect Size-AE (NSGA-II)	Effect Size-EC (NSGA-II)
Atomix	0.64	0.66	0.83	0.72	0.61
Tablesaw	0.82	0.62	0.75	0.69	0.53
Vavr	0.93	0.71	0.83	0.77	0.64
Takes	0.72	0.63	0.91	0.82	0.68
Dkpro-core	0.89	0.74	0.84	0.91	0.59
Pac4j	0.74	0.61	0.88	0.73	0.71
Android	0.91	0.77	0.94	0.68	0.63
OpenStack	0.83	0.82	0.83	0.73	0.69
Qt	0.72	0.64	0.86	0.77	0.57

Results for RQ3. Since it is not sufficient to compare our approach with just search-based algorithms, we compared the performance of NSGA-II to three different peer reviewer recommendation techniques which are not based on heuristic search, as described in Tables 6 and 7, and Figure 6.

Similar to the comparison with RevRec, we used the precision@k, recall@k and MMR measures with k ranging from 1 to 10. NSGA-II achieves better results, on average than the other three methods on all the three projects. For example, our approach achieved a Precision@k median of 63%, 59%, 48% and 43% are achieved for k= 1, 3, 5 and 10 respectively as described in Table 6. In comparison, CHrev achieved a median Precision@k of 58%, 47%, 39%, and 34% are obtained for k= 1, 3, 5 and 10. CHRev has the highest precision among all the remaining tools of REVFINDER and ReviewBot. Similar observations are also valid for the recall@k and MMR.

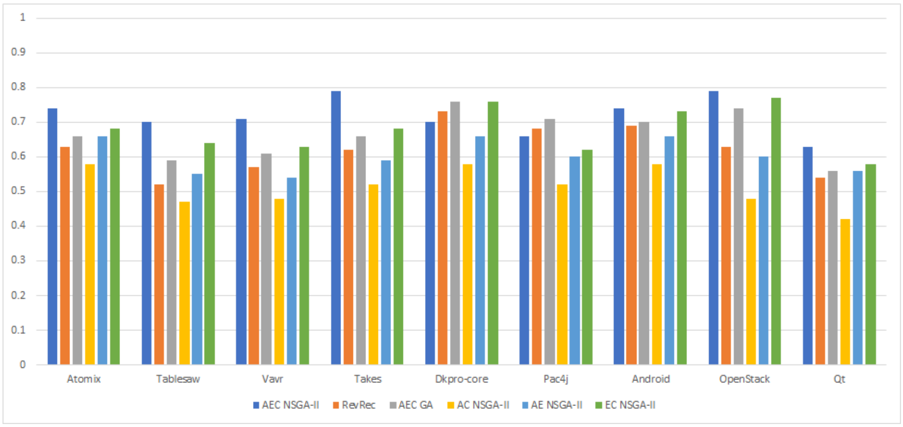


Fig. 5 Median MMR results for the different search algorithms on all systems based on 30 runs. All the results are statistically significant using the Friedman test with a 95% confidence level ($\alpha = 5\%$)

Table 6 Median Precision@k results for all the approaches on three systems based on 30 runs. All the results are statistically significant using the Friedman test and Wilcoxon test with a 95% confidence level ($\alpha = 5\%$)

Project	K	**Precision@k**								
		ACE (NSGA-II)	AEC (GA)	AC (NSGA-II)	AE (NSGA-II)	EC (NSGA-II)	RevRec (GA)	cHRev	REVFINDER	ReviewBot
Android	1	0.68	0.62	0.51	0.60	0.64	0.58	0.50	0.34	0.21
	3	0.62	0.53	0.44	0.51	0.56	0.47	0.35	0.25	0.17
	5	0.53	0.43	0.37	0.41	0.45	0.39	0.30	0.22	0.12
	10	0.47	0.39	0.31	0.36	0.41	0.34	0.26	0.18	0.09
OpenStack	1	0.72	0.64	0.52	0.61	0.64	0.59	0.48	0.32	0.24
	3	0.61	0.54	0.46	0.52	0.56	0.51	0.42	0.27	0.20
	5	0.64	0.50	0.39	0.48	0.52	0.43	0.38	0.25	0.16
	10	0.54	0.39	0.33	0.36	0.43	0.36	0.31	0.21	0.11
Qt	1	0.58	0.51	0.46	0.47	0.53	0.49	0.45	0.30	0.22
	3	0.61	0.50	0.43	0.43	0.55	0.45	0.40	0.27	0.19
	5	0.54	0.45	0.39	0.38	0.48	0.41	0.37	0.21	0.13
	10	0.46	0.39	0.31	0.32	0.39	0.34	0.31	0.16	0.09

5 Threats to validity

Conclusion validity is concerned with the statistical relationship between the treatment and the outcome. We addressed conclusion threats to validity by performing 30 independent simulation runs for each problem instance and statistically analyzing the obtained results using the Friedman test with a 95% confidence level ($\alpha = 5\%$). However, the parameter tuning of the different optimization algorithms used in our experiments creates another internal threat that we need to evaluate in our future work. The parameter values used in our experiments were determined by trial-and-error [23]. In addition, the estimation of the availability of reviewers on open source systems may not be very accurate.

Construct validity is concerned with the relationship between theory and what is observed. The definition of expertise and collaborations can be subjective and hard to formalize thus further empirical studies are required to validate the different metrics used in our work. We are planning to consider

Table 7 Median Recall@k results for all the approaches on three systems based on 30 runs. All the results are statistically significant using the Friedman test and Wilcoxon test with a 95% confidence level ($\alpha = 5\%$)

Project	K	**Recall@k**								
		ACE (NSGA-II)	AEC (GA)	AC (NSGA-II)	AE (NSGA-II)	EC (NSGA-II)	RevRec (GA)	cHRev	REVFINDER	ReviewBot
Android	1	0.57	0.51	0.36	0.48	0.54	0.38	0.27	0.18	0.11
	3	0.72	0.63	0.48	0.60	0.67	0.51	0.50	0.39	0.19
	5	0.76	0.66	0.53	0.63	0.71	0.61	0.61	0.48	0.29
	10	0.79	0.77	0.66	0.71	0.77	0.71	0.65	0.54	0.38
OpenStack	1	0.59	0.49	0.38	0.45	0.56	0.41	0.31	0.15	0.12
	3	0.68	0.62	0.51	0.60	0.65	0.54	0.39	0.29	0.20
	5	0.76	0.68	0.53	0.64	0.72	0.61	0.52	0.37	0.32
	10	0.81	0.77	0.68	0.69	0.77	0.74	0.66	0.46	0.39
Qt	1	0.56	0.48	0.38	0.43	0.50	0.41	0.33	0.14	0.90
	3	0.66	0.58	0.47	0.50	0.61	0.50	0.47	0.27	0.16
	5	0.68	0.63	0.53	0.61	0.63	0.59	0.52	0.35	0.24
	10	0.76	0.68	0.57	0.65	0.71	0.65	0.60	0.43	0.30

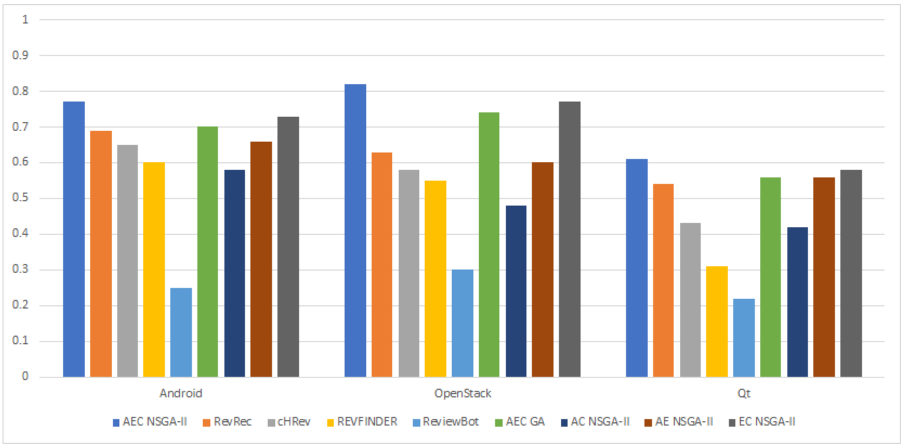


Fig. 6 Median MMR results for all the approaches on three systems based on 30 runs. All the results are statistically significant using the Friedman test and Wilcoxon test with a 95% confidence level ($\alpha = 5\%$)

other possible formations as part of our future work and compare between them. Additionally, our current definition of the availability needs further improvement. In fact, reviewers can be assigned other types of development activities than coding (e.g., testing, design/architecture, requirements analysis, etc.). The data about these activities are not always available. However, the formulation of our fitness function is easy to modify in a way that enables managers to enter the number of tasks per reviewer, especially the ones that they are beyond code reviews.

External validity refers to the generalizability of our findings. In this study, we performed our experiments on different widely used open-source systems belonging to different domains and having different sizes. However, we cannot assert that our results can be generalized to other systems. Future replications of this study are necessary to confirm our results with a larger set of pull requests and reviewers.

Another threat to our approach could be the effort required by the manager to select the preferred solution. In general, the preferences are defined based on the current context such as: the urgency to release code quickly, available resources, speedy growth phase of the project, etc. These different contexts are not changing daily and they are not related to only one or few pull-requests but they are more related to the situation of the whole project. To mitigate this threat, we provide the distribution of the solutions of the Pareto front which can be ranked based on the preferred fitness functions or based on the current context. Thus, the preferred solution can be selected in an easier and faster way.

6 Related work

Expertise has been the most important factor in the studies proposing peer reviewer recommendation. Zanjani et al. found that expertise changes over time and thus both frequency and recency of reviews must be accounted for to find the most appropriate reviewers. Therefore their approach builds a reviewer expertise model, generated from past reviews, that combines a quantification of review comments and their recency [46].

Balachandran et al. first suggested to use the Review Bot tool, as a recommendation system to reduce human effort and improve review quality by finding source code issues, which need to be addressed, but could be missed during reviewer inspection. The bot can review the code by integrating the static analysis of the source code [4]. The bot, as part of a review process, is able to recommend the most appropriate human reviewer. In cases when the project has been modified frequently and there is a history of the changes for the source code, the bot is a suitable solution. However, Patanamon et al. [36] showed that the Review Bot's algorithm had poor performance on other projects with no or little change in their files due to the lack of history in line-by-line source code. In the same work, they introduced the idea of using file location (but not content) as an indicator for similarity of reviews. Their reviewer recommender approach, called File Path Similarity (FPS), implementing this idea, assumes that files that are located in similar file paths would be managed and reviewed by similarly experienced expert code reviewers. To improve their previous idea, Patanamon et al. [37] introduced REVFINDER, a file location-based code-reviewer recommendation approach. REVFINDER uses the similarity of previously reviewed file paths to recommend an appropriate code-reviewer. However, they did not consider the reviewer's work load and availability.

Xia et al. [41] used bug reports and developer information to recommend developers to resolve bugs. However, the most notable limitation of these works is that the socio-technical aspect of the code review process is not considered.

Several other studies focused on human factors and socio-technical aspects of code review. Cohen et. al., in [13] discuss that code review is a complex process involving both social and personal aspects. Fagan [18], to ensure the quality of software, introduced software inspection as a systematic peer review activity. Other studies [7–9, 28, 34, 35, 35, 43] motivate the need for a peer review recommendation system, considering the volunteer nature of open-source software (OSS) developers and the peer review structure, suggest that different human factors influence the OSS peer review. Baysal et al. conducted several studies [5, 6, 28] to explore the relationships between a set of personal and social factors and code review.

Bosu et al. [8] conducted a survey on four aspects of peer impression formation: trust, reliability, perception of expertise, and friendship. They concluded that there is a high level of trust, reliability, perception of expertise, and friendship between OSS peers who have participated in code review for a period of time. In another survey on how social interaction networks influence peer impressions formation [10], they found that code review interactions have the most favorable characteristics to support impression formation among OSS participants.

Based on search based software engineering [2,20,21,29,38], Ouni et. al [30] combined both aspects in their proposed approach, called RevRec, to provide decision-making support for code change submitters and reviewer assigners to identify the most appropriate peer reviewers for code changes. RevRec uses a genetic algorithm to assign reviewers to review a code change based on expertise and history of collaboration. Their single objective optimization approach aims to find appropriate reviewers for a given patch based on the reviewer's expertise with the submitted patch files, and the reviewer's prior collaborations with the review request submitter. Although this is the closest work in the literature to our proposed approach, our work differs from their work in a few ways: their solution representation determines if any of the reviewers are recommended to review a single file, therefore in cases when there are more files to review, let say k files, then the single objective optimization must run k times independently from each other which may not necessarily match the reality of the task. Our solution representation recommends reviewers for all the files that need to be reviewed at the same time. Furthermore, they do not consider the current workload of the reviewers and when they might be available to review the current files that match their expertise. In our method, we account for a reviewer's availability and we provide a ranking for the recommended reviewers so that if one reviewer is the best match, but busy with other work, we do not recommend the reviewer as the first choice for reviewing that file. This will decrease the overall delay in the system for files to get reviewed. Additionally, to capture the complexity of peer code review task, we formulate the problem as interaction among the competing objectives of expertise, availability and history of collaborations. More comprehensive studies on search-based software maintenance can be found in [27],[19],[12, 21, 24, 26].

7 Conclusion

In this paper we formulated the recommendation of peer code reviewers as a multi-objective problem to find a trade-off between the competing objectives of expertise, availability and history of collaborations. Unlike existing approaches, our approach can sacrifice expertise to avoid a delay caused by limited resources (e.g. low peer reviewer availability). Our evaluation results confirm the efficiency of our multi-objective approach on 9 open source projects in finding better reviewer recommendations, as compared to the state of the art [30]. Furthermore, our survey with practitioners highlighted the importance of managing code reviews to reduce delays while ensuring high expertise as much as possible.

As part of our future work, we plan to consider the use of additional projects and feedback. Furthermore, we will extend our collaboration model of code reviews beyond the history of data from a single project. We are also planing to extend the definition of the expertise by taking into consideration the recency and the quality of past reviews. Since there is a lack of empirical evidence on how to define *"good quality"* in code reviews, we are planning to perform a rigorous empirical study via conducting extensive surveys to answer this subjective question.

References

1. Almhana, R., Mkaouer, W., Kessentini, M., Ouni, A.: Recommending relevant classes for bug reports using multi-objective search. In: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, pp. 286–295. ACM, New York, NY, USA (2016). DOI 10.1145/2970276.2970344. URL `http://doi.acm.org/10.1145/2970276.2970344`
2. Amal, B., Kessentini, M., Bechikh, S., Dea, J., Said, L.B.: On the use of machine learning and search-based software engineering for ill-defined fitness function: a case study on software refactoring. In: International Symposium on Search Based Software Engineering, pp. 31–45. Springer, Cham (2014)
3. Bacchelli, A., Bird, C.: Expectations, outcomes, and challenges of modern code review. In: Proceedings of the 2013 international conference on software engineering, pp. 712–721. IEEE Press (2013)
4. Balachandran, V.: Reducing human effort and improving quality in peer code reviews using automatic static analysis and reviewer recommendation. In: Proceedings of the 2013 International Conference on Software Engineering, pp. 931–940. IEEE Press (2013)
5. Baysal, O., Holmes, R.: A qualitative study of mozilla's process management practices. David R. Cheriton School of Computer Science, University of Waterloo, Waterloo, Canada, Tech. Rep. CS-2012-10 (2012)
6. Baysal, O., Kononenko, O., Holmes, R., Godfrey, M.W.: The influence of non-technical factors on code review. In: 2013 20th Working Conference on Reverse Engineering (WCRE), pp. 122–131. IEEE (2013)
7. Bird, C., Pattison, D., D'Souza, R., Filkov, V., Devanbu, P.: Latent social structure in open source projects. In: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering, pp. 24–35. ACM (2008)
8. Bosu, A., Carver, J.C.: Impact of peer code review on peer impression formation: A survey. In: 2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, pp. 133–142. IEEE (2013)

9. Bosu, A., Carver, J.C.: How do social interaction networks influence peer impressions formation? a case study. In: IFIP International Conference on Open Source Systems, pp. 31–40. Springer (2014)
10. Bosu, A., Carver, J.C.: Impact of developer reputation on code review outcomes in oss projects: An empirical investigation. In: Proceedings of the 8th ACM/IEEE international symposium on empirical software engineering and measurement, p. 33. ACM (2014)
11. Bosu, A., Carver, J.C., Bird, C., Orbeck, J., Chockley, C.: Process aspects and social dynamics of contemporary code review: Insights from open source development and industrial practice at microsoft. IEEE Transactions on Software Engineering **43**(1), 56–75 (2016)
12. Boussaa, M., Kessentini, W., Kessentini, M., Bechikh, S., Chikha, S.B.: Competitive coevolutionary code-smells detection. In: International Symposium on Search Based Software Engineering, pp. 50–65. Springer, Berlin, Heidelberg (2013)
13. Cohen, J., Brown, E., DuRette, B., Teleki, S.: Best kept secrets of peer code review. Smart Bear Somerville (2006)
14. Committee, S.E.S., et al.: IEEE standard for software reviews. IEEE Std pp. 1028–1997 (1997)
15. Deb, K., Gupta, S.: Understanding knee points in bicriteria problems and their implications as preferred solution principles. Engineering optimization **43**(11), 1175–1204 (2011)
16. Deb, K., Pratap, A., Agarwal, S., Meyarivan, T.: A fast and elitist multiobjective genetic algorithm: Nsga-ii. IEEE transactions on evolutionary computation **6**(2), 182–197 (2002)
17. Emmerich, M.T., Deutz, A.H.: A tutorial on multiobjective optimization: fundamentals and evolutionary methods. Natural computing **17**(3), 585–609 (2018)
18. Fagan, M.: Design and code inspections to reduce errors in program development. In: Software pioneers, pp. 575–607. Springer (2002)
19. Ghannem, A., El Boussaidi, G., Kessentini, M.: Model refactoring using examples: a search-based approach. Journal of Software: Evolution and Process **26**(7), 692–713 (2014)
20. Ghannem, A., El Boussaidi, G., Kessentini, M.: On the use of design defect examples to detect model refactoring opportunities. Software Quality Journal **24**(4), 947–965 (2016)
21. Ghannem, A., Kessentini, M., El Boussaidi, G.: Detecting model refactoring opportunities using heuristic search. In: Proceedings of the 2011 Conference of the Center for Advanced Studies on Collaborative Research, pp. 175–187 (2011)
22. Harman, M., Mansouri, S.A., Zhang, Y.: Search-based software engineering: Trends, techniques and applications. ACM Computing Surveys (CSUR) **45**(1), 11 (2012)
23. Jackson, R.R., Carter, C.M., Tarsitano, M.S.: Trial-and-error solving of a confinement problem by a jumping spider, portia fimbriata. Behaviour **138**(10), 1215–1234 (2001)
24. Kalboussi, S., Bechikh, S., Kessentini, M., Said, L.B.: Preference-based many-objective evolutionary testing generates harder test cases for autonomous agents. In: International Symposium on Search Based Software Engineering, pp. 245–250. Springer, Berlin, Heidelberg (2013)
25. Keller, A.A.: Multi-Objective Optimization in Theory and Practice II: Metaheuristic Algorithms. Bentham Science Publishers (2019)
26. Kessentini, M., Mahaouachi, R., Ghedira, K.: What you like in design use to correct bad-smells. Software Quality Journal **21**(4), 551–571 (2013)
27. Kessentini, M., Ouni, A., Langer, P., Wimmer, M., Bechikh, S.: Search-based metamodel matching with structural and syntactic measures. Journal of Systems and Software **97**, 1–14 (2014)
28. Kononenko, O., Baysal, O., Guerrouj, L., Cao, Y., Godfrey, M.W.: Investigating code review quality: Do people and participation matter? In: 2015 IEEE international conference on software maintenance and evolution (ICSME), pp. 111–120. IEEE (2015)
29. Ouni, A., Kessentini, M., Ó Cinnéide, M., Sahraoui, H., Deb, K., Inoue, K.: More: A multi-objective refactoring recommendation approach to introducing design patterns and fixing code smells. Journal of Software: Evolution and Process **29**(5), e1843 (2017)

30. Ouni, A., Kula, R.G., Inoue, K.: Search-based peer reviewers recommendation in modern code review. In: 2016 IEEE International Conference on Software Maintenance and Evolution (ICSME), pp. 367–377. IEEE (2016)
31. Ouni, A., Kula, R.G., Kessentini, M., Ishio, T., German, D.M., Inoue, K.: Search-based software library recommendation using multi-objective optimization. Information and Software Technology **83**, 55–75 (2017)
32. Rachmawati, L., Srinivasan, D.: Multiobjective evolutionary algorithm with controllable focus on the knees of the pareto front. IEEE Transactions on Evolutionary Computation **13**(4), 810–824 (2009)
33. Rigby, P.C., Bird, C.: Convergent contemporary software peer review practices. In: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, pp. 202–212. ACM (2013)
34. Rigby, P.C., German, D.M., Storey, M.A.: Open source software peer review practices: a case study of the apache server. In: Proceedings of the 30th international conference on Software engineering, pp. 541–550. ACM (2008)
35. Rigby, P.C., Storey, M.A.: Understanding broadcast based peer review on open source software projects. In: 2011 33rd International Conference on Software Engineering (ICSE), pp. 541–550. IEEE (2011)
36. Thongtanunam, P., Kula, R.G., Cruz, A.E.C., Yoshida, N., Iida, H.: Improving code review effectiveness through reviewer recommendations. In: Proceedings of the 7th International Workshop on Cooperative and Human Aspects of Software Engineering, pp. 119–122. ACM (2014)
37. Thongtanunam, P., Tantithamthavorn, C., Kula, R.G., Yoshida, N., Iida, H., Matsumoto, K.i.: Who should review my code? a file location-based code-reviewer recommendation approach for modern code review. In: 2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER), pp. 141–150. IEEE (2015)
38. Wang, H., Kessentini, M., Ouni, A.: Bi-level identification of web service defects. In: International Conference on Service-Oriented Computing, pp. 352–368. Springer, Cham (2016)
39. Wilcoxon, F., Katti, S., Wilcox, R.A.: Critical values and probability levels for the wilcoxon rank sum test and the wilcoxon signed rank test. Selected tables in mathematical statistics **1**, 171–259 (1970)
40. Xia, X., Lo, D., Wang, X., Yang, X.: 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), pp. 261–270. IEEE (2015)
41. Xia, X., Lo, D., Wang, X., Zhou, B.: Accurate developer recommendation for bug resolution. In: 2013 20th Working Conference on Reverse Engineering (WCRE), pp. 72–81. IEEE (2013)
42. Yang, X., Kula, R.G., Yoshida, N., Iida, H.: Mining the modern code review repositories: A dataset of people, process and product. In: Proceedings of the 13th International Conference on Mining Software Repositories, pp. 460–463. ACM (2016)
43. Yang, X., Yoshida, N., Kula, R.G., Iida, H.: Peer review social network (person) in open source projects. IEICE TRANSACTIONS on Information and Systems **99**(3), 661–670 (2016)
44. Yu, Y., Wang, H., Yin, G., Wang, T.: Reviewer recommendation for pull-requests in github: What can we learn from code review and bug assignment? Information and Software Technology **74**, 204–218 (2016)
45. Zanjani, M.B., Kagdi, H., Bird, C.: Automatically recommending peer reviewers in modern code review. IEEE Transactions on Software Engineering **42**(6), 530–543 (2015)
46. Zanjani, M.B., Kagdi, H., Bird, C.: Automatically recommending peer reviewers in modern code review. IEEE Transactions on Software Engineering **42**(6), 530–543 (2016)