2016 IEEE International Conference on Software Maintenance and Evolution

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., Gerrit[1], Codestriker[2], and ReviewBoard[3]. 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].

[1]https://code.google.com/p/gerrit/
[2]http://codestriker.sourceforge.net/
[3]http://www.reviewboard.org/

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 reviews 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 improtant 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-

978-1-5090-3806-0/16 $31.00 © 2016 IEEE
DOI 10.1109/ICSME.2016.65

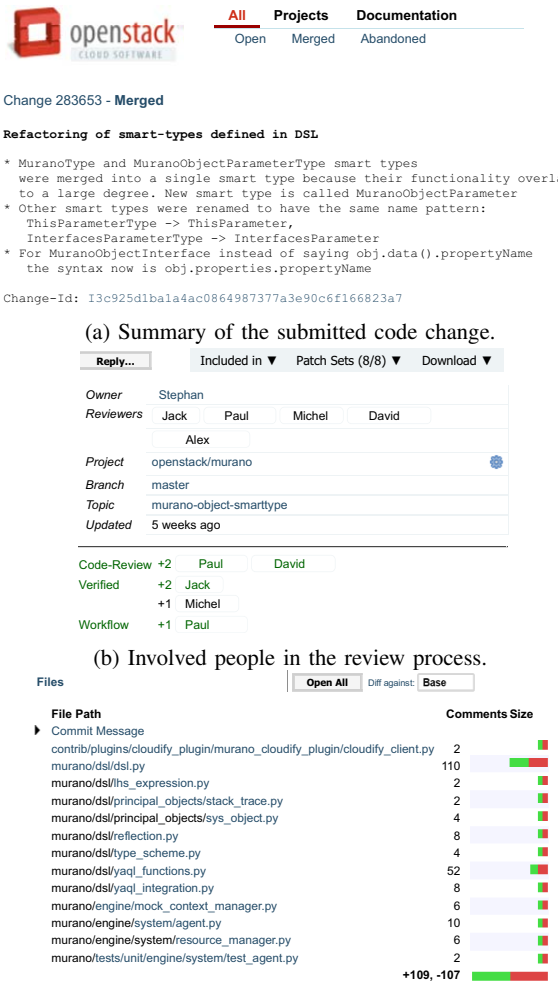
lem as a combinatorial search-based optimization problem. The aim is to find appropriate reviewers for a given patch based on their *expertise* with the patch files and their prior review *collaboration* (co-review) rate with the review request submitter. To this end, we used genetic algorithm (GA) [18], 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.



(a) Summary of the submitted code change.

(b) Involved people in the review process.

(c) Changed files in the submitted code change.

Fig. 1: A Gerrit-based review example[4] from the OpenStack project (code change #283653).

II. Background and Motivation

A. Modern code review

Recently, the lightweight modern code review process has been adopted by many open source (e.g., Android, Qt, OpenStack) and industry (e.g., Google, Microsoft, Cisco) projects. In contrast to the formal peer review, the modern peer review process is a tool-based management system. It entails contributors submitting patches of code that need to be reviewed and approved before they are committed to the code base.

To give the reader perspective, Figure 1 presents an example to describe the typical review process using the Gerrit tool based-code review within the OpenStack. The figure shows the Gerrit interface of the code change #283653[4] which is related to some refactoring operations. This code change refers to a

[4]https://review.openstack.org/#/c/283653/, For privacy reasons, we have hide some details from the interface, and used alias instead of the original developer names.

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 wwill 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, same 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].

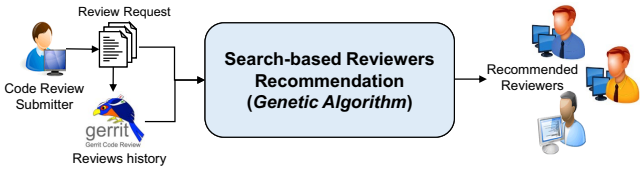


Fig. 2: RevRec overview.

We believe the peer reviewers recommendation problem is well suited for applying a search-based technique to aid developers find appropriate reviewers to their patches. Indeed, the number of reviewers combinations 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. Even, if a code change consists of one single file, there may be several reviewers that may have expertise with it.

Problem complexity. Finding the appropriate reviewers for a patch is not an obvious task for reviewer assigners as the number of possible reviewers can be very large causing a combinatorial problem. The search space tends to be enormous as the number of possible reviewers combinations is equals at least to $N = \binom{k}{n} \times p = \frac{k!}{n!(n-k)!} \times p$, where N counts the number of different possibilities of how a set of k reviewers can be identified from a given set of n reviewers working on the project, and p is a count of the number of files in the project. Note that the order of the reviewers is not considered.

Indeed, the search space is prohibitively large for an exhaustive approach. Therefore, we consider a meta-heuristic search and use a genetic algorithm to find a near-optimal solution representing the most appropriate set of reviewers.

B. Problem formulation

This section gives definitions and characteristics of the peer reviewers recommendation problem.

It is assumed that for an existing software system S, there is a set of developers, $D = \{d_1, ..., d_n\}$, and set of reviewers $R = \{r_1, ..., r_m\}$. The software system consists of a set of q source files, $F = \{f_1, ..., f_q\}$. A patch P, i.e., code change, is submitted by a developer $d \in D$, and itself consists of a set of t changed files $F_p = \{f_{p_1}, ..., f_{p_t}\}$, and denoted by $P\langle d, F_p\rangle$.

The set of possible peer reviewers for the patch P is denoted by:

$$R_p = \{r_1, ..., r_k\}$$

where $R_p \subseteq R$, and $1 \leq k \leq |R|$.

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 review 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 ($\mathbf{RevRec^{RE}}$). 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 fp_j, and denoted by:

$$E_{r_i} = \{Exp_{(r_i, f_{p_1})}, ..., Exp_{(r_i, f_{p_t})}\}$$

where $Exp_{(r_i, f_{p_j})}$ 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:

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

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

To calculate $Exp(r_i, f_{p_j})$, 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:

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

where $tokens(f_i)$ 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.

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

where $comments(r_i, f_j)$ 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 follow.

$$cr_{(r_i,f)} = 1 - \frac{T_f}{T} \in [0,1] \quad (4)$$

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.

$$Exp(r_i, f) = cr_{(r_i,f)} \times CF(r_i, f) \quad (5)$$

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:

$$RE(R_p, F_p) = \frac{\sum_{\forall r_i \in R_p} E(r_i, F_p)}{|R_p|} \quad (6)$$

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

Reviewer collaboration model ($\mathbf{RevRec^{RC}}$). 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 $R_p \backslash r_i$ 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:

$$RC(d, R_p) = \frac{|E_p|}{|V_p| \times (|V_p| - 1)/2} \times \sum_{\forall e_{p_i} \in E_p} e_{p_i} \quad (7)$$

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 = Stephan`, and reviewed by a set of reviewers $R_p = \{$`Paul, Michel, David, Alex`$\}$. Then, the reviewer collaboration score RC is calculated from the collaboration sub-graph as follows, $RC = \frac{9}{5*4/2} \times 285 = 256.5$.

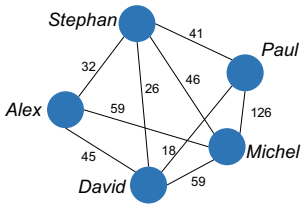


Fig. 3: Example of review collaboration sub-graph.

C. Genetic Algorithm Adaptation

In the following, we describe our SBSE model for RevRec. To adopt a seach-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, *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 2^{nd}, 4^{th}, 5^{th}, 7^{th}, and the 10^{th} reviewers (`Paul`, `Michel`, `David`, `Alex`, and `Jack`, respectively) are recommended for this code change submitted by `Stephan`.

Daniel	Paul	Sarah	Michel	David	Olivia	Alex	Liu	John	Jack
0	1	0	1	1	0	1	0	0	1

Fig. 4: Solution representation.

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 ($\text{RevRec}^{\text{RE}}$) and reviewers collaboration ($\text{RevRec}^{\text{RC}}$). To calculate the fitness of a candidate solution R_p for a patch $P\langle d, F_p\rangle$ submitted by a code author d and consists of a set of F_p files, we use the following function:

$$Fitness(R_p, P) = \alpha \times RE(R_p, F_p) + \beta \times RC(d, R_p) \quad (8)$$

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

Note that both RE and 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 *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, *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.

$\mathbf{RQ_1}$. How accurate is RevRec in recommending peer reviewers for code changes?

$\mathbf{RQ_2}$. What are the effects of each of the reviewers *expertise* and *collaboration* on the accuracy of RevRec?

$\mathbf{RQ_3}$. 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, controling 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 I: Studied systems statistics.

System	Period studied	#Reviews	#Reviewers	#Files
Android	2008-10 $\sim$ 2012-01	5,126	94	26,840
OpenStack	2011-07 $\sim$ 2012-05	6,586	82	16,953
Qt	2011-05 $\sim$ 2012-05	23,810	202	78,401

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, REVFINDER, 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 cutter 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 $\mathbf{RQ_1}$, 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.

$$precision@k = \frac{TP}{TP + FP} \tag{9}$$

$$recall@k = \frac{TP}{TP + FN} \tag{10}$$

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, $auto \in [minRev, maxRev]$ 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:

$$MMR = \frac{\sum_{\forall r \in R} rank(r)}{|R|} \tag{11}$$

where $rank(r)$ 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 $\mathbf{RQ_2}$, we investigate the effectiveness of each of the components of RevRec, the reviewers expertise ($\text{RevRec}^{\text{RE}}$), and the reviewers collaboration ($\text{RevRec}^{\text{RC}}$). To assess the effectiveness of $\text{RevRec}^{\text{RE}}$, we set up $\alpha = 1$ and $\beta = 0$ in the fitness function (Equation 8), and inversely for the $\text{RevRec}^{\text{RC}}$ component (i.e., $\alpha = 0$ and $\beta = 1$). To this end, we use our evaluation metrics precision@k, recall@k, and MRR.

To answer $\mathbf{RQ_3}$, 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 *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 $minRev$ is set to 1, and $maxRev$ is set to 10. We set the parameters α and β to 0.5 as default parameters. Note that RevRec allows the

TABLE II: Parameter settings used for GA, SA and PSO.

Algorithm	Parameter	Value
GA	population size	200
	crossover probability	0.85
	mutation probability	0.1
	number of crossing points	1
	selection	Roulette selection
SA	initial temperature	100
	final temperature	0.157
	cooling coefficient	0.98
	number of iterations	25
PSO	number of particles in a swarm	200
	acceleration coefficient c1	2
	acceleration coefficient c2	2

TABLE III: The precision and recall results achieved by RevRec, cHRev, REVFINDER, and ReviewBot, for each system.

System	k	precision@k				recall@k			
		RevRec	cHRev	REVFINDER	ReviewBot	**RevRec**	cHRev	REVFINDER	ReviewBot
Android	auto	0.52	0.38	0.24	0.17	0.54	0.42	0.38	0.18
	1	0.58	0.50	0.34	0.21	0.38	0.27	0.18	0.11
	3	0.47	0.35	0.25	0.17	0.51	0.50	0.39	0.19
	5	0.39	0.30	0.22	0.12	0.61	0.61	0.48	0.29
	10	0.34	0.26	0.18	0.09	0.71	0.65	0.54	0.38
OpenStack	auto	0.54	0.41	0.26	0.19	0.56	0.40	0.31	0.22
	1	0.59	0.48	0.32	0.24	0.41	0.31	0.15	0.12
	3	0.51	0.42	0.27	0.20	0.54	0.39	0.29	0.20
	5	0.43	0.38	0.25	0.16	0.61	0.52	0.37	0.32
	10	0.36	0.31	0.21	0.11	0.74	0.66	0.46	0.39
Qt	auto	0.46	0.39	0.26	0.18	0.51	0.45	0.31	0.22
	1	0.49	0.45	0.30	0.22	0.41	0.33	0.14	0.9
	3	0.45	0.40	0.27	0.19	0.50	0.47	0.27	0.16
	5	0.41	0.37	0.21	0.13	0.59	0.52	0.35	0.24
	10	0.34	0.31	0.16	0.09	0.65	0.60	0.43	0.30

review submitter to easily set these parameters according to his preferences.

E. Statistical Test Methods

We used the Wilcoxon signed rank test in a pairwise fashion [31], [32] in order to detect significant performance differences between the search algorithms GA, SA, PSO, and RS under comparison. 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. We set the confidence limit, α, at 0.01.

Moreover, we investigate the effect size using Cliff's delta (d) [33]. The effect size is considered: (1) negligible (N) if $\mid d \mid < 0.147$, (2) small (S) if $0.147 \leq \mid d \mid < 0.33$, (3) medium (M) if $0.33 \leq \mid d \mid < 0.474$, or (4) large (L) if $\mid d \mid \geq 0.474$. The goal of these tests is to assess whether such compared results are statistically better than each other, to cope with the stochastic nature of such analyzed data [28].

V. Study Results

This section reports the analysis of the results for the three research questions formulated in Section IV-A.

A. RQ_1: Accuracy of the proposed approach

Table III reports the precision@k and recall@k of our approach RevRec compared to cHRev, REVFINDER, and ReviewBot, for each studied system. The experiment shows that RevRec achieves a precision@1 at 0.58, 0.59 and 0.49 for Android, OpenStack and Qt, respectively. Lower precision scores are obtained with top-10 but still higher than 0.34 for all systems. For the three systems, we observe that our approach RecRec achieved significantly superior precision values compared to cHRev, REVFINDER and ReviewBot.

In terms of recall@k, RevRec achieved the top-10 recall scores at 0.71, 0.74 and 0.60, for Android, OpenStack and Qt, respectively. The lowest recall scores were at $k = 1$, but still higher than 0.51 in worst case for Qt project. Clearly, the recall results achieved by RevRec are considerably superior than cHRev, REVFINDER and ReviewBot. We also observe that the accuracy performance of RevRec is consistent across the three studied systems.

Furthermore, results show that RevRec achieved significantly superior precision and recall results when k is automatic (auto), i.e., the default set of recommended reviewers without ranking. Indeed, this an interesting feature of the RevRec approach to recommend a set (number) of reviewers, as reviewer assigners are unlikely to look through a long recommendation list, and manually find reviewers combination.

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

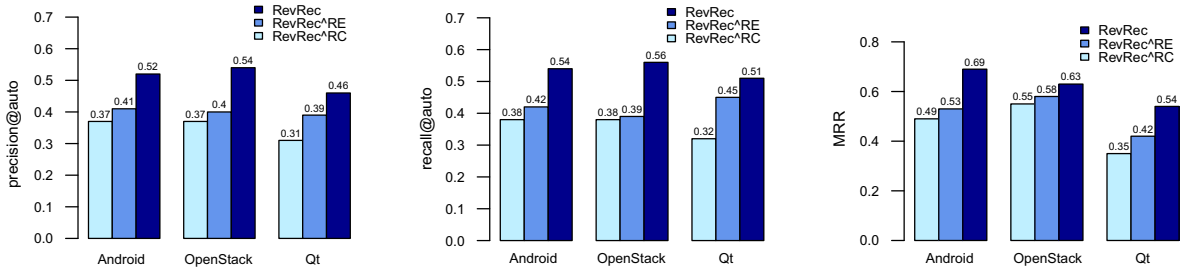
System	RevRec	cHRev	REVFINDER	ReviewBot
Android	0.69	0.65	0.60	0.25
OpenStack	0.63	0.58	0.55	0.30
Qt	0.54	0.43	0.31	0.22

Table IV reports the Mean Reciprocal Rank (MRR) reflecting the overall ranking performance. As shown in the table, RevRec achieves a MRR score of at least 0.54 for all the benchmark systems. This indicated that our approach provides a higher chance of recommending appropriate reviewers in the first ranks, than cHRev, REVFINDER, and ReviewBot.

This superiority of RevRec has an actionable finding supporting the claim that combining reviewers expertise and reviewers social relationships is beneficial for accurately recommending peer reviewers.

B. RQ_2: Effectiveness of each of the reviewers expertise and collaboration

To investigate the effectiveness and the contribution of each of the reviewers expertise and reviewers collaboration heuristics, we assess the accuracy of each of them individually with $k = auto$. Figure 5 reports the results of each individual component $\text{RevRec}^{\text{RE}}$ and $\text{RevRec}^{\text{RC}}$. We observe that the $\text{RevRec}^{\text{RE}}$ component performs slightly better than $\text{RevRec}^{\text{RC}}$ in terms of precision, recall and MRR. Clearly, both heuristics are lower than the RevRec formulation. Interestingly, we found that based only on social relationships (collaboration) without reviewers expertise, we are able to recommend code reviewers with an acceptable precision and recall score, more than 0.31 and 0.39, respectively. Thus, based on the results of Figure 5, we can conclude that combining both expertise and collaboration is an appropriate formulation of the code reviewers recommendation problem, as it improves precision, recall and MRR.



(a) Precision. (b) Recall. (c) MRR.

Fig. 5: The precision, recall and MRR results achieved by each of the RevRec, RevRec$^{\mathrm{RE}}$ and RevRec$^{\mathrm{RC}}$.

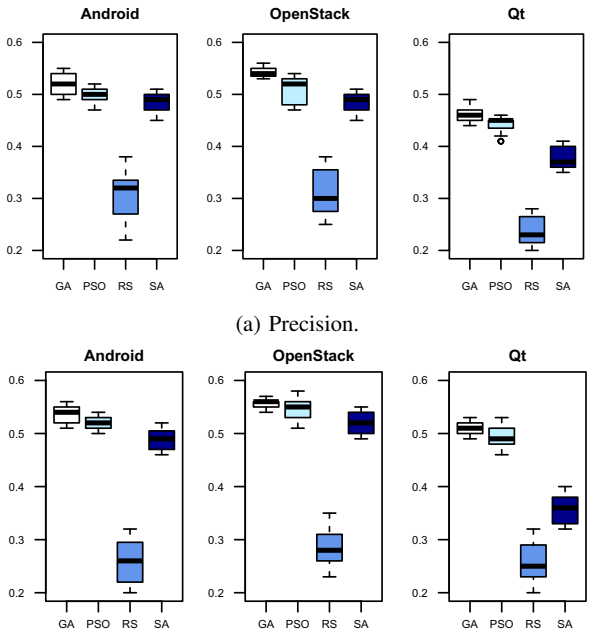
An important observation to highlight from the results of Figure 5 and Tables III and IV is the importance of social relationships in peer code review. Results show that RevRec$^{\mathrm{RE}}$ achieves comparable results to cHRev for all studied systems, but considerably better performance is achieved 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$^{\mathrm{RE}}$ and RevRec$^{\mathrm{RC}}$, 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. RQ_3: 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 $\times$ 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



(a) Precision.

(b) Recall.

Fig. 6: Boxplots of precision an recall results achieved by GA, PSO, SA and RS through 31 independent runs.

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

TABLE V: Statistical tests results of GA *vs*. PSO, SA and RS.

System	GA vs	Precision		Recall	
		p-value	d	p-value	d
Android	PSO	<0.01	0.43 (M)	<0.01	0.52 (L)
	SA	<0.01	0.74 (L)	<0.01	0.92 (L)
	RS	<0.01	0.96 (L)	<0.01	0.96 (L)
OpenStack	PSO	<0.01	0.88 (L)	<0.01	0.40 (M)
	SA	<0.01	0.96 (L)	<0.01	0.94 (L)
	RS	<0.01	0.96 (L)	<0.01	0.96 (L)
Qt	PSO	<0.01	0.58 (L)	<0.01	0.66 (L)
	SA	<0.01	0.96 (L)	<0.01	0.96 (L)
	RS	<0.01	0.96 (L)	<0.01	0.96 (L)

N: *negligible* S: *small* M: *medium* L: *large*

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, α and β, between our components, $\text{RevRec}^{\text{RE}}$ and $\text{RevRec}^{\text{RC}}$, 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, theretofore 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 collegues 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

[1] "Ieee standard for software reviews," *IEEE Std 1028-1997*, pp. i–37, March 1998.

[2] A. F. Ackerman, P. J. Fowler, and R. G. Ebenau, "Software inspections and the industrial production of software," in *Symposium on Software Validation: Inspection-testing-verification-alternatives*, 1984, pp. 13–40.

[3] A. Ackerman, L. Buchwald, and F. Lewski, "Software inspections: an effective verification process," *IEEE Software*, vol. 6, no. 3, pp. 31–36, may 1989.

[4] P. C. Rigby and C. Bird, "Convergent contemporary software peer review practices," in *9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, 2013, pp. 202–212.

[5] V. Balachandran, "Reducing human effort and improving quality in peer code reviews using automatic static analysis and reviewer recommendation," in *35th International Conference on Software Engineering (ICSE)*, 2013, pp. 931–940.

[6] E. S. Raymond, *The Cathedral and the Bazaar*, 1st ed. Sebastopol, CA, USA: O'Reilly & Associates, Inc., 1999.

[7] S. McIntosh, Y. Kamei, B. Adams, and A. E. Hassan, "The impact of code review coverage and code review participation on software quality: A case study of the qt, vtk, and itk projects," in *11th Working Conference on Mining Software Repositories (MSR)*, 2014, pp. 192–201.

[8] A. Bacchelli and C. Bird, "Expectations, outcomes, and challenges of modern code review," in *35th International Conference on Software Engineering (ICSE)*, Piscataway, NJ, USA, 2013, pp. 712–721.

[9] A. Bosu and J. C. Carver, "Impact of peer code review on peer impression formation: A survey," in *International Symposium on Empirical Software Engineering and Measurement*, 2013, pp. 133–142.

[10] P. C. Rigby and D. M. German, "A preliminary examination of code review processes in open source projects," *University of Victoria, Tech. Rep. DCS-305-IR*, 2006.

[11] P. C. Rigby and M.-A. Storey, "Understanding broadcast based peer review on open source software projects," in *33rd international conference on Software engineering (ICSE)*, New York, New York, USA, may 2011, p. 541.

[12] O. Kononenko, O. Baysal, L. Guerrouj, Y. Cao, and M. W. Godfrey, "Investigating code review quality: Do people and participation matter?" in *International Conference on Software Maintenance and Evolution (ICSME)*, 2015, pp. 111–120.

[13] T. Patanamon, T. Chakkrit, G. K. Raula, Y. Norihiro, I. Hajimu, and M. Ken-ichi, "Who Should Review My Code? A File Location-Based Code-Reviewer Recommendation Approach for Modern Code Review," in *22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, 2015.

[14] M. Zanjani, H. Kagdi, and C. Bird, "Automatically recommending peer reviewers in modern code review," *IEEE Transactions on Software Engineering*, vol. PP, no. 99, 2015.

[15] A. Bosu and J. C. Carver, "How do social interaction networks influence peer impressions formation? A case study," in *Open Source Software: Mobile Open Source Technologies - 10th International Conference on Open Source Systems (OSS)*, 2014, pp. 31–40.

[16] X. Yang, N. Yoshida, R. G. Kula, and H. Iida, "Peer review social network (person) in open source projects," *IEICE TRANSACTIONS on Information and Systems*, vol. 99-D, no. 3, pp. 661–670, 2016.

[17] J. Cohen, E. Brown, B. DuRette, and S. Teleki, *Best kept secrets of peer code review*. Smart Bear, 2006.

[18] J. H. Holland, *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence.* U Michigan Press, 1975.

[19] M. Beller, A. Bacchelli, A. Zaidman, and E. Juergens, "Modern code reviews in open-source projects: Which problems do they fix?" in *11th Working Conference on Mining Software Repositories (MSR)*, ser. MSR 2014, 2014, pp. 202–211.

[20] M. Harman, S. A. Mansouri, and Y. Zhang, "Search-based software engineering: Trends, techniques and applications," *ACM Computing Surveys (CSUR)*, vol. 45, no. 1, p. 11, 2012.

[21] M. Harman and B. F. Jones, "Search-based software engineering," *Information and Software Technology*, vol. 43, no. 14, pp. 833–839, 2001.

[22] M. Harman, "The current state and future of search based software engineering," in *Future of Software Engineering (FOSE)*, 2007, pp. 342–357.

[23] M. Mitchell, *An Introduction to Genetic Algorithms*. Cambridge, MA, USA: MIT Press, 1996.

[24] X. Yang, R. Gaikovina Kula, N. Yoshida, and H. Iida, "Mining the modern code review repositories: A dataset of people, process and product," in *13th Working Conference on Mining Software Repositories (MSR)*, 2016.

[25] F. J. Solis and R. J.-B. Wets, "Minimization by random search techniques," *Mathematics of operations research*, vol. 6, no. 1, pp. 19–30, 1981.

[26] S. Kirkpatrick, "Optimization by simulated annealing: Quantitative studies," *Journal of statistical physics*, vol. 34, no. 5-6, pp. 975–986, 1984.

[27] J. Kennedy and R. Eberhart, "Particle swarm optimization," in *International Conference on Neural Networks*, 1995, pp. 1942–1948.

[28] A. Arcuri and L. Briand, "A practical guide for using statistical tests to assess randomized algorithms in software engineering," in *33rd International Conference on Software Engineering (ICSE)*, 2011, pp. 1–10.

[29] I. Avazpour, T. Pitakrat, L. Grunske, and J. Grundy, "Dimensions and Metrics for Evaluating Recommendation Systems," in *Recommendation Systems in Software Engineering*. Springer Berlin Heidelberg, 2014, ch. 10, pp. 245–273.

[30] A. E. Eiben and S. K. Smit, "Parameter tuning for configuring and analyzing evolutionary algorithms," *Swarm and Evolutionary Computation*, vol. 1, no. 1, pp. 19–31, 2011.

[31] J. Demšar, "Statistical comparisons of classifiers over multiple data sets," *The Journal of Machine Learning Research*, vol. 7, pp. 1–30, 2006.

[32] J. Cohen, *Statistical power analysis for the behavioral sciences*. Academic press, 1988.

[33] N. Cliff, "Dominance statistics: Ordinal analyses to answer ordinal questions." *Psychological Bulletin*, vol. 114, no. 3, p. 494, 1993.

[34] J. Anvik, L. Hiew, and G. C. Murphy, "Who should fix this bug?" in *28th International Conference on Software Engineering (ICSE)*, 2006, pp. 361–370.

[35] X. Xia, D. Lo, X. Wang, and B. Zhou, "Accurate developer recommendation for bug resolution," in *20th Working Conference on Reverse Engineering (WCRE)*, 2013, pp. 72–81.

[36] M. E. Fagan, "Design and code inspections to reduce errors in program development," *IBM Systems Journal*, vol. 15, no. 3, pp. 182–211, 1976.

[37] P. C. Rigby, D. M. German, and M.-A. Storey, "Open source software peer review practices: a case study of the apache server," in *30th international conference on Software engineering*, 2008, pp. 541–550.

[38] C. Bird, D. Pattison, R. D'Souza, V. Filkov, and P. Devanbu, "Latent social structure in open source projects," in *16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. SIGSOFT '08/FSE-16, 2008, pp. 24–35.

[39] O. Baysal, O. Kononenko, R. Holmes, and M. W. Godfrey, "The influence of non-technical factors on code review," in *20th Working Conference on Reverse Engineering (WCRE)*, 2013, pp. 122–131.

[40] O. Baysal and R. Holmes, "A qualitative study of mozillas process management practices," *David R. Cheriton School of Computer Science, University of Waterloo, Waterloo, Canada, Tech. Rep. CS-2012-10*, 2012.

[41] A. Bosu and J. C. Carver, "Impact of developer reputation on code review outcomes in oss projects: An empirical investigation," in *8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, 2014, pp. 33:1–33:10.