Search-Based Peer Reviewers Recommendation in Modern Code Review

Ali Ouni, Raula Gaikovina Kula, Katsuro Inoue
Department of Computer Science, Osaka University, Japan
Email: {ali,raula-k,inoue}@ist.osaka-u.ac.jp

Abstract—Code review is of primary importance in modern software development. It is widely recognized that peer review is an efficient and effective practice for improving software quality and reducing defect proneness. For successful review process, peer reviewers should have a deep experience and knowledge with the code being reviewed, and familiar to work and collaborate together. However, one of the main challenging tasks in modern code review is to find the most appropriate reviewers for submitted code changes. So far, reviewers assignment is still a manual, costly and time-consuming task. In this paper, we introduce a search-based approach, namely RevRec, to provide decision-making support for code change submitters and/or reviewers assigners to identify most appropriate peer reviewers for their code changes. RevRec aims at finding reviewers to be assigned for a code change based on their expertise and collaboration in past reviews using genetic algorithm (GA). We evaluated our approach on a benchmark of three open-source software systems, Android, OpenStack, and Qt. Results indicate that RevRec accurately recommends code reviewers with up to 59% of precision and 74% of recall. Our experiments provide evidence that leveraging reviewers expertise from their prior reviews and the socio-technical aspects of the team work and collaboration is relevant in improving the performance of peer reviewers recommendation in modern code review.

I. INTRODUCTION

Software code review is a disciplined engineering practice that has been commonly employed for several years in both industrial development and the open-source software (OSS) community [1]. Peer code review is a manual inspection of a code change, i.e., patch, by third-party developers, before it is committed to the project’s code base, in order to detect and correct potential defects and ensure quality software [2], [3].

Modern code review (MCR) has become a vital and essential practice in contemporary software development [4], [5]. Taking the Linus’s Law that [6]:

“many eyes make all bugs shallow”,

the OSS community have incorporated MCR, where developers utilize dedicated tools that facilitate the code review process, e.g., Gerrit1, Codestriker2, and ReviewBoard3. It is widely recognized that peer code review is a valuable and effective practice that can be applied to software development at all stages of the life cycle, to improve software quality, decrease defect-proneness, share knowledge and increase learning through rich communication [3], [7], [8].

Although MCR tools provide efficient and automated techniques to support the code review process, still a significant amount of human effort involved. In typical software projects, author of a code change need to invite/assign reviewers mainly based on their expertise with the changed files and previous review collaborations, in order for their change to be merged [5], [9]. One of the main challenges in code review is to find the most appropriate reviewers for pending reviews in a timely manner. Inappropriate reviewers assignment may lead to an inaccurate, time consuming and non effective review process.

Identifying appropriate peer reviewers is a non-trivial decision-making task for developers. If a patch affects several modules in the project, then generally the review should be performed by several peers of each affected module. For example, in VMware, most of the projects require at least two independent reviewers for every commit [10]. Moreover, since a file could be edited by multiple developers, and reviewed by multiple reviewers, it is difficult to find and assign appropriate reviewers if the number of files involved and the changes within them are large.

In code review, the reviewers expertise with the code fragments being reviewed is crucial to ensure time effective and high quality review. Recent studies showed that when reviewers have a priori knowledge of the context and the code, they complete reviews more quickly and provide more valuable feedback to the author [8], [11]. On the other hand, as code review is basically a human process involving personal and social aspects, thus the socio-technical factor plays an extremely important role in finding peer reviewers [9], [12].

Ultimately, efficient reviewers recommendation tools are essential to provide quality review and reduce the time taken for the review process. Unfortunately, so far, the field of peer reviewers recommendation in MCR is still in its infancy, and there has been little effort in building automatic recommendation techniques. The first attempts in addressing this problem, [5], [13], and [14], formulated the peer reviewers recommendation problem from a single perspective to find and rank candidate reviewers based on their experience with the code being reviewed. Reviewers are ranked and recommended in an independent manner, neglecting the socio-technical factor related to the relationships between review contributors which is a crucial aspect that affects the review quality as pointed out by many researchers and practitioners [9], [12], [15]–[17].

In this paper, we propose a novel approach, namely RevRec, that formulates the peer code reviewers recommendation prob-

1https://code.google.com/p/gerrit/
2http://codestriker.sourceforge.net/
3http://www.reviewboard.org/
lem as a combinatorial search-based optimization problem. The aim is to find appropriate reviewers for a given patch based on their expertise with the patch files and their prior review collaboration (co-review) rate with the review request submitter. To this end, we used genetic algorithm (GA) [18], in order to explore this large search space of possible reviewers combinations. Indeed, the search space is not determined only by the number of reviewers in a project, but also with the number of files, modules and contributors in the project.

The reviewer expertise refers to the frequency and recency of reviews performed by a reviewer to the patch’s files or module. The frequency is a count of the number of participated review comments, while the recency refers to the time span since the most recent review on each file. Indeed, as expertise may change over time, we consider both reviews frequency and recency as primary factors to capture the reviewers expertise. Second, the concept of review collaboration refers the number of times the candidate reviewers previously reviewed for a code author. Indeed, as the code review process is mainly a human process, we thus leverage the socio-technical aspects of the teamwork and collaboration which has been proved as important factor to the review quality and efficiency [9], [12], [19]. Indeed, complex combinatorial decision problems, such as peer reviewers recommendation, are best suited to search-based software engineering (SBSE) [20].

The main contributions of the paper can be summarized as follows:

1) We introduce a search-based formulation for the peer code reviewers recommendation problem. Our approach, RevRec, aims at finding, among a large list of reviewers, an appropriate set of reviewers, to review a code change, by leveraging reviewers expertise and collaboration.

2) We evaluate our approach on a benchmark of three open-source projects, Android, OpenStack, and Qt using Gerrit code review. We report the results of an empirical study on an implementation of our approach with a comparison with available state-of-the-art techniques on peer reviewers recommendation. Results indicate that RevRec significantly outperforms three existing techniques by accurately recommending code reviewers with up to 59% of precision and 74% of recall.

3) We present the results of a second empirical study that serves as ‘sanity check’ to compare our GA-based approach with existing search techniques including simulated annealing (SA), particle swarm optimization (PSO), and random search (RS) for solving the problem.

The rest of the paper is structured as follows. Section II describes necessary background. Section III introduces our RevRec approach for reviewers recommendation. Section IV describes the design of the empirical study we employ to evaluate our approach, while Section V presents and discusses the obtained results. Section VI describes the threats to validity. Section VII outlines the related work. Finally, in Section VIII, we conclude and describe our future research directions.
patch that is being reviewed before it is merged into the code base which follows four main phases.

1) **Submission phase** - Owner (Stephan) creates a change and submits it for a code review. At this stage, the review is labeled ‘open’. Open reviews can be accessible by any reviewer to make review comments.

2) **Review phase** - Reviewers (Alex and David) start to make comments on the code, which may lead to additional change revisions. In this case, the change consisted of 6 revised patch sets.

3) **Testing and Verification phase** - Running concurrently in the review phase, reviewers Jack and Michel, run tests and verify that the change will not cause failure in other parts of the code base. In this example, both reviewers are ‘bot’ applications that perform the tests automatically.

4) **Decision Phase** - Once satisfied with the state of the patch and all the testing and verification have been completed, reviewers Paul and David approve the patch. Both reviewers have higher roles in the project, thus are able to approve this review. As shown, both reviewers need to score either +1 or +2 to approve the change. Negative scores such as -1 or -2 will lead to the change being rejected. The change is then successfully merged into the codebase. It is then labelled as ‘merged’. Rejected changes are labeled as ‘abandoned’.

From the example in Figure 1, due to the nature of OSS, we find that attracting reviewers to be assigned to a review can be tedious, especially in large projects. Each reviewer has different roles, expertise, experience, and collaboration which are critical factors for review to be completed in a timely and efficient manner. For instance, finding a reviewer who has had experience involving the same files in that particular modules of the code base, and preferably familiar with the code author. Also, a reviewer who is able to approve a code change is critical in order for the change to be successfully merged. To help with this tedious task of selecting appropriate reviewers, we propose a recommendation technique to explore and search for possible reviewers to perform a given review.

**B. Search-Based Software Engineering**

Search-Based Software Engineering (SBSE) consists of the application of a computational search to solve optimization problems in software engineering [21]. The term SBSE was coined by Harman and Jones in 2001, and the goal of the field is to move software engineering problems from human-based search to machine-based search, using a variety of techniques from the metaheuristic search and evolutionary computation paradigms [21], [22]. SBSE provides best practice in formulating a software engineering problem as a search problem, by defining a suitable solution representation, fitness function, and solution change operators. Indeed, there are a multitude of search algorithms ranging from single to multi-objective techniques that can be applied to solve that problem [20].

**C. Genetic algorithms**

Genetic Algorithms (GA) [18] are computer algorithms that search for good solutions to a problem from among a large number of possible solutions. These computational paradigms were inspired by the mechanics of natural evolution, including survival of the fittest, reproduction, and mutation.

GAs begin with a set of random population of candidate solutions, also called individuals or chromosomes. Each individual of the population is evaluated by a fitness function that determines a quantitative measure of its ability to solve the target problem. The exploration of the search space is achieved by the evolution of candidate solutions using selection and genetic operators such as crossover and mutation. The selection operator ensures selection of individuals in the current population proportionally to their fitness values, so that the fitter an individual is, the higher the probability is that it be allowed to transmit its features to new individuals by undergoing crossover and/or mutation operators.

A new population is created from individuals of an old population in hope of getting a better population. Solutions which are chosen to form new solutions (offspring) are selected according to their fitness. The more suitable the solutions are the bigger chances they have to reproduce. This process is repeated until some condition is satisfied. The result of GA (the best solution found) is the fittest individual produced along all generations.

**III. RevRec: SEARCH-BASED PEER REVIEWERS RECOMMENDATION**

In this section, we describe our approach, RevRec, for recommending appropriate reviewers for code changes. Then, we describe our formulation of the peer reviewers recommendation problem as a search-based optimization problem.

**A. Approach overview**

Our approach aims at supporting review request submitter to find and invite appropriate reviewers in order to reduce the time taken for the review process and provide quality review. Figure 2 presents an overview of the RevRec approach. RevRec takes as input a review request which consists of a patch, i.e., a set of changed files submitted by a developer, and the history of completed code reviews recorded from the project’s review tool, e.g., Gerrit. As output, RevRec recommends a set of peer reviewers that are most appropriate to review the submitted change. Our approach used genetic algorithm to find the best set of reviewers based on two heuristics (1) the reviewers expertise with the submitted patch files, and (2) the reviewers collaboration with the review request submitter.

1) **Expertise.** The reviewers expertise is one of the main factors for a successful code review. It is widely accepted that if the reviewers have a priori knowledge of the code, they complete reviews more quickly and effectively [8], [11].

2) **Collaboration.** Each reviewer has his own collaboration sub-team, i.e., social network, within the project. It is
common that, in large projects, some reviewer(s) frequently review the patches from a particular developer. Moreover, it is common that sub-groups of reviewers use to collaborate together, i.e., co-review, based on repositories or modules. In practice, the reviewer(s) and the code author communicate more frequently with each other than with other peers, which may form a friendship [9], [19]. On the other hand, code review can be a source of conflict, as a code author may consider a reviewer’s rejection or critique of his code to be unfair and become offended. Thus code authors often invite or assign reviewers from their peers that they used to work with [9].

Each reviewer $r_i$ has (i) his own expertise with the files $F_p$, and (ii) his review collaboration history with the developer $d$, i.e., the reviewer request submitter, and also with the rest of other co-reviewers $R_p \setminus r_i$ in the project.

Our approach consists of the two following components:

**Reviewer expertise model (RevRec)**. Each reviewer $r_i$ has a degree of expertise with each of the patch files $F_p$ that can be represented as a vector. Each vector’s dimension is associated with its relative file $f_{p_j}$, and denoted by:

$$E_{r_i} = \{\text{Exp}(r_i, f_{p_1}), \ldots, \text{Exp}(r_i, f_{p_q})\}$$

where $\text{Exp}(r_i, f_{p_j})$ denotes the expertise of the reviewer $r_i$ with the file $f_{p_j}$.

Formally, the expertise of the reviewer $r_i$ with the patch files $F_p$ is denoted as $E(r_i, F_p)$ and calculated as follows:

$$E(r_i, F_p) = \sum_{\forall f_{p_j} \in F_p} \text{Exp}(r_i, f_{p_j}) \quad (1)$$

where $\text{Exp}(r_i, f_{p_j})$ refers to the expertise of the reviewer $r_i$ with the file $f_{p_j} \in F_p$.

To calculate $\text{Exp}(r_i, f_{p_j})$, we collect all previous reviews Rev performed by $r_i$ that are (i) closed, i.e., marked as “Merged”or“Abandoned”, and (ii) created before the patch $P$. For each file $f_{p_j}$ in the patch $P$, we check if similar files are previously reviewed in each collected review in Rev. Our similarity measure is based on file path similarity. For each file, we first split its path into components using file separator (i.e., slash character) as a delimiter. Each component represents a module, directory, package or file in the system. Thereafter, we use a camel case splitter to break down each component to its constituent tokens. Then, the file path similarity measure $(Sim)$ represents the common tokens between two files $f_a$ and $f_b$ based on the Jaccard similarity, as follows:

$$Sim(f_a, f_b) = \frac{|\text{tokens}(f_a) \cap \text{tokens}(f_b)|}{|\text{tokens}(f_a) \cup \text{tokens}(f_b)|} \in [0, 1] \quad (2)$$

where $\text{tokens}(f_i)$ is the function that returns all tokens in a file path $f_i$ as described above.

Our expertise model combines two main aspects. For each previously reviewed file, the reviewer expertise combines the review comments (1) frequency, and (2) recency. These two aspects are calculated as follows.

1) **Comments frequency (CF)**. The $CF$ of the reviewer $r_i$ for a file $f$ is a simple count of the number of comments of all its similar files $F$ having a Sim score greater than a given threshold $k$, to be set up by the review request submitter. Formally, $CF$ is given as follows.

$$CF(r_i, f) = \sum_{\forall f_j \in F} \text{comments}(r_i, f_j) \quad (3)$$

where $\text{comments}(r_i, f_j)$ is a count of the number of review comments participated by the reviewer $r_i$ for the file $f_j$.

2) **Comments recency (cr)**. The measure $cr$ is a weight that reflects the recency, i.e., freshness, of the review...
comments on the file \( f \) by the reviewer \( r_i \). The weight \( cr \) is given as follows.

\[
    cr(r_i, f) = 1 - \frac{T_f}{T} \in [0, 1]
\]

(4)

where \( T_f \) is a count of the number of calendar days since the most recent comment in all similar files \( F \), and \( T \) is a count of number of calendar days since the whole project is created. That is, if the most similar (or same) file is commented at the day the patch \( P \) is submitted, then \( cr = 1 \).

Then, \( Exp(r_i, f) \) is defined as follows.

\[
    Exp(r_i, f) = cr(r_i, f) \times CF(r_i, f)
\]

(5)

Finally, the expertise of a (set of) reviewer(s) \( R_p \) to be invited/assigned for the patch \( P \) having a set of files \( F_p \) is denoted as \( RE \) and calculated as follows:

\[
    RE(R_p, F_p) = \frac{\sum_{r_i \in R_p} E(r_i, F_p)}{|R_p|}
\]

(6)

where \( E(r_i, F_p) \) is given by Equation 1.

**Reviewer collaboration model (RevRecRC).** Each reviewer \( r_i \) may have a review collaboration \( RC \) with both (i) the review request submitter, and/or (ii) the rest of candidate reviewers \( R_p \setminus r_i \) based on past reviews. The review collaboration forms a social network within the project. This network is represented as a graph \( G = (V, E) \) where the vertices \( V \) represent the people (code authors and reviewers) and the edges \( E \) represent the collaboration measure between them as a count of the number of exchanged review comments during their past co-reviews. For a patch \( P \), the code author and the candidate reviewers can be represented as a weighted, undirected sub-graph \( G_p = (V_p, E_p) \).

Formally, let \( R_p \) a candidate set of reviewers recommended for a patch \( P \) submitted by a code author \( d \), then, we calculate the reviewers collaboration, \( RC \), from the sub-graph \( G_p = (V_p, E_p) \) based on two aspects: the sub-graph connectivity, and the sum of weights on the edges (comments count), as follows:

\[
    RC(d, R_p) = \frac{|E_p|}{|V_p| \times (|V_p| - 1)/2} \times \sum_{e_{p_i} \in E_p} e_{p_i}
\]

(7)

where the coefficient \( \frac{|E_p|}{|V_p| \times (|V_p| - 1)/2} \in [0, 1] \) reflects the sub-graph connectivity (ideally equals to 1 if the graph is complete), and \( e_{p_i} \) is the weight on the edge \( e_i \), i.e., a count of the number of review comments between each pair of contributors (reviewers and code authors).

To illustrate the collaboration model, let us consider the example sketched in Figure 3, where a patch is submitted by a developer \( d = \text{Stephan} \), and reviewed by a set of reviewers \( R_p = \{\text{Paul}, \text{Michel}, \text{David}, \text{Alex}\} \). Then, the reviewer collaboration score \( RC \) is calculated from the collaboration sub-graph as follows, \( RC = \frac{9}{3 \times 47/2} \times 285 = 256.5 \).

![Fig. 3: Example of review collaboration sub-graph.](image.png)

**C. Genetic Algorithm Adaptation**

In the following, we describe our SBSE model for RevRec. To adopt a search-based technique, a set of essential elements should be defined including the solution representation, fitness function, computational search algorithm, selection, genetic operators, and termination criteria [21].

**Solution representation.** A solution \( R_p \), i.e., set of peer reviewers, for a patch \( P \), can be represented in an array of size equals to the total number of current reviewers working on the project, \( \text{numReviewers} \). Each element from this array denotes whether a reviewer is recommended for the code change (‘1’) or not (‘0’). For example, Figure 4 encodes a candidate solution for the review example of the OpenStack project #283653 sketched in Figure 1. For sake of simplicity, we assume that OpenStack has only ten reviewers. This solution example indicates that the 2nd, 4th, 5th, 7th, and the 10th reviewers (Paul, Michel, David, Alex, and Jack, respectively) are recommended for this code change submitted by Stephan.

<table>
<thead>
<tr>
<th>Daniel</th>
<th>Paul</th>
<th>Sarah</th>
<th>Michel</th>
<th>David</th>
<th>Olivia</th>
<th>Alex</th>
<th>Liu</th>
<th>John</th>
<th>Jack</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

![Fig. 4: Solution representation.](image.png)

**Constraints.** Each solution has to comply with a set of constraints. The first constraint considered in RevRec, is that the recommended list should contain at least one approver, i.e., core reviewer. This constraint, could be either activated or inactivated by the review request submitter. Moreover, our solutions should satisfy two other constraints, the minimum and the maximum number of recommended reviewers, \( \minRev \) and \( \maxRev \), respectively. The two latter constraints are checked by simply calculating the sum of the ‘1’ elements in the solution’s array. RevRec uses these three constraints as configuration parameters to be set up by the developer according to the complexity and the size of his code change.

**Computational search algorithm.** As a search method, we employed a widely used metaheuristic algorithm, namely genetic algorithm (GA) [18].

GA starts by creating an initial population of solutions. A solution is a set of peer reviewers. In the first generation, these solutions are generated randomly. Then in an iterative process, every iteration produces a new generation of solutions derived from the previous ones. For each iteration, GA pushes the “fittest” candidates to the next generation (elitism), and then GA generates the rest of the solutions composing the next generation by combining/modifying existing solutions using
crossover and/or mutation operators. The fitness of a solution is computed using a function implementing the heuristics stated earlier (reviewers expertise and collaboration).

**Fitness function.** The fitness function quantifies the quality of each individual in the current population. To evaluate the fitness of each candidate solution we employed a fitness function that combines the reviewers expertise (RevRecRE) and reviewers collaboration (RevRecRC). To calculate the fitness of a candidate solution \( R_p \), for a patch \( P(d, F_p) \) submitted by a code author \( d \) and consists of a set of \( F_p \) files, we use the following function:

\[
    \text{Fitness}(R_p, P) = \alpha \times \text{RE}(r_i, F_p) + \beta \times \text{RC}(d, R_p)
\]

where the functions \( \text{RE}(r_i, F_p) \) and \( \text{RC}(d, R_p) \) are given by Equations 6 and 7, respectively, and \( \alpha + \beta = 1 \).

Note that both \( \text{RE} \) and \( \text{RC} \) values are normalized in the range \([0,1]\) using min-max normalization based on all solutions in the current population.

**Selection.** The selection process is based on fitness. Solutions that are evaluated with higher values (fitter) will most likely be selected to reproduce, whereas, those with low values will be discarded. In RevRec, we used roulette-wheel selection [23], combined with elitism, where a number of fittest solutions are copied without changes to the new population, so the best solutions found will not be lost.

Roulette-wheel is a simple method of implementing fitness-proportionate selection. It is conceptually equal to giving each individual a slice of a circular roulette wheel equal in area to the individual’s fitness. The wheel is spun \( n \) times, and on each spin, the individual under wheel’s marker is selected to be in the pool of parents for the next generation [23].

**Genetic operators.** We defined our genetic operators crossover and mutation for RevRec as follows.

**Crossover.** We employ a single random cut-point crossover. This operator is performed by generating a random number \( k \) between 0 and \( \text{numReviewers} - 1 \). Then, it exchanges the subsequences before and after \( k \) between two parent individuals to create two offsprings. We perform crossover with a certain probability.

**Mutation.** Our mutation changes the new offspring by flipping bits from 1 to 0 or from 0 to 1. Mutation can occur at each bit position in the string with some probability. It aims at preventing falling all solutions in the population into a local optimum of solved problem.

**Termination conditions.** The population converges when either 90\% of the solutions in the population have the same fitness value or the number of evaluation functions is greater than a fixed number, \( \text{maxEval} \). Then the best (fittest) solution is returned by RevRec, based on its fitness value.

**D. Reviewers ranking**

Although our approach recommends a set of candidate reviewers to co-review a submitted patch, each has his expertise with some particular (or all) files and used to co-review with some (or all) other peers, RevRec provides a second component to rank all candidate reviewers in the project. Our ranking technique is as follows. Top \( k \) solutions of the last population obtained in the last iteration of GA are copied in a single pool. Then, the rank of each reviewer corresponds to his frequency count in the pool. That is, reviewers that are recommended in many solutions are ranked first.

**IV. EMPIRICAL EVALUATION**

In this section, we present the results of our evaluation for the proposed approach. The aim of this study is to investigate the effectiveness of the RevRec approach in providing peer reviewers recommendation solutions. We conduct two empirical studies to compare our approach with (1) existing state-of-the-art approaches in peer reviewers recommendation, and (2) existing metaheuristic search techniques.

**A. Research Questions**

We design our experiments to address three research questions.

RQ1. How accurate is RevRec in recommending peer reviewers for code changes?  
RQ2. What are the effects of each of the reviewers expertise and collaboration on the accuracy of RevRec?  
RQ3. How does GA compared to random search (RS) and other popular search algorithms, SA and PSO?

**B. Context Selection**

**Studied systems.** We evaluate our approach on a benchmark [24] of three well-known open-source systems, Android\(^5\), OpenStack\(^6\), and Qt\(^7\). Android is a free software stack for a wide range of mobile devices led by Google. OpenStack is software platform for cloud computing, controlling large pools of compute, storage, and networking resources throughout a datacenter. Qt is a comprehensive cross-platform framework and development tools that is widely used for developing application software. All collected reviews are closed, i.e., marked as “Merged” or “Abandoned”, and contain at least one file. Table I summarizes the statistics of the three systems.

<table>
<thead>
<tr>
<th>System</th>
<th>Period studied</th>
<th>#Reviews</th>
<th>#Reviewers</th>
<th>#Files</th>
</tr>
</thead>
<tbody>
<tr>
<td>Android</td>
<td>2008-10 ~ 2012-01</td>
<td>5,126</td>
<td>94</td>
<td>26,840</td>
</tr>
<tr>
<td>OpenStack</td>
<td>2011-07 ~ 2012-05</td>
<td>6,586</td>
<td>82</td>
<td>16,953</td>
</tr>
<tr>
<td>Qt</td>
<td>2011-05 ~ 2012-05</td>
<td>23,810</td>
<td>202</td>
<td>78,401</td>
</tr>
</tbody>
</table>

We selected these three systems for our evaluation because they range from medium to large-sized open-source projects, which have been actively developed for more than five years, and have been extensively studied in the software review literature [7], [13], [14], [24].

**State-of-the-art techniques.** To evaluate the efficiency of our approach we compare it with available state-of-the-art techniques, chRev, REVINDER, and ReviewBot, in order to investigate what improvements such an approach will bring. The

---

\(^{5}\)https://source.android.com/

\(^{6}\)http://www.openstack.org/

\(^{7}\)http://qt-project.org/
cHRev approach [14] is based on a reviewer expertise model, generated from completed reviews, that combines a quantification of review comments and their recency. REVFINDER [13] is based on the past reviews of files with similar names and paths to build an expertise model. ReviewBot [5] is based on the modification history of source code using static analysis tools to find experienced reviewers.

Moreover, we compare our GA-based approach with random search (RS) [25] as a ‘sanity check’ and also with existing search techniques including simulated annealing (SA) [26] and particle swarm optimization (PSO) [27] for solving the problem, as they are the most popular techniques in solving software engineering problems [20], [22].

C. Method Analysis

We evaluate our approach on the dataset obtained from our three studied systems as described in Table I. Due to the stochastic nature of the used search algorithms, they may give slightly different results in each run. To cuter with this stochastic nature, we repeat the simulation run 31 times in each case, and report the median value as commonly used in metaheuristic algorithms [28].

Our evaluation process is as follows. For each system, we take the most recent 1,000 reviews, in their chronological order, as test reviews. For each test review $T_r$, we consider its actual reviewers, that are known, as the ground truth. Thereafter, we execute RevRec on $T_r$ by deriving the expertise and collaboration models from all reviews performed before $T_r$ was created. Similarly, cHRev, REVFINDER, and ReviewBot are evaluated with the same process.

To answer RQ1, we used three common evaluation metrics for recommendation systems, the precision, recall, and mean reciprocal rank (MRR) [5], [13], [14], [29]. For each test review $T_r$, of each studied system, we calculate the precision and recall as follows.

\[
\text{precision@}k = \frac{TP}{TP + FP} \quad \text{(9)}
\]

\[
\text{recall@}k = \frac{TP}{TP + FN} \quad \text{(10)}
\]

where TP (True Positive) corresponds to the number of top-$k$ reviewers recommended by the approach and also actual reviewers; FP (False Positive) corresponds to the number of top-$k$ reviewers recommended by the approach, but not actual reviewers; FN (False Negative) corresponds to the number of actual reviewers, that are not among the top-$k$ reviewers recommended by the approach.

We calculate these metrics with different $k$ values, 1, 3, 5, 10 and auto. The auto refers to the size of the best solution (set of reviewers) recommended by RevRec, auto $\in [\text{minRev, maxRev}]$ of size $n$ which is automatic by RevRec. To ensure fair comparison, we select the top $n$ reviewers for each approach.

In addition to the precision and recall, we calculate the Mean Reciprocal Rank (MRR) as the average of reciprocal ranks of true positive reviewers in a recommendation list. The reciprocal rank of a reviewers recommendation list is the multiplicative inverse of the rank of the first true positive reviewer. The mean reciprocal rank is the average of the reciprocal ranks of results for a sample of recommendation list. Given a reviewers recommendation lists $R$, the score MRR is calculated as follows:

\[
MRR = \frac{\sum_{r \in R} \text{rank}(r)}{|R|} \quad \text{(11)}
\]

where $\text{rank}(r)$ is the rank score of the first reviewer in the recommendation list $r$. The higher is the MRR score, the better is the recommendation approach.

To answer RQ2, we investigate the effectiveness of each of the components of RevRec, the reviewers expertise (RevRec$^{\text{RE}}$), and the reviewers collaboration (RevRec$^{\text{RC}}$). To assess the effectiveness of RevRec$^{\text{RE}}$, we set up $\alpha = 1$ and $\beta = 0$ in the fitness function (Equation 8), and inversely for the RevRec$^{\text{RC}}$ component (i.e., $\alpha = 0$ and $\beta = 1$). To this end, we use our evaluation metrics precision@k, recall@k, and MRR.

To answer RQ3, we compare the effectiveness of GA against SA, PSO and RS in terms of precision and recall. This evaluation is a sanity check to make sure that we used an appropriate search algorithm. Moreover, if the proposed formulation does not allow an intelligent search algorithm to outperform random search, then the proposed formulation is not appropriate. For all algorithms, we use the same formulation given in Section III-B. The parameters setting and statistical tests used are described in sections IV-D and IV-E, respectively.

D. Parameter Setting

The initial population/solution of GA, SA, PSO, and RS are completely random. The stopping criterion is when the maximum number of function evaluations $\text{maxEval}$, set to 8,000, is reached. After several trial runs of the simulation, the parameter values were fixed. Indeed, there are no general rules to determine these parameters, and thus we set the combination of parameter values by trial-and-error, a method that is commonly used by the SBSE community [28], [30]. For replicability, we report in Table II the parameter settings of GA, SA and PSO. For RS, we switched off the individual selection based on fitness value in our GA adaptation.

For our experiments, the parameters $\text{minRev}$ is set to 1, and $\text{maxRev}$ is set to 10. We set the parameters $\alpha$ and $\beta$ to 0.5 as default parameters. Note that RevRec allows the

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>GA</td>
<td>population size</td>
<td>200</td>
</tr>
<tr>
<td></td>
<td>crossover probability</td>
<td>0.85</td>
</tr>
<tr>
<td></td>
<td>mutation probability</td>
<td>0.1</td>
</tr>
<tr>
<td></td>
<td>number of crossing points</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>selection</td>
<td>Roulette selection</td>
</tr>
<tr>
<td>SA</td>
<td>initial temperature</td>
<td>100</td>
</tr>
<tr>
<td></td>
<td>final temperature</td>
<td>0.157</td>
</tr>
<tr>
<td></td>
<td>cooling coefficient</td>
<td>0.98</td>
</tr>
<tr>
<td></td>
<td>number of iterations</td>
<td>25</td>
</tr>
<tr>
<td>PSO</td>
<td>number of particles in a swarm</td>
<td>200</td>
</tr>
<tr>
<td></td>
<td>acceleration coefficient c1</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>acceleration coefficient c2</td>
<td>2</td>
</tr>
</tbody>
</table>
TABLE III: The precision and recall results achieved by RevRec, chRev, REVFINHER, and ReviewBot, for each system.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Android</td>
<td>auto</td>
<td>0.52</td>
<td>0.38</td>
<td>0.24</td>
<td>0.17</td>
<td>0.54</td>
<td>0.42</td>
<td>0.38</td>
<td>0.18</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0.58</td>
<td>0.50</td>
<td>0.34</td>
<td>0.21</td>
<td>0.51</td>
<td>0.27</td>
<td>0.18</td>
<td>0.11</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>0.57</td>
<td>0.35</td>
<td>0.25</td>
<td>0.17</td>
<td>0.51</td>
<td>0.50</td>
<td>0.39</td>
<td>0.19</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>0.39</td>
<td>0.30</td>
<td>0.22</td>
<td>0.12</td>
<td>0.61</td>
<td>0.61</td>
<td>0.48</td>
<td>0.29</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>0.34</td>
<td>0.26</td>
<td>0.18</td>
<td>0.09</td>
<td>0.73</td>
<td>0.65</td>
<td>0.54</td>
<td>0.38</td>
</tr>
<tr>
<td>OpenStack</td>
<td>auto</td>
<td>0.54</td>
<td>0.41</td>
<td>0.26</td>
<td>0.19</td>
<td>0.56</td>
<td>0.40</td>
<td>0.31</td>
<td>0.22</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0.59</td>
<td>0.48</td>
<td>0.32</td>
<td>0.24</td>
<td>0.41</td>
<td>0.31</td>
<td>0.15</td>
<td>0.12</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>0.51</td>
<td>0.42</td>
<td>0.27</td>
<td>0.20</td>
<td>0.54</td>
<td>0.39</td>
<td>0.29</td>
<td>0.20</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>0.43</td>
<td>0.38</td>
<td>0.25</td>
<td>0.16</td>
<td>0.61</td>
<td>0.52</td>
<td>0.37</td>
<td>0.32</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>0.36</td>
<td>0.31</td>
<td>0.21</td>
<td>0.11</td>
<td>0.74</td>
<td>0.66</td>
<td>0.46</td>
<td>0.39</td>
</tr>
<tr>
<td>Qt</td>
<td>auto</td>
<td>0.46</td>
<td>0.39</td>
<td>0.26</td>
<td>0.18</td>
<td>0.51</td>
<td>0.45</td>
<td>0.31</td>
<td>0.22</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0.49</td>
<td>0.45</td>
<td>0.30</td>
<td>0.22</td>
<td>0.41</td>
<td>0.33</td>
<td>0.14</td>
<td>0.09</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>0.45</td>
<td>0.40</td>
<td>0.27</td>
<td>0.19</td>
<td>0.50</td>
<td>0.47</td>
<td>0.27</td>
<td>0.16</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>0.41</td>
<td>0.37</td>
<td>0.21</td>
<td>0.13</td>
<td>0.59</td>
<td>0.52</td>
<td>0.35</td>
<td>0.24</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>0.34</td>
<td>0.31</td>
<td>0.16</td>
<td>0.09</td>
<td>0.63</td>
<td>0.60</td>
<td>0.43</td>
<td>0.30</td>
</tr>
</tbody>
</table>

TABLE IV: The mean reciprocal rank (MRR) of the approaches RevRec, chRev, REVFINHER, and ReviewBot, for each studied system.

<table>
<thead>
<tr>
<th>System</th>
<th>RevRec</th>
<th>chRev</th>
<th>REVFINHER</th>
<th>ReviewBot</th>
</tr>
</thead>
<tbody>
<tr>
<td>Android</td>
<td>0.69</td>
<td>0.65</td>
<td>0.60</td>
<td>0.25</td>
</tr>
<tr>
<td>OpenStack</td>
<td>0.63</td>
<td>0.58</td>
<td>0.55</td>
<td>0.30</td>
</tr>
<tr>
<td>Qt</td>
<td>0.54</td>
<td>0.43</td>
<td>0.31</td>
<td>0.22</td>
</tr>
</tbody>
</table>
An important observation to highlight from the results of Figure 5 and Tables III and IV is the importance of social relationships in peer code review. Results show that RevRec RC achieves comparable results to cHRev for all studied systems, but considerably better performance is achieved by RevRec when the reviewers collaboration is incorporated in the search process. This finding provides evidence that code reviewers assignment is influenced by social relationships between code authors and reviewers. That is, code authors are likely to be more familiar and invite their code review collaborators more than their non-code review collaborators.

In more details, we studied the intersection between the recommendation lists identified by each of RevRec RE and RevRec RC, individually in each of the three systems. We observed that more than 68% of common reviewers are recommended by both expertise and collaboration models. This indicates that the peer code review process is forming a review expertise between code review collaborators.

C. RQ3: Performance of GA

Figure 6 reports the boxpots, while Table V reports the Wilcoxon significance tests and the Cliff’s delta (d) effect size test to compare each of GA, PSO, SA and RS in terms of precision and recall. Each algorithm is executed 31 times and the default output is reported, without reviewers ranking (for this reason we do not consider the MRR metric). Clearly, GA, PSO and SA significantly outperform random search (RS) in all the three systems with a large effect size. This provides evidence that our problem formulation is appropriate, as an intelligent search is required to explore the search space.

From the results, we observe that GA achieves significantly better results than SA in terms of precision and recall with large effect size in all the cases (two metrics × three systems). Moreover, GA achieved significantly better results than PSO with large effect size, except in 2 out of 6 cases the effect size was medium (the precision with Android, and the recall with OpenStack). Furthermore, SA turns out to be remarkably lower than both GA and PSO in all metrics and all systems. More specifically, we observed that for smaller systems (Android and OpenStack), the performance of SA in terms of precision and recall is better than larger systems (Qt), as can be seen in the boxplots of Figure 6. This results indicate that population-based algorithms (GA and PSO) are more appropriate than local search algorithms (SA) for the reviewers recommendation problem.

Based on these results we can conjecture that GA achieves significantly better results than SA and PSO, in all the studied systems for the code reviewers recommendation problem.

VI. THREATS TO VALIDITY

This section discusses threats to the validity of our study.

Construct threat to validity can be related to the set of correct reviewers, i.e., the ground truth, to calculate precision and recall. During the code review process, sometimes reviewers are assigned to a code change but they do not contribute to it. This may due to several reasons including his current workload, availability or the social relationship with the code change submitter. To mitigate this threat, we considered the ground truth as the set of reviewers who are assigned and contributed by at least one comment, as our collaboration graph is based on the number of comments.
participated between two reviewers in their previous reviews. To accurately measure the level of reviewers collaboration, we have to consider other types of socio-technical interactions between them, e.g., exchanged emails, collaboration in related (sub-)projects, etc.

**Internal threats** to validity can be related to the stochastic nature of the metaheuristic algorithms [20]. To mitigate this threat, we used the Wilcoxon test and Cliff’s delta effect size over 31 independent runs of each algorithm [28]. The choice of the best weights, \(\alpha\) and \(\beta\), between our components, RevRec\textsubscript{RE} and RevRec\textsubscript{SC}, can affect the the overall results. Although we set the average \((\alpha = \beta = 0.5)\), we plan to empirically investigate several combinations.

**External threats** to validity can be related to the generalizability of our results. Although we empirically evaluated our approach on three large size open-source systems from different application domains, Android, OpenStack, and Qt, we cannot claim that the same results would be achieved with other projects or other periods of time. Moreover, our empirical evaluation is based on open-source projects using the Gerrit code review tool, therefore we cannot generalize our results on other code review tools. As future work, we plan to extend our evaluation on other open-source and industrial projects using different code review platforms.

### VII. RELATED WORK

Related work to this study could be divided into two main categories, (i) peer reviewers recommendation systems, and (ii) studies on social and human aspects in peer review.

**Peer code reviewers recommendation.** Balachandran first suggested to use review bot tool, as a recommendation system to reduce human effort and improve review quality in an global industrial setting, where face-to-face meetings are not possible [5]. Later, Patanamon et al. [13] adapted and improved the recommendation system to assist with the tool-based modern peer review process. The authors used the file review history to recommend the best reviewer. Recently, Zanjani et al. [14] proposed a reviewer recommendation approach based on a reviewer expertise model, generated from past reviews, that combines a quantification of review comments and their recency. Anvik et al. [34] used machine learning techniques to recommend developers to fix new bug reports. Xia et al. [35] used bug report 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.

**Human aspects in peer code review.** It is widely agreed that code review is a complex process involving personal and social aspects [17]. Fagan first proposed software inspection as a vigorous and systematic peer review activity to ensure the quality of software [36]. Due to the volunteer nature of OSS developers and the peer review structure, studies such as Rigby et al. [11], [37], Bosu et al. [9], [15] Baysal et al. [12], Bird et al. [38], and Yang et al. [16] found that different human factors and socio-technical issues influence the OSS peer review, motivating the need for a peer review recommendation system.

Baysal and her colleagues conducted several studies to investigate non-technical and human aspects in code review. They found that organizational, personal and participation dimensions influence the review process [12], [39]. Moreover, Baysal et al. found that the review process can be sensitive due to its nature of dealing with people’s egos [40].

Bosu et al. [9] studied different aspects of peer impression formation, and found 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. Also developers reputation is an important aspect to receive quicker first feedback, and core developers are more likely to have their code changes accepted into the project codebase [41].

### VIII. CONCLUSION AND FUTURE WORK

In this paper, we introduced a new search-based approach, namely RevRec, for peer code reviewers recommendation. Our approach uses genetic algorithm to find the appropriate peer reviewers to be invited/assigned to review code change based on their expertise and their collaboration in past reviews. The proposed approach is evaluated on a benchmark of three open source systems. Results provide evidence that formulating code reviewers recommendation based on a combination of reviewers expertise and their socio-technical relationships is more accurate and effective in practice. Moreover, statistical analysis of the obtained results indicate that GA is more efficient in solving the peer reviewers recommendation problem.

As future work, we plan to extend our approach to incorporate the reviewers workload in the recommendation task, and consider the number of recommended reviewers based on the code change size and complexity. Moreover, we plan to investigate and integrate the commit history in our expertise model. Another interesting research direction that interest us is to analyze the rich information embodied in the review comments in order to better understand the social relationships between code authors/reviewers to improve the reviewers recommendation accuracy.

**ACKNOWLEDGMENT**

This work was supported by Japan Society for the Promotion of Science, Grant-in-Aid for Scientific Research (S) JP25220003.
REFERENCES


